MOPping up Exceptions

S. E. Mitchell, A. Burns and A. J. Wellings,

Department of Computer Science, University of York, UK

Abstract: This paper describes the development of a model for the reflective treatment of both application and environmentally sourced exceptions. We show how a variety of exception models can be implemented using an exception handler at the metalevel. The approach described allows for better separation of exceptional and normal error-free program code producing systems that are easier to understand and therefore maintain.

Keywords: metalevel architecture, reflection, exceptions.

1 Introduction

The principle reason for the inclusion of exception handling mechanisms in programming languages is a desire to separate error handling from the programs normal operation [Burns and Wellings, 1996]. This is in line with the justification for the use of reflection in system design – in that its use aids in the separation of concerns.

Despite this desire for a separation, in many approaches the handler code is still intermingled with application code, albeit moved to the end of a program block. In this paper we explore the use of reflective principles to complete the separation of concerns by attempting to operate on exceptions at the metalevel.

The remainder of this introduction describes the exception model that we have used as the basis for our work and explores the motivation for producing a reflective treatment of exceptions. The second section describes two variations of the model, the first in Section 2.1 considers a model where the raising of the exception is reified whereas Section 2.2 considers a scenario where the exception itself is the reified entity. Next Section 3 explores further issues with particular emphasis on the relationship between concurrency and reflective exceptions. Finally, we evaluate the effectiveness of the work with respect to the success of reflection in producing the required separation of concerns.

1.1 Non-reflective Exception Models

The exception model that we explore within this paper is derived from the common non-reflective one of exceptions being raised (thrown) and subsequently (possibly) caught within a separate section of code (termed the *exception handler)* usually at the end of an *exception block*. Exceptions that are not caught by any exception handler currently within scope are propagated back up to the previous level – the caller – and the search for a handler repeated. This model follows that used in many current object-oriented and procedural languages such as Ada95 [Intermetrics, 1995] and C++ [Stroustrup, 1997]. Both Ada95 and C++ use the *termination model* of exception handling – once an exception has been handled the exception block is terminated and control returns to the containing block or caller as appropriate. The alternative method of processing is the *resumption model* where control is returned to the containing block

after the exception has been handled. This is not as common as the termination model and is not implemented in any 'mainstream' language. Furthermore, the resumption model can be implemented in a number of flavours – re-execute from the start of the exception block, re-execute the instruction which caused the exception or resume execution at the instruction following the "faulty" instruction.

Within a program it is possible, and important, to distinguish between two different types of exception depending on their source:

- **Environmental** – The exception is raised by an entity (e.g. the run-time system) outside the current program due to an event occurring in the environment within which the program is running, for example, a floating point error raising the `CONSTRAINT_ERROR` exception in an Ada95 program.
- **Application** – The exception was defined within the program code and is raised as a result of a program action (or inaction). For example, an exception raised as the result of a failed application assertion.

In addition, either type of exception can be *synchronous* or *asynchronous.* A synchronous exception is raised as a direct result of current program activity, e.g. resource locked, whereas an asynchronous exception is not related to the *current* program activity though the exception may have been raised as a result of past activity or current in-activity. Note that for application exceptions the asynchrony can be viewed as the insertion of an arbitrary delay between the exception being raised and the subsequently being handled

1.2 Reflective Programming Model

Within this paper we aim to produce a reflective treatment of exceptions and thus follow the usual reflective object-oriented language model [Watanabe and Yonezawa, 1988; Maes, 1987] inorder to develop the most general result. Within this general reflective architecture, the structure and behaviour of an object is controlled by its metaobject, which itself is controlled by a meta-metaobject. Unlike many reflective object-oriented languages, we allow a base-level object to have *multiple* metaobjects each responsible for some aspect of structure and/or behaviour of the base-level object.

1.3 Motivation

Exception handling can be viewed as providing error containment for an object and helps to prevent errors propagation within the system. The issues related to exception handling within an object-oriented context have been considered by a number of authors, notably [Miller and Tripathi, 1997]. However, we are not aware of any work that addresses issues related to reflection and exceptions. In a reflective system, the metalevel reifies (makes concrete in terms of actually implementing) the abstractions used by the base-level. We thus view exceptions as implementing the abstraction of fault-free objects/modules within a system and that using a more explicit reflective model would enable us to exploit the further advantages offered by reflection. These are:

- **A strong, disciplined separation of concerns** – Our initial motivation for this work stemmed from a belief that the separation of concerns offered by a reflective treatment of exceptions could provide a useful extension of their use to divide functional and error-handling code. In addition, we want to be able to support a wider range of features (e.g. different exception models) than can be accommodated in a non-reflective language. A reflective system permits an effective discipline to be imposed on the separation since the metalevels can

impose the required restraints on change within the system.

- **Transparency and self-containment** – These refer to the principles that the base-level is not reliant on the metalevel for correct *functional* behaviour and that the facilities implemented by the metalevel are transparent to the base-level. Viewed from the perspective of an application, facilities reified by the metalevel are seen as not relevant to the provision of correct functional operation. Note that system requirements, e.g. fault tolerance or security, may well be implemented at the metalevel. However, if they are not available the application will still operate except that it would execute without the safety net of the fault-tolerance or security layers insulating the application from an unreliable insecure execution environment.
- **Recursive** – The metalevel reifies abstractions used at the base level and the recursive nature of the reflective model means that the meta-metalevel reifies abstractions used at the metalevel. This principle means that we seek consistency of programming language entities at each meta-level.

2 Exceptions and Reflection

This section describes the development of our first model of reflective exceptions. The basic principle of operation is that the act of raising of an exception becomes a *reified operation*. After briefly presenting the model we will discuss why such an approach is not adequate and show how the basic model can be changed to enhance the separation between functional and exceptional code.

2.1 Reifying the raising of an exception

The first reflective model of exception handling we shall consider uses a computational model with two distinct entities; objects and exception handlers. An object throws an exception which is subsequently processed by an exceptions handler. Reflection is used to reify the act of raising/throwing the exception; thus when an exception is raised, a jump to the metaobject occurs that then handles exception through the exception handler along with any other processing undertaken by the metalevel. This procedure is analogous to the reification of a method call that causes an interception of the call to the base-level object with subsequent processing controlled by the metaobject.

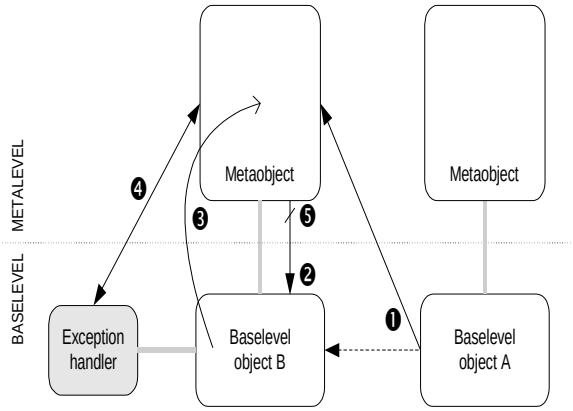

Figure 1: Exception handling with reified exception raise.

Figure 1 illustrates how an exception is raised and handled within this system. The flow of control proceeds as follows:

1. Base-level object “A” invokes a method in “B”. The invocation is trapped by the metalevel.
2. The metaobject invokes an appropriate method in the base class in response to the invocation request from “A”.

3. At some point an exception is raised which, since this is a reified operation, causes a jump to metaobject computation.
4. The metaobject invokes the selected handler which may access/modify the state of "B" to resolve the exception. The handler returns information to the metaobject indicating the success or otherwise of the exception handling.
5. The metaobject decides on the fate of the method invocation request from "A". In this example it terminates the original call at point ❺ (indicated by the line crossing the invocation arrow).

2.1.1 Exception Propagation

If the exception cannot be processed (e.g. there is no handler present) then the metaobject must propagate the exception to the invoker's meta-object with possible subsequent termination of the method invocation that raised the exception. The call to the base-level object which actually raised the exception should be terminated irrespective of the exception termination model currently in use – after propagation, even for the resumption case, the invoker object is the one to be resumed. This model is illustrated in Figure 2.

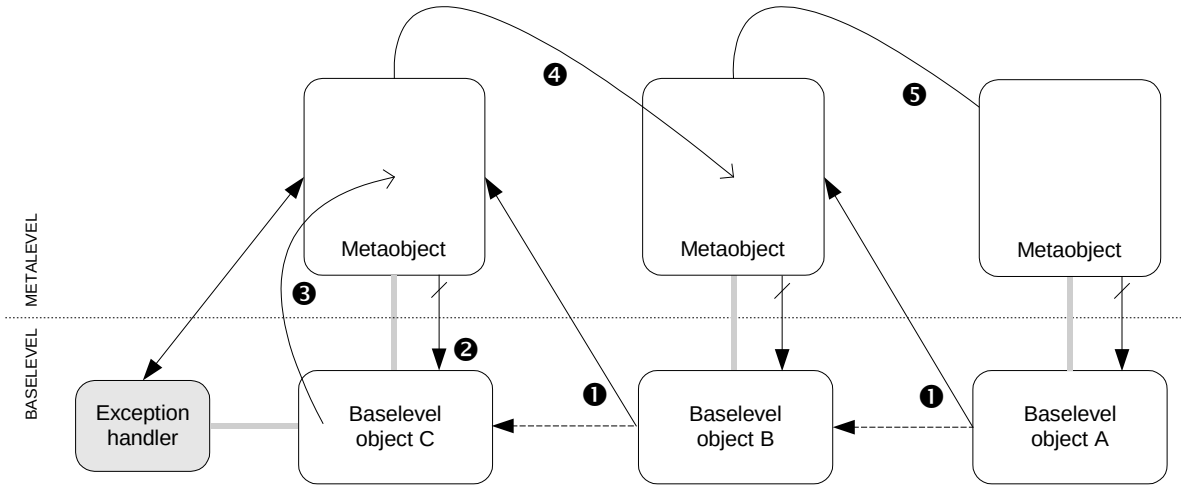

Figure 2: Termination model using reified exception raise.

As in Figure 1, the base-level object "A" invokes "B" (❶) through metaobject ↑B, but in this example the method at the base-level then invokes "C" through ↑C (also labelled ❶). The method invoked by ↑C in the base-level object raises an exception (❸) and the reification of the raise causes processing to proceed in the metaobject. The exception handler selected by the metaobject fails to handle the exception so the method invocation in the base-level object is terminated by the metaobject (❷) and the exception propagated to ↑B (❹). There is no handler associated with ↑B and so the exception is propgated again to ↑A (❺) and the call to "B" terminated. In ↑A there is also no handler – and nowhere to propagate to – and so the program must also terminate with an "Unhandled Exception" error.

The model can readily be adapted to the *non-termination* case by a change in semantics of the metaobject which causes control to return to base-level object B, at the point where the call to object C occurred, after the exception has been handled by the metaobject. One may also need to change

the behaviour of application object B to retry the operation perhaps using a replicated object C. An important advantage over conventional exception mechanisms is that the metaobject controls the behaviour and syntax of the base level. Thus the metalevel exception handler can *modify* the base level (e.g. change the base-level state) before returning control and thus prevent the exception being re-raised.

If a handler for a synchronous application exception does not exist within the metaobject, the propagation continues at the metalevel, a process that is illustrated in the figure above. If no handler is found at the caller's metaobject then that method invocation is also terminated and the propagation repeated.

2.1.2 Environmental Exceptions

The class of exceptions that can be raised by entities outside the current program are said to have an *environmental* cause. This section is split into two subsections dealing with *asynchronous* and *synchronous* environmental exceptions respectively.

2.1.2.1 Asynchronous Environmental Exceptions

An asynchronous environment exception (AEE) is raised as a result of an event external to the program and not as a direct result of current program activity (though it may be raised as a result of current program *inactivity).* As examples, a run-time system may raise a garbage "low memory" exception to cause each object in the system to release resources or a health monitor may signal that a patient requires immediate attention.

Within our model, AEEs are raised immediately at the metalevel without being reified entities from the base-level. Since reflective components may process in parallel with the base-level components then there is no fundamental requirement to stop the base-level from executing its current task, however, since this is dependent on the semantics defined at the meta-level this is a design option. Certain classes of AEE, such as the garbage collection exception, may never require the base-level to be halted whereas others may require interruption and a subsequent change in the flow of control at the base-level. Note that even if the model is such that the base level is interrupted when an environmental exception occurs, the base level may subsequently be resumed without any changes after the exception has been processed or alternatively, a whole section of the program may be abandoned.

As an example of how AEEs might interact with the meta-level, consider a base-level object that makes a *non-blocking* attempt to access a resource (e.g. communication channel). At the point where the attempt is made the resource is unavailable and so a *future*[1] is returned to the base-level. Whilst the base-level continues processing, the environment raises an AEE in all meta-level objects that results in the freeing of the resource from another object. A further AEE is raised in the base-level object's meta-object which allocates the resource to the future assigned to the base-level. When the base-level attempts to access the future it can use the resource without ever being aware of meta-programs allocation attempts.

2.1.2.2 Synchronous Environmental Exceptions

Synchronous environmental exceptions are exceptions that are raised by the

[1] A *future* is reference to a resource that is returned to a program by the run-time system when the resource is not immediately available. It enables the requesting program to continue to perform work whilst waiting for the resource to become available. If the future is *sticky* then further attempts to access the contents before the resource is available will cause the program to block pending allocation otherwise access attempts result in a further 'not-available' indication

environment as a result of current program activity and so may require handling in different manner to asynchronous environmental exceptions.

A synchronous environmental exception is raised by an external entity as a result of actions or in-actions taken by the current program. In this manner we can consider an environmental exception as a hidden program assertion in that if the exception is raised then the assertion can be said to have failed. Since these exceptions are raised as a result of evaluating the hidden assertion it is clear that the exception be raised by the environment at the base-level and then treated as a synchronous application exception (i.e. reified and then handled at the meta-level). After the exception has been handled the system will then terminate the exception block or resumes execution according to the particular exception model.

2.1.3 Discussion

By the simple act of moving the *control* of exception handling to the metalevel confers a major advantage over a conventional, non-reflective, model. Significantly, it enables the metaobject to perform work both pre- and post- handler execution – for example to check that an exception handler has performed correctly and that object invariants hold after its execution. Such checks can then be applied to all exceptions raised in the object providing obvious advantages of single point of control.

The choice of handler for a particular exception is made at run-time and can vary both from object to object and also during program execution. In addition the action taken by the metaobject after an exception has been raised but prior to invoking a particular handler can vary, for example it could:

1. Return the exception to the caller without attempting further processing or error recovery. This may be useful for exceptions for which no state restoration is required in the base-level object or a fast response is required.
2. Restore the base-level object's state (e.g. through roll-back) and attempt re-invocation of the method. If the subsequent invocation goes to a different method then the handler has effectively implemented an *acceptance block.*
3. Abandon the method invocation and delegate subsequent handling of the invocation that caused the exception to a replicated instance of the base-level object whilst, in parallel, restoring the now faulty primary.

The decision as to which action to pursue need not be fixed – the metaobject can examine the system self-representation to determine the optimal method of exception handler. This run-time dynamism can extend to the modification of handler by metalevel to process new exceptions or existing exceptions in new ways. The use of run-time reflection means that change in the system can occur dynamically at any point, however, within a reflective system the disciplined "opening up" of the run-time system means that the system designer can constrain change to occur only at specified points. Restricting the reflection mode to compile time constrains the degree of change that can be applied to a system and thus permits only re-configuration when the system is built.

Unfortunately, this model does not enable a simple change from the resumption to termination exception models since the application will have been programmed with certain assumptions in mind. These assumptions cannot be changed by simply altering how exceptions are handled – one must also alter how the exception handler behaves and how the calling program behaves. Whilst changing the base-level

program to reflect these changed assumptions is entirely possible within a reflective system is constitutes a major cost.

2.1.3.1 Location of Exception handler

Within this system, the exception (i.e. the point where an exception is raised and the exception handlers) are located at the base-level but the exception control is located at the metalevel. Thus we have only achieved a limited separation of concerns since both handlers and functional objects exist at the same level and only the exception control has been reified. Also we have been forced to introduce a new language entity, the *exception handler* which further weakens our aim of developing a fully recursive model.

Recall that our principle aspiration was to make the separation of functional code and error code more explicit, and therefore we surmise that moving the location of the exception handler(s) to the metalevel will enable us to enhance the separation of concerns. However this leaves open whether the exception handler should be:

1. Integrated as part of the base-level object and invoked by the metaobject,
2. Located in separate object at the metalevel (though not itself a metaobject) to which exception handling is delegated by the metaobject,
3. Located as part of the base-level object's metaobject,
4. An alternative architecture which does not use 'exception handlers' as they are commonly known.

The first approach is identical to the one described above and would therefore suffer from the same problems regarding lack of transparency and separation of concerns. The second is appealing since it is highly flexible (for example, the destination of the delegation can be changed at run-time and thus the system designer easily alter the error treatment semantics) and enables different metaobjects to share exception code. However the third approach is in fact equally flexible since reflection allows both the syntax and behaviour of the metaobject (in this case, the structure and behaviour of the embedded exception handler) to be modified through method invocation in the meta-metalevel.

However, the fourth scheme enables us to combine the advantages of separate handlers without introducing new language entities through the use of an alternative exception architecture where the exception itself is reified and responsible for its own handling. We thus consider this to be the most promising and is the subject of the next section.

2.2 Reifying Exceptions

The previous section considered how to map the common exception models into a reflective framework through the reification of the *raising* of the exception. As a result, the clear separation of functional and exceptional code was compromised with a mixing of responsibilities between base and metalevels. This section explores the use of an alternative architecture where the exception itself controls the handling and will enable the metaobject to modify the exception prior to handling or propagation. We will show how this architecture produces a more disciplined separation without introducing any new entities into the computational model.

First consider that an exception is an object (as in, for example, C++) with the extension that this object is *reified* and therefore has a associated metaobject. Thus when an *exception object* is created, a jump to metalevel processing will occur. This is in contrast to the previous discussion where the act of raising the exception was reified and therefore caused the jump to metalevel

processing. We shall term the metaobject associated with each exception object when it is created (that is raised) the *metaexception object.*

The role of the exception object is not only to provide the link to the metalevel but also to maintain state about the exception which is not part of the base-level object's state. Note that the latter may well be replaced/modified by exception handler and thus we need a place to retain information during the exception handling.

Within this model we view the base-level's computation as follows; the point where an exception can be raised is considered to be a program assertion which will be true when executed. If the assertion fails, i.e. is *false*, then the reification process causes the metalevel to take whatever action is required to make the assertion evaluate to *true*. From the applications viewpoint the assertions are always true. This approach maintains the transparency requirement – if the base-level is running without metalevel support then it will continue as long as assertions are *true* and will simply not benefit from corrective actions.

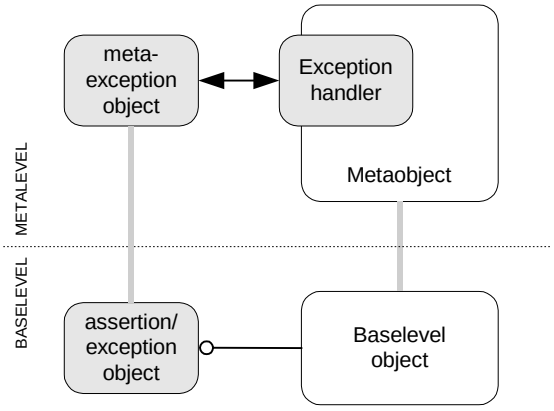

Figure 3: Object diagram for the raising of a reified exception.

The metaexception object operates in exactly the same way as any reified metalevel entity in that it controls both the structure and behaviour of the exception at the base level. When an exception is raised, an exception object is instantiated with an appropriate metaexception object. The metaobject is then responsible for managing exception resolution or propagation. This process is illustrated in Figure 4.

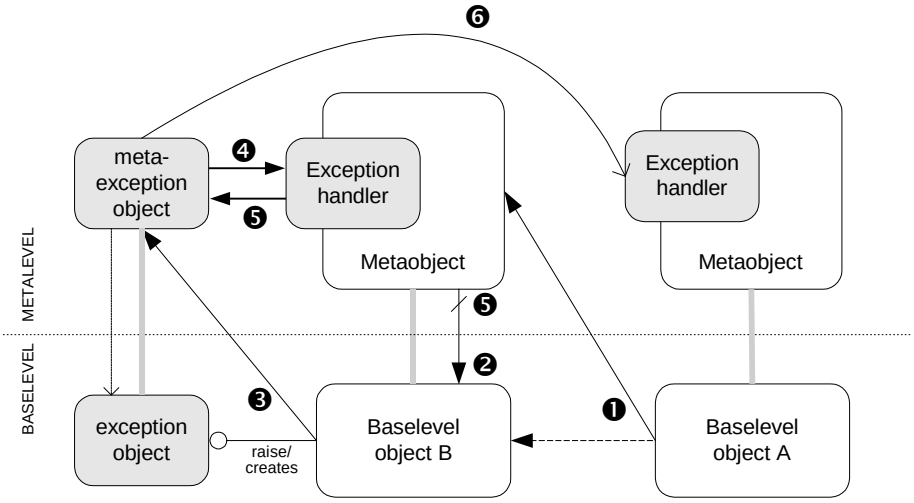

Figure 4: Exception handling and propagation with reified exceptions

Figure 4 shows a complete flow of control for reified exception handling using the termination model. The six stages are as follows:

1. Base level object A invokes a method in object B. The message receipt is intercepted by the metaobject.
2. The metaobject invokes the requisite method script.
3. An assertion fails and an exception object is created. The executing method script blocks. The exception object creation is reified and so control passes to the metalevel metaexception object.
4. The metaexception object controls the *semantics* of exception handling. In this case it invokes routines in a handler in the metaobject. The handler(s) at the metalevel have full access to the state of the baselevel object through the reflective self-representation and so can attempt to rectify the cause the of the exception before returning control. The metaexception object has access to any state stored in the exception object. Note that since the metaobject can execute in parallel with the base level object it can continue to process information even though the latter is blocked after creating the exception object.

   After an exception has been successfully handled control is returned to the object that created the exception object. Since the metalevel will have altered the object state to resolve the problem continued execution will not immediately result in repetition of the error.
5. However, in this case the exception cannot be processed and the handler returns a failure message to the metaexception object along with a reference to the invoking object/metaobject pair. Simultaneously, the ongoing method call is terminated by the metaobject.
6. The metaexception object repeats the attempt to handle the exception with the handler(s) at the onvoking metaobject. If, after all possible propagation attempts have occurred the exception remains unprocessed then the metaexception object is responsible for handling and may terminate the program.

This change enables us to take the model described in previous sections and extend it by allowing the *exception to control its future*. For example, a system could be created in which raised exceptions are by default bound to a metaexception object that implemented the termination model. However, at run-time when an exception is raised it may prove possible to successfully handle of the cause of the exception (e.g. re-issue a prompt to receive further input from the user) and thus the metalevel exception handler could invoke methods in the metaexception object to change the semantics to implement the resumption model. Should the system designer wish to prohibit such change for a particular exception or class of exceptions then it is a simple matter of binding to a different metaobject which will not accept such invocations and so does not permit the semantics to change.

The metaexception object is responsible for the controlling the handling of the exception object either through delegation to a separate handler (e.g. one located in the metaobject as in the figures above) or through its own methods if no handler can be found.

2.2.1 Discussion

The approach of reifying exceptions provides the full and complete separation of concerns that we were seeking. It is also a very flexible model – one can change the exception handling semantics at run-time

and this can be on a per-exception basis so that different exceptions raised by the same object can have different semantics. The desired separation of concerns arises because both exceptional and functional code are completely orthogonal – the latter exists at the base-level where as the exception code exists at the metalevel and manipulates the reflective self-representation of the base-level.

This approach has the advantage of introducing no new language or conceptual entities into the system since exceptions are base level objects whose semantics are handled by meta-level entities. The handlers for exceptions are simply methods of the metalevel object which are manipulated by the meta-metalevel to change either the semantics of how exceptions are handled (e.g. termination or resumption) or to change the actions necessary to recover from an exceptional state. Such a high degree of run-time flexibility is necessary since we want the model to be able to handle exceptions for any methods which are created at run-time and thus we need to be able to insert new handlers or modify existing ones on the fly.

From a metalevel viewpoint, one can view exception handlers as minimal recovery blocks – simple sections of code to restore the state of an object to a valid state – and that the metalevel then decides what action to take; invoke a replica, repeat the operations etc. Such a system would then produce an exception model similar to that in Eiffel [Meyer, 1992] for exceptions raised as a result of a failed pre- or post-condition. In Eiffel, after an exception handler has restored the state of an object it can redirect the exception using a "`retry`" statement.

The model as described is, of course, fully recursive in that exceptions created at the meta-level will have meta-meta-level objects that control their semantics. Thus it fits with the recursive reflective computational model. It is also transparent – the base-level program will proceed (in the absence of errors) in the same fashion with or without the metalevel.

Another significant advantage is that, unlike the previous model, the reification of the exceptions itself makes possible the handling of exceptions raised as a result invocations where the caller is a non-reified object (i.e. it does not have a metaobject). With the previous model we needed to make these a special case since no metaobject was available to continue the propagation. However, the actions in this model are controlled by the metaexception object which can unwind the call chain through both objects and metaobjects until a suitable handler is found.

We retain the benefit of multiple exception handlers (since the metaexception object can invoke any present handler in the metaobject) and also promote the re-use of handlers. Handlers can be re-used (that is imported from other metaobjects) by the meta-level and this is of particular significance where we have a hierarchy of exceptions since the handler will may only need to be written once and automatically imported as the "default" handler by the metalevel without further intervention by the system designer.

3 Concurrency

3.1 The Concurrency Model

We have adopted a concurrency model based around active objects as we feel that such a scheme has both practical applications as well as fitting naturally within a reflective framework. Each active object possesses one or more embedded threads that are created when the object is created and execute independently. Consequently, all objects within a system (whether active or passive) are multi-threaded and thus require synchronisation to constrain access to shared data. Similar

models have been adopted by a number of concurrent object-oriented languages, for example, TAO [Mitchell and Wellings, 1996; Mitchell, 1995].

The model of reflective object-oriented system permits objects to have multiple metaobejcts and to delegate aspects of the control of their behaviour or state to different metaobjects. We use distinct metaobjects to separate the metacontrol of concurrency from the normal metacontrol of an object into a *metatask* object. This object controls the behaviour of the thread embedded in the base-level object and is also responsible for initiating thread execution and implementing context switches.

In keeping with the design of the base-level object that contains an embedded thread to form the active object, the controlling metatask object is embedded within the relevant metaobject. This model is illustrated in Figure 5 shows a single base-level and metalevel object pair with arrows showing the actions of the metatask, namely, the initial start of thread execution, one or more context switches and finally the thread termination.

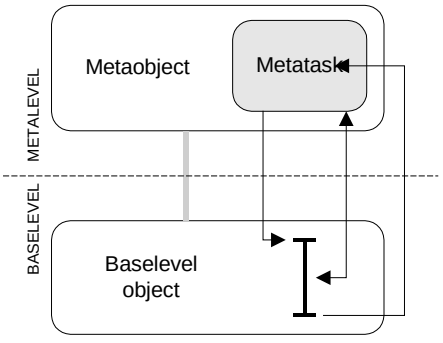

Figure 5: Illustration of the concurrency model of objects with embedded threads and metaobjects with embedded metatasks. The metatask controls thread creation, context switch and termination.

The above model is derived from the concurrency model used in TAO, however, in adopting this model we do not intend to limit the applicability of our work and consider that our results can be usefully used with a wide range of concurrency models. For example the Ada95 model has a “task” construct which is separate from objects and in such a case metaobjects and metatasks would also be separate entities rather than embedded within the same object.

3.2 Reflective Exceptions and Concurrency

Within a concurrent computational model which uses either an active object model or a task based model, the threads can operate independently of any ongoing method calls. Thus it is not necessary to terminate embedded threads when an exception cannot be handled within the metaobject. However, the exception handler at the metalevel will need to co-operate with the metatask when an exception is raised by a base-level task.

When a base-level tasks raises an exception, an attempt must be made, as described above, to handle it within the metaobject. Should this exception handling be successful, the task may be able to resume or it may be necessary for the metaobject to cause the metatask to recreate the task (if the raising of the exception had caused it to be terminated) or to cause a context switch to resume the task (resumption model) at the appropriate point.

However, perhaps the more interesting case occurs when the metalevel cannot handle the exception. In this case the task must be terminated – again through cooperation with the metatask object. Embedded threads within an active object are created along with the active object and thus are directly to analogous to the *main* thread of control in many programming languages. Consequently there is no sensible place to propagate the exception to, and like a *main*

program, the active object must now be considered fatally flawed and terminated. It passing is marked, however, by an *asynchronous application exception* being raised in the object which created the active object which will allow it to take appropriate action to correct the error (re-create the object perhaps trying a different implementation to avoid repetition of the exception). Note that due to the autonomous operation of parent and child active objects, the parent active object may no longer exist in which case we need to raise the "notification of termination" exception within the *grandparent* object and thus require a run-time system which tracks task creation hierarchies.

4 Summary and Evaluation

This paper has explored how the implementation of both synchronous and asynchronous environmental and application exceptions can be programmed at the metalevel. This section presents an overall evaluation of the advantages of a reflective approach to exceptions and also a final summary and suggestion for future work. Note that many of the points discussed in this evaluation apply to either of the reflective models discussed in this paper.

The foremost advantage is that the reflective exception mechanism is extremely flexible and adaptive. In [Miller and Tripathi, 1997] the authors devise three categories of evolution of exception; Exception evolution (raising new exceptions derived from ones already in the interface), Function evolution (raising entirely new exceptions not currently in the interface) and mechanism evolution (a change in implementation leading to exception overloading). A reflective system is ideal suited to overcoming these problems due to its in-built mechanisms for introspection and modification which means that new handlers can be added, the semantics of existing ones altered or split into several different handlers.

Within our system, the flexibility arises because the metalevel exception handler has its syntax and semantics defined at the meta-metalevel. This means the designer of the system can control the changes permitted at run-time and thus produce anything from a static system fixed at compile time to a fully flexible, but inherently less predictable system, that can be changed at will to reflect the current environment the program finds itself in. A reflective treatment of exceptions also produces a very adaptable system since the specification of the semantics of the exception handling is separated from the handler itself. The system described in this paper can covers both the termination and non-termination models for both synchronous and asynchronous exceptions whether application or environmentally sourced. The semantics of exception are not necessarily fixed according to the whims of the programming language, designers are free to adapted to suit the needs of a particular program.

This flexibility and adaptability means that different components of a system can have different approaches to how exceptions are handled and can potentially swap amongst them at run-time. If such freedom is undesirable since it would lead to a too unpredictable system, the designer is able to control the degree of change allowed by appropriate programming of the meta-levels. Furthermore, by reifying exceptions, rather than the raising of the exception, one can permit more localised control over the future course of the exception and also change the semantics of exception handling whilst it is being processed. The reified exception model is highly expressive as illustrated by the fact that we can easily implement a wide range of exception models including those in the mainstream languages C++, Ada 95 and Eiffel. The model is also highly flexible in

that the metalevel program that handles the exception can manipulate the behaviour of the base-level program. Using this mechanism the reflective layer can modify the base level after an exception has occurred to prevent it being re-raised – for example, by forcing the base-level program down a different branch in an acceptance block.

Finally, a reflective treatment of exceptions has achieved our primary goal of further enforcing a clear separation of concerns – there is no mixing of the exception handlers in with normal "functional" program code. This paper has shown that even a simple reflective approach where exception control occurs at the metalevel confers a number of advantages. However, we have also shown that extending this model so that both exception control and handling occurs at the metalevel and that the base-level simply views exceptions as assertions can simplify programs through ensuring a strong, disciplined, separation of functional and exceptional concerns.

4.1 Acknowledgements

This work has been done as part of the *Design for Validation* (DeVa) project, ESPRIT long-term research project 20072. We would like to acknowledge the contributions that other members of the DeVa project have made towards this work, especially for the comments on the work during its various stages of preparation. Special thanks must go to Dr. A. Romanovsky for the many discussions on exceptions and to Dr. R. J. Stroud for the helpful insights into the principles of reflection.

5 References

[Burns and Wellings, 1996] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, Second ed: Addison-Wesley, 1996.

[Intermetrics, 1995] Intermetrics, "Ada Reference Manual," ISO/IEC 8652:1995, 1995.

[Stroustrup, 1997] B. Stroustrup, *The C++ Programming Language*, Third ed: Addison-Wesley, 1997.

[Watanabe and Yonezawa, 1988] T. Watanabe and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," *ACM SIGPLAN Notices - Proceedings of OOPSLA'88*, (23)11, pp. 306-315, 1988.

[Maes, 1987] P. Maes, "Concepts and Experiments in Computational Reflection," *ACM SIGPLAN Notices - Proceedings of OOPSLA'87*, (22)12, pp. 147-155, 1987.

[Miller and Tripathi, 1997] R. Miller and A. Tripathi, "Issues with Exception Handling in Object-Oriented Systems," in *Proceedings of ECOOP'97*, vol. LNCS-1241, M. Askit and S. Matsuoka, Eds. Jyväskylä, Finland: Springer-Verlag, 1997, pp. 85-103.

[Meyer, 1992] B. Meyer, *Eiffel: The Language*: Prentice Hall, 1992.

[Mitchell and Wellings, 1996] S. E. Mitchell and A. J. Wellings, "Synchronisation, Concurrent Object-Oriented Programming and the Inheritance Anomaly," *Computer Languages*, (22)1, pp. 15-26, 1996.

[Mitchell, 1995] S. E. Mitchell, "TAO - A Model for the Integration of Concurrency and Synchronisation in Object-Oriented Programming", PhD Thesis, *Department of Computer Science, University of York, UK*, YCST-95-009, available through FTP from "ftp://ftp.cs.york.ac.uk/reports", 1995.