*Marcin Kuta**

TRENDS IN MODERN EXCEPTION HANDLING

Exception handling is nowadays a necessary component of error proof information systems. The paper presents overview of techniques and models of exception handling, problems connected with them and potential solutions. The aspects of implementation of propagation mechanisms and exception handling, their effect on semantics and general program efficiency are also taken into account. Presented mechanisms were adopted to modern programming languages. Considering design area, formal methods and formal verification of program properties we can notice exception handling mechanisms are weakly present what makes a field for future research.

Keywords: *exception handling, exception propagation, resumption model, termination model*

TRENDY WE WSPÓŁCZESNEJ OBSŁUDZE WYJĄTKÓW

Obsługa wyjątków jest współcześnie nieodzownym składnikiem systemów informatycznych odpornych na błędy. W artykule przedstawiono przegląd technik i modeli obsługi błędów, związane z nimi problemy oraz ich potencjalne rozwiązania. Uwzględniono również zagadnienia dotyczące implementacji mechanizmów propagacji i obsługi błędów, ich wpływ na semantykę oraz ogólną efektywność programów. Przedstawione mechanizmy znalazły zastosowanie we współczesnych językach programowania. Jeśli chodzi o dziedzinę projektowania, metody formalne oraz formalne dowodzenie własności, to mechanizmy obsługi wyjątków nie są w nich dostatecznie reprezentowane, co stanowi pole dla nowych badań.

Słowa kluczowe: *obsługa błędów, progagacja wyjątków, semantyka wznawiania, semantyka zakończenia*

1. Introduction

Analysing modern computer systems we notice increasing efforts to deal with problem of software reliability. This feature becomes crucial in real time systems, where least lack of reliability can make them completely useless, even dangerous and significantly decides about working costs. Beside regular services offered by software modules, which we can describe as expected and welcome, much attention should be paid to other types of services. This article deals with issue of exceptional services e.g. services we perceive as unwelcome but expected. The issue of hazards (events unwelcome and unexpected) is beyond a scope of the paper. It is also necessary to distinguish two kinds of exceptions: synchronous and asynchronous.

*Faculty of Electrical Engineering, Automatics, Computer Science and Electronics AGH University of Science and Technology, Cracow, Poland, mkuta@agh.edu.pl

Complex information systems devote more than half of code to errors handling. Exception handling mechanism solves the problem of control flow when unexpected event occurs. These mechanisms, although relatively general, are tightly related to error handling and contribute to program reliability. Due to formal requirements it is worth adding that exception mechanisms can be applied to other problems like unstructured control flow. This article gives review of ancient exception handling techniques, then presents the newest solutions in the area of exception handling, including description of the models, exception features and issues related to their implementation.

2. Traditional exception handling mechanisms

Historically, the first methods to deal with problem of exceptions were status return value and status flag techniques. Status flag technique, which is widely known for people, who have had even few experiences in C language, uses global variable to indicate exceptional state. The second technique – return value technique binds error with result of function invocation. The set of function return values is divided into two subsets: set corresponding to regular use and set corresponding to exceptional use. There is remaining problem how to project exception to exceptional return value. It is usually addressed by extra definitions by means of pre-processor or creation of special interfaces.

Main advantage of traditional techniques is that they introduce only small overhead to programs' size or speed. At the same time list of drawbacks prevails (see also [4]):

- Unnecessary increase of program complexity. Each error check introduces new nested block in worst case, ending up with many additional levels of nesting. The regular flow and exceptional flow aren't clearly separated, what makes programs difficult to read and maintain.
- Ease of omitting some errors cause of fine granularity of exception handling (potentially each called function should be checked for returned value).
- Difficulties in extending a program with new exceptions, because, as mentioned earlier, two program flows are interlaced within the same syntax constructs.
- Problematic in concurrent programming in case of usage a status flag technique. As global variable is used to indicate errors, it can indicate a state that isn't already up-to-date, since threads use this variable without synchronisation mechanisms. From the other hand introducing by users synchronisation mechanisms would make error handling excessively complex and would end up with mixing program implementation and language implementation.
- Imposes unintuitive changes of functions return values in order to contain special error value, for example C function *char getchar*() must be replaced by *int getchar*(). In the effect documentation of systems using such functions is less valuable.

- Sometimes it is not possible to return appropriate value that would indicate an error. Following example, although limited to one programming language (C++), shows potential problems:

```
Base& b;
...
Derived& d = dynamic_cast<Derived&>(b);
```

  The key issue is that reference variable must always refer to some variable, null references are not possible. Returning value 0 by operator *dynamic_cast* in case of unsuccessful type cast is not appropriate because in that case a temporary object is generated and its value is set to 0. A correct approach is to apply new handling mechanism, raising *bad_cast* exception by *dynamic_cast* when it is impossible to perform properly operation.
- Functions like object constructors may be declared to return no values at all (including void type), making return value technique impossible to apply.

3. Signals

Signal mechanism is appropriate for handling asynchronous events (e.g. arrival of alarm) and events related to internal errors caused by program's faulty execution (e.g. segment violation, bus error, floating point overflow). Signals are implemented by functions *longjmp*, *setjmp* and handled implicitly, without users' additional checks or function calls. The only effort is to install signal handling procedure. This is done by means of *signal* or *sigaction* functions. Signals can be ignored, or if no signal handling procedure is provided, default action is executed, according to signal meaning. Signals SIGKILL and SIGSTOP can't be ignored neither its default handling procedure changed. Signals can be explicitly sent from one process to another or to itself (using functions like *kill* or *sigsend*). For detailed discussion of signal mechanisms one can refer to [6, 13].

Among main difficulties related to signal handling we can encounter:

- Signal set varies with operating system type and release.
- Semantics of each signal is not preserved for all operating systems.
- There is no uniform signal API over operating systems, additionally it is susceptible to introducing errors like race conditions, lost signals and interrupted slow system functions.
- Information other than signal has arrived is not provided.
- Most of the signals are predefined, only few, like SIGUSR1, SIGUSR2, are available for programmers.

In the effect programs dealing with signals may not be portable even within different releases of the same operating system. Due to their nature, signals can't be chosen as a framework for synchronous exception handling.

Some implementations like Exceptional C adopt the same syntax to handle both synchronous exceptions and asynchronous exceptions (called signal exceptions).

4. Modern exception handling models

The main accent in modern exception handling is put to clearly distinguish two program flows: regular flow and exception flow. In this way advanced exception handling models addresses mentioned above problems with excessive growth of program complexity. Although it is possible to bind exception handlers with expressions or operators, modern exception handling techniques use notion of coarser guarded block, to which exception handler (special routine called when exception occurs within guarded block) is bound. Larger guarded block makes exception handling procedures occur relatively rarely, hence handling design is less burdensome. Another advantage is that exception can be handled at place, where we find it convenient, not only where it occurred, possibly handling a group of exception at one place. The result of exception propagation is that it is not necessary for each function to be error proof, since exceptions can be handled higher in call hierarchy. In this case exception flow is reverse to regular control flow. Usually dynamic propagation is used (i.e. invocation hierarchy is applied to find a handler). Another concept–static propagation (i.e. lexical hierarchy is applied to find a handler) proposed by Knudsen ([8, 9]) was important historically but hasn't turned out to be of huge practical significance.

Next feature is that exceptions can be parameterised and contain as many information as needed to describe exception event, without additional unnecessary efforts, overhead or affecting program's structure. There should be also possibility to raise or reraise exceptions explicitly (using keywords like *raise*, *throw*, *resume*). Reraising an exception allows functions to be partially error proof, doing some necessary cleanup, at the same time passing the main responsibility higher in program's hierarchy. Finally, modern models allow exceptions safe usage in multithreaded environment.

Following piece of code shows usage and essential difference between three main exceptions handling models.

```
void bar(void){
...
label2: raise e; // here exception is raised
...
}

void foo(void){

label1: try{
    ...
    bar();
    ...
}catch(Exception e) {
```

```
label4: ... // here exception is handled
}
label3: ...
}
```

Assuming the exception was thrown at label `label2`, then handled at label `label4`, next the control flows, depending on handling model, respectively to label `label1` (retrying model) or `label2` (resumption model) or `label3` (termination model). This distinction is at present perceived as the main classification of handling models and is widely reported in literature [1, 10, 12, 14].

4.1. Retry model

Retry model can be easily mimicked by termination model, as shown below.
Retry block (n is number of retries):

```
try{
   ...
}retry(n, Exception e) {
   // do necessary cleanup
}
```

can be replaced to use termination model by:

```
for(int i = 0; i < n; i++){
    try{
        ...
        break;
    }catch(Exception e) {
        // do necessary cleanup
    }
}
```

What is very important, such remodelling doesn't make function less structured, as concepts like *goto* are not needed.

Retry model isn't supported by programming languages as embedded mechanism because:

- Each user can simulate this model by termination model on its own.
- Retry model is slightly more error prone than termination model (retry clause at the end of the block versus try clause at the beginning of the block in case of termination model).

4.2. Comparison of termination and resumption semantics

It remains to consider two most important models: termination and resumption, both of them having significant advantages.

Termination model advantages:

- Simpler in implementation and usage than resumption model.
- Most of software was written by assuming termination model.
- Provides enough strong semantics for all applications.
- Contributes to creation of systems easier for maintenance.

Resumption model advantages:

- More general then termination model (if not implemented, must be mimicked by tricks).
- It gives simple solution of problem of resource exhaustion.
- It doesn't require significantly more expensive implementation than termination model.
- Important for complex systems like OS/2.

Resumption model seems more attractive since it offers the strongest, most general semantics in comparison to other models. Unfortunately, after analysis of millions lines of code, it has turned out not to be very useful in practice (see also [14]). With resumption model it is also possible to create infinite loops, highly undesirable and difficult to detect bug. The simplest example of such recursive resuming is shown below:

```
try{
    resume e;
}catch(Exception e){
    resume;
}
```

In the example it is easy to preview an incorrect loop, but in complex one such loop may appear due to dynamic handler selection, making debugging hard.

Resumption model is more expensive in implementation than termination model, as in the latter part of the context of program can be destroyed (stack unwinding) and simply forgotten. Resumption model requires context saving when exception occurs and in the effect applying more advanced data structures like cactus stack to implementation. Termination model gained more popularity over resumption model and is implemented in most of the languages.

4.3. Other concepts

We can also imagine coexisting of resumption and termination semantics within one language. In this approach we could distinguish three kinds of exceptions:

1) throw-only,
2) resume-only,
3) both thrown or resumed (depending on context).

With such generalised exceptions following problems arise:

- Resume exception can be overridden to be throw exception (but reverse is not true). Such construct introduces potential place for errors, as exceptions can be used without an awareness of different model to be tight to exception.
- Exception inheritance becomes questionable. Changing kind of derived exceptions makes code hard to read and analyse since control flow can be reckoned at run time and depends on exact type of exception.
- Matching exceptions to handlers, e.g. throw exception to resume handler or resume exception to throw handler, is not correct, being another place leading to potential errors.

Above problems and low usefulness of resumption model cause such construct doesn't seem justified as model is unclear, hardens implementation and usage, leads to unintended errors, finally creates more problems than solves.

Besides handling model, exceptions can be described by additional features, what is summarised in the Table 1.

Table 1

Exception features occurrence depending on language

	PL/1	CLU	Mesa	Ada83	Cui	C++	Java	C#	Eiffel	ML	Modula 3	Argus
termination		•	•	•	•	•	•	•	•	•	•	•
resumption	•		•		•				•			
multilevel			•	•	•	•	•	•	•	•	•	
scoped naming	•	•	•	•	•	•	•	•		•	•	•
inheritance						•	•	•				
parameters		•	•			•	•	•		•	•	•
bound					•							
asynchronous												

Scoped naming feature is related rather to language design than exception design and exists in almost all important languages. Flat namespace becomes polluted with program growth and then name collision is getting more probable. Exceptions should profit from scoped naming possibility.

5. Propagation depth

Three propagation models are considered:

1) single,
2) multilevel,
3) mixed.

- Single level model

  In single level model exception is propagated only through one active block. If no handler is provided there program is terminated or exception is converted to an error which usually leads to program termination. The only important language to implement this model is CLU. The idea behind this model is to establish mapping. Caller of function is relaxed from details of exceptions raised by functions used in the implementation of called function. Unfortunately, there are also important limitations for the model. First, it requires almost each function to handle with errors. The better idea is to deal with exception in a few, well defined interfaces. It may be also impossible for function to handle error due to language or library limitations and then its only strategy is to propagate it further. This concerns multilingual environments like CORBA, DCOM.

- Multilevel model

  According to multilevel model exception is propagated through active blocks until explicit or implicit exception handler catches it. Because exceptions can be propagated through many blocks each function has to decide whether to catch an exception (we say that callee function masks exception to caller, because caller acts as if no exception took place) or propagate it further.

  Let's suppose a function propagates an exception declared to catch (raised exception doesn't figure on function's list of raised exceptions) outside its body. Two approaches to problem are possible. In first solution such behaviour is treated like a run-time error and a special handler is called (which usually terminates the program). This solution is adopted by C++. Former behaviour can be considered as undesirable. Possibly it is better to reject such cases at compile time, reporting an error.

  Last solution, in shape adopted by Java, distinguishes two kinds of exceptions: checked exceptions and runtime exceptions ([3]). In case of checked exceptions each function must catch all such exception raised within its body or explicitly declare uncaught exceptions in its signature (control flow analysis and control flow graphs judge whether function definitions and calls conform to their specification). However for some exceptions (run-time exceptions) that can be thrown almost anywhere, necessity of providing handler would be too cumbersome, so above rules don't apply to them.

  It is also not obvious reading signature's declaration, what is function's default propagation behaviour. In Java signature *void foo(void)* means that function throws no but runtime exceptions (which are raised and handled implicitly). Contrarily to Java in C++ the above signature means that function can throw any exception. To assure function call will always end successfully, function with signature *void foo(void)throw()* should be provided.

- Intermediate model

  In this model exception can be propagated if it is specified in function's header. Otherwise it may be transformed to more general type using exception hierarchy.

Only the multilevel model turns out to be useful and flexible. With single level model we loose possibility of propagating an exception to handle it at convenient place, maybe with other exceptions, grouped within one handler. Choosing a multilevel model, we have to decide a problem of checked exceptions. A declaration of raised exception in function signature is in a way global, since affects all functions in chain of invocations. Therefore checked exceptions don't exist in C++, heterogeneous and compiled language. In this way necessity of change of huge parts of code and precompiled libraries is avoided. Pure language like Java can require checked exceptions without above limitations.

6. Exception list

In many languages functions can declare that they have possibility to generate or pass from invoked functions some exceptions. If there is more than one type of uncaught exceptions we get exception list which looks like this:
void foo(void) throw(XException, YException, ZException);

Sometimes it can be difficult to fix which exceptions are thrown by function as in example taken from [1]:

```
template<typename T> void sort(T items[]){
    // using bool operator< (const T &, const T &)
}
```

Authors of [1] state that it is impossible to know what types of exceptions may be propagated from *sort* since *operator<* function is overloaded from each instantiation of template. But the problem can be fixed:

```
template <typename T>
void sort(T items[]) throw (XException, YException, Zexception)
{
    ...
    try{
        // using bool operator< (const T&, const T&)
    }
    catch(Xexception xe){ throw; }
    catch(Yexception ye){ throw; }
    catch(Zexception ze){ throw; }
    catch(...){ // warn
    }
}
```

The same trick can be used to handle functions taking function pointer as argument or functions calling directly or indirectly virtual functions. Additionally in languages with strict type control we can make an exception list a part of function pointer's declaration, for example declaration:

*void (*pf)(void) throw (int, string);*

constrains *pf* to point only to functions which can raise *int* or *string* exceptions but of no other types. In this way we can control exceptions propagation.

```
// function bar can raise any exception
void bar( void(*pf)(void) ) {... }

// function foo can raise only its own exceptions
// and pf's int and string exceptions
void foo( void(*pf)(void) throw (int, string) ) { ... }

void f1(void) { ... }
void f2(void) throw (int) { ... }

foo(f1); // illegal
foo(f2); // O.K.
```

7. Exception inheritance

It seems natural that in object oriented languages exception handling mechanisms profit from ability to create new types. This means that exceptions are represented as objects and classes, which don't differ from general-purpose ones. Languages without OO support can benefit only from traditional handling techniques or extensions with special exception type must be provided. Introducing new language dialect is questionable and needs special caution. Although some authors (e.g. see [2]) argue that it is worth distinguishing them by special keyword (for example *exception*) for documentation purposes, it doesn't seem reasonable. Such solution leads to unnecessary doubling the same functionality e.g. inheritance, operator overloading, polymorphism. It would end up with problem known from C++, where difference between *enum*, *struct* and *class* types becomes blur.

In languages where inheritance is obligatory and inheritance relation is described by tree (e.g. Java, C#) all exceptions can be lead out from one system exception (a root of exceptions tree), exempting a programmer from dealing with forest of hierarchies.

In languages where multiple inheritance is possible it is usually up to programmer to handle exceptions in that way that exception having multiple base types (more specific exceptions) are handled before base exceptions. It is always possible because cycles are not allowed in inheritance hierarchy, so partial order can be set up in inheritance relation. Relaxing from such behaviour would lead to less structured code.

8. Exception handling and language design

Introducing exception handling in object oriented approach requires additional effort from compiler creators. Its direct consequence is necessity of embedding a sort of

RTTI mechanism as a compulsory language feature. The idea behind this is to enable recognition of more specific raised exception, when only basic exceptions are declared to handler.

Traditionally language reflection is implemented in the following way (more technicalities can be found in [10]): each polymorphous type is bound with an instance of a special metaclass (the instance is common for all objects of inspected type). This metaclass serves as a type descriptor. Type descriptor is implemented as object of a special *type_info* class.

Additionally object exception handling imposes two extensions to RTTI implementation:

1. *type_info* class becomes a root of hierarchy of metaclasses providing more information about specific types like pointers, functions etc. Classes like *pointer_type_info*, *function_type_info*, inheriting from *type_info* base class, provide information concerning special features of pointers, function, tables, classes, etc.
2. *type_info* classes are generated not only for polymorphous types but for nonpolymorphous and embedded types as well. These classes are useful because raised exceptions, which can be of basic types like strings, integer or float numbers, must be identified at run time and identification mechanism should be uniform with complex types objects.

9. Impact of exceptions on semantics of program

Advanced exception handling mechanisms requires paying more attention to problem of program correctness by compiler. Let's consider following function:

```
void foo(void)
{
    try{
    label1: bar();
    }catch(...){
       // do some cleanup
    }
    label2: bar();
    ...
    Point p;
    ...
    label3: bar();
}
```

The above example shows three semantically different effects of raising an exception in function *bar*, depending on place it was raised (termination model assumed):

- Exception raised at label `label1`: the installed default handler catches an exception and after some cleanup function *foo* continues to execute instructions directly after *catch* block.

- Exception raised at label `label2`: no handler is executed inside function *foo*, instead function is popped of the program stack (stack unwinding is performed).

- Exception raised at label `label3`: function *foo* is popped of the program stack but before that active local objects must be destroyed (destructors to be invoked).

The presence of semantically different regions imposes maintaining by compiler or a program a list of current local active objects. This allows to remember program state and properly destruct objects at labels `label2`, `label3`, when exception occurs. Destructors can't do that work in this case, as it is not possible to put them at compile time. Such lists however can be generated at compile time or at run time depending on what are our priorities: program size or program speed.

Apart from compiler, programmers also should take into account possibly new semantics of program after introduction exception mechanism. Correct piece of code:

```
void bar(void) throw();
void foo(void) {
    key = lock(resource);
    ...
    bar();
    ...
    unlock(key);
}
```

may become invalid if function *bar* is redefined not to mask some exception, with new signature *void bar(void) throw (Exception)*;. Before popping function's *foo* frame from stack locked resources must be freed. The same is true for reserved heap memory in purpose to avoid leaks. The remedy is to free previously locked resources or allocated memory within exception handler block, to restore program to correct state. But the special attention must be paid. If an exception would occur during resource allocation, then this unallocated resource would be freed in exception handler. To avoid such incorrectness, resource allocations should take place outside main guarded blocks as in following example:

```
void foo(void){
   key = lock(resource);
   try{
      ...
   }catch(...){
      unlock(key);
   }
   unlock(key);
}
```

10. Implementation issues

Very good implementation of termination model of exception handling comes from works on languages like Clu, Modula 2 and C++. To distinguish different areas of program from the point of view of exception handling (see labels `label1`, `label2`, `label3` from first example in section 9) some mechanisms should be provided. This task is a responsibility of compiler or linker, which equips program with structures like table of ranges of program counter (or its equivalent). The table describes program's state in relation to exception handling and contains, for each range, exception handling procedures and cleanup functions (e.g. destructors). When exception is raised current value of program counter is compared to appropriate table of ranges in order to distinguish whether exception is raised inside guarded block and find address of calling function if necessary.

Additional comparisons and data structures effects in program's size growth and speed plunge, use the program exceptions or not. This price is however justified in most of the cases by augmented error protection and robustness. From the other side exception handling implementation doesn't impose any changes to object model (memory arrangement, virtual functions) and its relations like inheritance or composition.

11. Conclusions and considerations

In article there were presented various exception handling models. The ancient ones like status flag or return value don't fulfil requirements imposed to contemporarily created systems, consisting even from millions lines of code. The necessity of better handling mechanisms became apparent. Comparing static vs. dynamic propagation only the last one turned out to be of practical significance and became synonym of propagation mechanism.

Two models: resumption model and termination model have an advantage of clear separation of regular and exceptional flow and can be a framework for concrete implementation. Among them termination model gained more popularity over resumption model due to easier usage and implementation and despite slightly weaker semantics. Exception handling became easier and less burdensome thanks to larger responsibility of compiler. Still programmers should pay attention to new problems connected with issue of program correctness in presence of exception raising and handling.

Historical concepts like static propagation are of big theoretical importance but didn't enter widely to programming languages. Probably the most precise and close to ideal exception handling mechanisms are implemented in Java. Less precise model is adopted by C++ but in terms of time and space efficiency its implementation is considered to be the best. All mechanisms described above are good enough for languages of implementation but in area of design we can observe still lack of advanced exception modelling mechanisms e.g. for process algebras and formal verification of program properties.

Formal verification of exception handling correctness remains an area for future work. The presented paper is a first step towards development of new methods of exception handling analysis and modelling.

References

[1] Buhr P., Mok R.: *Advanced Exception Handling Mechanisms.* IEEE Transactions on Software Engineering, vol. 26(9), September 2000

[2] Buhr P., MacDonald H.: *Synchronous and Asynchronous Handling of Abnormal Events in the iSystem.* Software-Practice and Experience, vol. 22(9), September 1992

[3] Eckel B.: *Thinking in Java.* Prentice Hall Inc. 1997

[4] Gehani N. H.: *Exceptional C or C with Exceptions.* Software-Practice and Experience, vol. 22(10), October 1992

[5] Goodenough J. B.: *Exception Handling: Issues and A proposed Notation.* Communications ACM, vol. 18(12), December 1975

[6] Goodheart B., Cox J.: *The Magic Garden Explained. The Internals of UNIX System V Release 4. An Open System Design.* Prentice Hall of Australia Pty. Ltd. 1994

[7] Górski J.: *Inżynieria oprogramowania w projekcie informatycznym.* Warszawa, Wydawnictwo Informatyki Mikom 2000

[8] Knudsen J.: *Exception Handling – A Static Approach.* Software-Practice and Experience, vol. 14(5), May 1984

[9] Knudsen J.: *Better exception handling in block structured systems.* IEEE Software, vol. 4(3), May 1987

[10] Lippman S.: *Inside the C++ Object Model.* Addison-Wesley Publishing Company 1996

[11] Lippman S., Lajoie J.: *C++ Primer.* Addison-Wesley Publishing Company 1998

[12] Mok R.: *Concurrent Abnormal Exception Handling Mechanisms.* Waterloo, Canada, Univ. of Waterloo, N2L3G1, September 1997, internet: ftp://plg.uwaterloo.ca/pub/uSystem/MokThesis.ps.gz (master thesis)

[13] Stevens R.: *Unix Network Programming.* Prentice Hall 1990

[14] Stroustrup B.: *The Design and Evolution of C++.* Addison-Wesley Publishing Company 1994

[15] Stroustrup B.: *The C++ Programming Language.* Addison-Wesley Publishing Company 2000