SOFTWARE FAULT TOLERANCE IN OBJECT-ORIENTED SYSTEMS: APPROACHES, IMPLEMENTATION AND EVALUATION

B. Randell, J. Xu and A. Zorzo

TECHNICAL REPORT SERIES

No. 597 June, 1997
Software Fault Tolerance in Object-Oriented Systems: Approaches, Implementation and Evaluation

B. Randell, J. Xu and A. Zorzo

Abstract

In this paper we introduce our experience with the use of C++ (an object-oriented language) and Open C++ (a reflective version of C++) to implement reusable, dependable control structures that can effectively support the provision of software fault tolerance in the application layer of an object-oriented system. Our implementation follows two different approaches - the object-library approach, using inheritance and delegation, and the reflective approach, based on metaobject protocols. The target environment used for our experiment is a distributed system consisting of multiple SPARC workstations connected through TCP/IP. Testing based on software fault injection is performed and performance-related analysis is provided. Both experimental and analytic results in this paper show that the object-oriented approach with the design of metaobject protocols is particularly promising for the development of computing systems with high dependability requirements.
Bibliographical details

RANDELL, Brian.

Software Fault Tolerance in Object-Oriented Systems: Approaches, Implementation and Evaluation
[By] B. Randell, J. Xu and A. Zorzo

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 597)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
Xu, Jie
ZORZO, Avelino Francisco

Abstract

In this paper we introduce our experience with the use of C++ (an object-oriented language) and Open C++ (a reflective version of C++) to implement reusable, dependable control structures that can effectively support the provision of software fault tolerance in the application layer of an object-oriented system. Our implementation follows two different approaches - the object-library approach, using inheritance and delegation, and the reflective approach, based on metaobject protocols. The target environment used for our experiment is a distributed system consisting of multiple SPARC workstations connected through TCP/IP. Testing based on software fault injection is performed and performance-related analysis is provided. Both experimental and analytic results in this paper show that the object-oriented approach with the design of metaobject protocols is particularly promising for the development of computing systems with high dependability requirements.

About the author

Brian Randell has been a Professor of Computing Science in the Department of Computing Science at the University of Newcastle upon Tyne since 1969.

Jie Xu joined the Department of Computing Science at the University of Newcastle upon Tyne in April 1990 as a Research Associate and has been a Senior Researcher since 1995.

Avelino F. Zorzo joined the Department of Computing Science at the University of Newcastle upon Tyne in 1995, where he is a PhD student.

Suggested keywords

DELEGATION
METAOBJECT PROTOCOLS
SOFTWARE FAULT TOLERANCE

INHERITANCE
OBJECT-ORIENTED PROGRAMMING

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425 001.64404 U.D.C. 681.322.06 519.687
Software Fault Tolerance in Object-Oriented Systems: Approaches, Implementation and Evaluation

Brian Randell, Jie Xu and Avelino Zorzo
Department of Computing Science
University of Newcastle upon Tyne

Abstract: In this paper we introduce our experience with the use of C++ (an object-oriented language) and Open C++ (a reflective version of C++) to implement reusable, dependable control structures that can effectively support the provision of software fault tolerance in the application layer of an object-oriented system. Our implementation follows two different approaches — the object-library approach, using inheritance and delegation, and the reflective approach, based on metaobject protocols. The target environment used for our experiment is a distributed system consisting of multiple SPARC workstations connected through TCP/IP. Testing based on software fault injection is performed and performance-related analysis is provided. Both experimental and analytic results in this paper show that the object-oriented approach with the design of metaobject protocols is particularly promising for the development of computing systems with high dependability requirements.

1 Introduction

A promising approach to the improvement of software dependability is the application of some new ideas in program structuring, and particularly object-oriented programming, to the control of system complexity. However, despite of various new techniques, a realistic software system will still contain residual software design faults given the complexity of today's computing systems [Lee & Anderson 1990]. Approaches and mechanisms are therefore required to enable a system to tolerate software faults remaining in the system after its development. Practical techniques do exist and have been proved successful. (The Wiley book [Lyu 1995] presented a comprehensive and detailed survey of software-fault tolerance issues, where further references can be found.)

In this paper we investigate the problem of how to implement reusable fault-tolerant mechanisms in object-oriented software and perform an experimental evaluation to examine the practical effectiveness of the object-oriented approaches to software fault tolerance. We emphasize the reusability of the implemented mechanisms through two different implementation

---

1 This paper was mostly based on [Zorzo et al. 1996] and [Xu et al. 1996]
approaches: 1) a pre-defined C++ library, and 2) metaobject protocols with better user-
interfaces (using the Open C++ language).

Our study and experiment have the following novel characteristics:

1) high-level support for tolerating software faults in the object-oriented application layer;

2) pre-defined C++ library that supports the development of fault-tolerant programs by
means of existing software fault tolerance schemes such as recovery blocks [Randell
1975] and N-version programming [Avizienis 1985];

3) clear separation of concerns that reduces the burden of a specific programmer (e.g. user
of a class, designer of a fault-tolerant program, and system programmer) and thus
decreases the probability of producing software faults;

4) reflective meta-level interface [Kiczales et al. 1991] that hides most implementation
details of a fault-tolerant mechanism from the application-level, and enable the
programmer to flexibly change the use of different fault tolerance schemes without
changing the code at the application level; and

5) experiment-based analysis that verifies the effectiveness of our implementation and
shows acceptable runtime overheads.

The remainder of this paper is organized as follows. In section 2 we describe an
implementation framework for software fault tolerance and give a brief introduction of the
existing schemes. We introduce our experimental setup in the third section and address the
implementation of fault-tolerant mechanisms in the fourth section. Section 5 provides testing
results and the performance-related analysis. Section 6 deals with the reflective implementation.

2 Software Fault Tolerance: General Framework and Schemes

Software fault tolerance, in the context of this paper, is concerned with all the techniques
necessary to enable a system to tolerate software design faults. In order to discuss software
fault tolerance, we must first establish or obtain an abstract model of describing software
systems. A system is defined to consist of a set of components which interact under the control
of a design [Lee & Anderson 1990]. The components themselves may be viewed as systems in
their own right. In particular, the design of a system is also a component, but has special
characteristics such as the responsibilities for controlling the interactions between components
and determining connections between the system and its environment.

Because software faults are permanent in nature, techniques for software fault tolerance in
principle require redundancy of design and/or of data and/or of environment (e.g. different
processors). Design redundancy (or design diversity) is the approach in which the production
of two or more components is aimed at delivering the same service through independent
designs and realizations [Avizienis 1985][Laprie et al. 1987]. The components, produced
through the design diversity approach, or more generally involving data diversity and
environment diversity, from a common service specification, are called variants. By
incorporating at least two variants of a system, tolerance to design faults necessitates an
adjudicator (or a decision algorithm) that provides an (assumed to be) error-free result from the
execution of variants. To coordinate the execution of variants and the final adjudication, a control mechanism or controller is required. (For more details of a general framework for software redundancy, see Figure 1 and [Xu et al. 1995a].) Classical techniques for tolerating software design faults are mostly based on some form of design diversity, including recovery blocks (RB) [Randell 1975] and N-version programming (NVP) [Avizienis 1985].

![Diagram](image)

**Fig. 1: A Fault-Tolerant Component with Diverse Designs**

(reproduced from [Xu et al. 1995a])

The first scheme designed to provide software fault tolerance was the recovery block scheme (RB). In this approach, variants are named alternates and the main part of the adjudicator is an acceptance test that is applied sequentially to the results produced by variants: if the first variant fails to pass the test, the state of the system is restored and the second variant invoked on the same input data and so on, sequentially, until either some result passes the acceptance test or all the variants are exhausted. Software variants are organized in RB in a manner similar to the standby sparing techniques (dynamic redundancy) used in hardware and may be executed serially on a single processor. N-version programming (NVP) is a direct application of the hardware N-modular redundancy approach (NMR) to software. A voting mechanism determines a single adjudication result from a set or a subset of all the results of variants which are usually executed in parallel. Apart from the two well-known approaches, a number of other schemes for software fault tolerance have been developed such as N Self-Checking Programming [Laprie et al. 1987], k(n-1)-Variant Programming [Xu & Randell 1997], and Self-Configuring Optimal Programming [Xu et al. 1995c] for sequential programs, and Conversations [Randell 1975] and Coordinated Atomic Actions [Xu et al. 1995b] for concurrent systems. Due to the limitation of space we will not deal with them further.

Complexity control is particularly crucial in designing and implementing a fault-tolerant software system. Conceptually we classify programmers into three classes with respect to a fault-tolerant object-oriented program: the users of fault-tolerant objects, the ft-designers (i.e. designers of fault-tolerant objects) and the system programmers (e.g. meta-level programmers). If just using a fault-tolerant object or an improved version of a original object, a user needs to know a little information about whether the interested object is fault-tolerant or
not. However in order to produce a fault-tolerant program, an ft-designer needs to know more about various software fault tolerance schemes and is responsible for developing variants and the related adjudicator. The ft-designer has to choose a special scheme, such as RB or NVP, for the development of the fault-tolerant program. Note that the control mechanism for a fault tolerance scheme should be application-independent and so can be made reusable. It is the system programmer who is responsible for providing the ft-designer with the high-level programming interface for various software fault tolerance schemes, which hides the implementation details of the corresponding control mechanisms. Of course, what we emphasize here is the separation of different concerns; in practice a person may well play several different roles.

There are several possible solutions to the provision of supports for software fault tolerance in the application layer [Xu et al. 1995a], but most of previous proposals and implementations were based on pre-defined classes and run-time libraries (i.e. the object-library approach). Huang and Kintala of AT&T Bell Labs. [Huang & Kintala 1993] developed three software reusable components in C that provide software fault tolerance in the application layer, supporting fault-tolerant structures like checkpointing and recovery, replication, recovery blocks, N-version programming, exception handling, re-try blocks etc. Their modules have been ported to a number of UNIX platforms, already applied to some new telecommunications products in AT&T and the performance overhead due to these components has been shown to be acceptable. However, their implementation is not object-oriented.

Another implementation approach is based on reflection and metaobject protocols [Kiczales et al. 1991]. [Xu et al. 1995a] introduced a reflective system architecture for implementing fault-tolerant programs. The reflective meta-level interface used in this architecture can hide most implementation details of a fault-tolerant mechanism (e.g. recovery blocks and NVP) from the application-level and enables the programmer to flexibly change the use of different fault tolerance schemes without having to change the code of application programs. Both implementation approaches discussed here are examined carefully in our experiment.

3 Description of the Experiment

In our experiment GNU C++ version 2.6.3 [Stroustrup 1991] and Open C++ version 1.2 [Chiba 1993] were used as basic programming languages. Open C++ is a C++ pre-processor that provides the programmer with two levels of abstraction: the base level, dedicated to traditional C++ object-oriented programming, and the meta level which allows certain aspects of the C++ programming model to be re-defined [Maes 1987]. The target environment for our experiment consists of a set of workstations running UNIX, connected through TCP/IP (see Figure 2).

Our prototype implementation was organized in a client-server manner. Clients and servers communicated with each other by means of BSD sockets. We believe that this communication facility is effective enough to support our experiment on fault-tolerant computing and object-oriented development (since communication is not our major concern here, it is assumed that the communication mechanisms used are highly dependable). Other communication mechanisms may be considered in our further research, such as remote procedure calls (RPC) [Sun 1990]. We defined a special object that provides the functionality of communication
through sockets, named *Socket*. Two operations were attached to *Socket: ConnectWrite()* that is responsible for establishing the connection between the client and the server and for sending an object to the server; and *ReadClose()* responsible for receiving the result (or object) from the server and for closing the connection.

**Fig. 2: Target Environment (Hardware and Software)**

## 4 Implementation

In order to explain the implementation of our experiment system clearly, let us first consider a simple example of a sorting application. This example is made up of three sorting servers and an application program (as the client). Each sorting server receives an object which contains a list of integer numbers, sorts the list, and then sends the result back to the application. Both the client and the servers were implemented using C++ [Stroustrup 1991].

Given a base class called *ArrayList*, it is associated with a set of operations which can be re-defined and implemented in an alternative manner or a more dependable way, by means of the inheritance mechanism. Take the operation *Sort* as an example. When creating an array object, the user of the class (or object) must specify which kind of *ArrayList* (e.g. *NORMAL*, *QUICK*, *HEAP*, *SHELL*, *NVP*, *RB*, etc.) is wanted. The constructor of class *ArrayList* will create a corresponding object according to the given *kind* parameter. In C++ the *Sort* operation may be implemented as virtual in the base class. It can thus be re-defined in the sub-classes of *ArrayList*. However, if the sub-classes are not used, then this virtual *Sort* operation provides basic sorting based on a simple “bubble sort” algorithm.

```c++
class ArrayList {
    public:
        ArrayList(int kind);
        virtual int Sort();
        // other operations ...
    protected:
        Vector list(MAXelem);
        int size;
};
```
By inheritance and re-definition of the Sort operation, sub-classes of the ArrayList class can be used to implement faster versions of sorting (e.g. QuickArrayList, HeapArrayList, ShellArrayList) or dependable ones (e.g. RBArrayList, NVPArrayList based on the recovery blocks and the N-version programming schemes). Of course, other operations (e.g. Search()) can also be re-implemented in these sub-classes (see Figure 3). Let \( i = \{ \text{RB, NVP, Quick, Heap, Shell} \} \). We can define the following class.

```c++
class iArrayList : public ArrayList {
    public:
        virtual int Sort();
};
```

![Diagram](image)

**Fig. 3:** Inheritance and the Class Hierarchy

Further let \( j = \{ 1, 2, 3 \} \) and `machine` = \{ glororan, bowes, yardhope \}. We can define faster, remote sorting operations as follows.

```c++
int iArrayList::Sort()
{
    Socket<Vector> s;
    s.ConnectWrite(pj, machine, list);
    s.ReadClose(list);
    return NULL;
}
```

The system programmer implements control mechanisms for various fault tolerance schemes as C++ library programs and makes them reusable for different ft-designers. For example, an FTClass provides several operations that implement recovery blocks and N-version programming control structures. This `FTClass` class can be reused easily by ft-designers for various fault-tolerant applications such as searching and real-time control.

```c++
template <class TF> class FTClass {
    public:
        FTClass();
        void Variant(int port, char *machine);
        bool RecoveryBlock(TF& obj);
        bool NVersionProgramming(TF& obj);
        // other control structures
    private:
        // some internal states (e.g. port/machine-variants)
};
```

In the class definition above, TF is the class/type of the object that will be manipulated by the variants in the remote servers. The `port/machine` parameters of the `Variant()` operation are the address of the computer that contains the server for one of the variants used by
RecoveryBlock or NVersionProgramming. Both control structures return a Boolean result indicating if the variants produce an acceptable result (TRUE) or an incorrect result (FALSE). Since an adjudicator is very often application-specific, the object passed to the fault-tolerant control structures must provide a pre-defined Adjudicator operation.

More precisely, the RecoveryBlock operation receives an object and passes this object onto the first variant. When the result is returned, the adjudicator operation is executed. If the result is acceptable then TRUE is returned, or otherwise the object (state) is recovered and the next alternate is executed. If all alternates fail then the RecoveryBlock operation returns an error signal (FALSE) indicating that an acceptable result cannot be produced. Again, sockets have been used in this particular experiment. The communication among servers and application can easily be changed by replacing the code where the alternate is activated, or implementing different fault-tolerant classes, each one activating the variants using a different communication protocol, depending upon the servers. (For example, mechanisms used in the Arjuna system [Shrivastava et al. 1991] could be employed).

```cpp
template <class TF>
bool FTClass<TF>::RecoveryBlock(TF& obj)
{
    Socket<TF> s;
    TF bk = obj;
    for (int i=0; i<nvariants; i++) {
        // Try alternate/variant
        s.ConnectWrite(port[i],machine[i],obj);
        s.ReadClose(obj);
        if (!obj.Adjudicator()) {
            // Ensure test was not successful
            // recover the object and try
            // another alternate/variant
            obj = bk;
        } else
            return(TRUE);
    }
    // All alternates failed
    return (FALSE);
}
```

Similarly, we have the following algorithm for NVP:

```cpp
template <class TF>
bool FTClass<TF>::NVersionProgramming(TF& obj)
{
    Socket<TF> s[MAXVARIANT];
    TF auxL[MAXVARIANT], bk = obj;
    // activate all versions/variants
    for (int i=0; i<nvar; i++)
        s[i].ConnectWrite(port[i],address[i], obj);
    for (int i=0; i<nvar; i++)
        s[i].ReadClose(auxL[i]);
    if (!obj.Adjudicator(auxL)) {
        // the voter/adjudicator didn't find
```
// a good result, recover and return!
obj = bk;
return(FALSE);
}

// voter/adjudicator found a good result!
return(TRUE);

By using the reusable control structures provided above an ft-designer can now easily construct dependable versions or sub-classes of the ArrayList class. The ft-designer just needs to develop the variants and adjudicator, and to indicate which kind of object is handled by the FTClass.

```c
int RBAArrayList::Sort()
{
    FTClass<vector> ftObj;
    bool error;
    ftObj.Variant(p1,"glororan");
    ftObj.Variant(p2,"bowes");
    ftObj.Variant(p3,"yardhope");
    error=ftObj.RecoveryBlock(list);
    return error;
}

int NVPArrayList::Sort()
{
    FTClass<vector> ftObj;
    bool error;
    ftObj.Variant(p1,"glororan");
    ftObj.Variant(p2,"bowes");
    ftObj.Variant(p3,"yardhope");
    error=ftObj.NVersionProgramming(list);
    return error;
}
```

More simply, a user of the fault-tolerant sub-classes NVPArrayList or RBAArrayList, only needs to specify which sub-class will be used.

```c
main()
{
    ArrayList obj1(RB), obj2(NVP), ...
    // the rest of the user program
}
```

To summarize, table 1 shows the separation of different concerns according to our classification in section 2.

<table>
<thead>
<tr>
<th>User of the Class</th>
<th>ft-Designer</th>
<th>System Programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList class</td>
<td>RBAArrayList sub-class</td>
<td>FTClass class</td>
</tr>
<tr>
<td>Variable definition</td>
<td>NVPArrayList sub-class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variants and Adjudicator</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Separation of Concerns

5 Tests and Analysis Based on Software Fault Injection

In order to examine the behaviour and effectiveness of each implementation a set of tests was performed. The execution time was measured in microseconds (µseconds); the time of network communication is not involved and is assumed as constant. A data set of 200 elements was used to measure the times.
In Test One every Sort operation was executed without fault injection, and all lists received were correctly sorted. It is interesting to note that the dependable operations based on RB and NVP have longer execution time than the faster versions of the normal sorting operation. That overhead is mainly due to the coordination of variants and the execution of the adjudicator. In addition, since NVP needs to wait for the completion of the slowest variant (e.g. HeapSort in the example), it has longer execution time than RB which uses QuickSort as its first alternate. However, due to the use of an acceptance test, the overhead of RB is slightly higher than that of the QuickSort operation itself.

In Test Two, one software design fault was injected into the QuickSort server, by randomly commenting-out a line of the algorithm. If and when the sorting operation fails, the RBArrayList::Sort() will call its second alternate. Note that, compared with the results in Test One, the overhead of NVP is still the same, but the time of RB has increased because of the execution of the second variant.

In Test Three, a further design fault was inserted into the HeapSort server; this causes the failure of three algorithms (i.e. Quick, Heap and NVP). The other two (Shell and RB) can still deliver correct computation. NVP fails because the voter failed to find the majority of three different results. This time, we can see that the time of NVP is shorter than that of RB since RB has to activate all its alternates to reach a correct result. The total overhead of RB is the execution time of all the variants plus the cost of state restoration and adjudication.

In Test Four, a software fault which cannot be detected by the acceptance test was inserted into the QuickSort server, and both Quick and RB produce erroneous lists without signalling any error. The general overhead is similar to those in the first test. Note that NVP can tolerate such a fault very effectively.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Test</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick</td>
<td>Result</td>
<td>OK</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>Time</td>
<td>3198</td>
<td>397</td>
<td>397</td>
<td>3319</td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td>Result</td>
<td>OK</td>
<td>OK</td>
<td>fail</td>
<td>OK</td>
</tr>
<tr>
<td>Time</td>
<td>12037</td>
<td>12037</td>
<td>9688</td>
<td>12037</td>
<td></td>
</tr>
<tr>
<td>Shell</td>
<td>Result</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Time</td>
<td>2788</td>
<td>2788</td>
<td>2788</td>
<td>2788</td>
<td></td>
</tr>
<tr>
<td>RB</td>
<td>Result</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>fail</td>
</tr>
<tr>
<td>Time</td>
<td>5403</td>
<td>15464</td>
<td>19574</td>
<td>5669</td>
<td></td>
</tr>
<tr>
<td>NVP</td>
<td>Result</td>
<td>OK</td>
<td>OK</td>
<td>fail</td>
<td>OK</td>
</tr>
<tr>
<td>Time</td>
<td>17163</td>
<td>17163</td>
<td>14379</td>
<td>17163</td>
<td></td>
</tr>
</tbody>
</table>

| Table 2: Performance-Related Testing Results |

Finally, it is important to point out that our fault injection-based experiment is not to verify the effectiveness of software fault tolerance schemes such as RB and NVP (many experimental evaluations have been done by other researchers), but to verify the usefulness of our object-
oriented implementation approaches. The experiment data obtained from our investigation indicate that for the chosen application our object-library implementation approach can achieve actual fault tolerance in programs with an acceptable run-time overhead.

6 Metaobject Protocol

Metaobject protocols [Kiczales et al. 1991] are interfaces to a system that give users the ability to modify the system’s behaviour and implementation incrementally [Stroud & Wu 1995]. Using this metaobject approach, we can implement control mechanisms for various fault tolerance schemes at the meta-level instead of developing the class library at the base (application) level. The language used for metaobject protocols in our experiment is Open C++.

Although Open C++ supports meta-level programming, it only provides a limited model of reflection [Fabre et al. 1995]. One of limitations leads to the difficulty in changing, by a metaobject, some types of internal variables of a baseobject. We made some modifications to Open C++ such that a baseobject can be passed to a metaobject which will then send the baseobject to servers. In order to provide a general and reusable metaclass, whenever the Meta_MethodCall operation is activated, it will receive a baseobject passed as a special parameter. The type of the passed baseobject will be defined when the baseobject is created. Thus the following metaclass can manipulate multiple different classes (note: TM is the type of the baseobject).

```cpp
template <class TM> class FTMetaClass : public MetaObj {
  public :
    FTMetaClass(VariantList& list);
    void Meta_MethodCall(Id mid, Id cat, ArgPack& args,
      ArgPack& rep, TM& obj);
  
  private:
    VariantList vList;
};
```

Based on metaobject protocols, the experimental system is re-implemented without using the sub-classes described in the previous sections. Instead, a reflective class of ArrayList is implemented as FTMetaClass. When the Sort() operation is called at the base level, the metaobject intercepts the call, handles the operation, and sends the list to the server(s). If the result is sorted, then the Sort() operation will not be actually called and the application will continue.

```cpp
template <class TM> void FTMetaClass<TM>::Meta_MethodCall
  (Id mid, Id cat, ArgPack& args, ArgPack& rep, TM& obj) {
  // Similar code used to RBArrayList::Sort(), or
  // NVPArrayList::Sort()
  if (!obj.Adjudicator)
    Meta_HandleMethodCall(mid, args, rep);
}
```
Fig. 4: Fault-Tolerant Mechanism Implemented at the Meta-Level

class ArrayList {
    public:
        ArrayList();
        // other methods ...
    //MOP reflect:
        virtual void Sort();
    protected:
        Vector list(MAXLEN);
        int size;
    }
    //MOP reflect class ArrayList : FTMetaClass;
    // rest of the programme ...
    void main() {
        refl_ArrayList data(listVarArray);
        // ...
        data.Sort();
        // ...
    }

At the meta-level FTMetaClass is responsible for implementing the control mechanisms for software fault tolerance. Suppose that we need to make the Sort() operation fault-tolerant. We are able to first make the Sort() operation reflective and extend its semantics by means of the directive //MOP reflect: of open C++. Note that in the example above the base-level class ArrayList is associated with the meta-level class FTMetaClass. So any instance of the class refl_ArrayList such as the object data will have a corresponding, reflected replica (or metaobject) which is managed by FTMetaClass at the meta-level. The code of the control mechanisms implemented in FTMetaClass will not appear in the source code of ArrayList. The listVarArray provides the data object with a list of software variants. Once the reflected object is created, the addresses of servers are stored by the metaobject.

Fault injection-based testing was also conducted on the reflective system, similar to that performed previously in Section 5. Table 3 shows the different run-time overheads between calling a C++ operation and calling a Open C++ (reflective) operation. The ratio indicates the extra cost caused by a reflective operation call. However, combining the results of Table 3 with
Table 2 of Section 5, we find this cost is only a very small part of the whole overhead imposed by fault tolerance mechanisms. (It contributes a smaller part if the cost of communication and coordination are taken into account.) By the use of metaobject protocols, we can now flexibly change the way of achieving fault tolerance (e.g. the use of different fault tolerance schemes) without having to change the application programs at the base level. (More information about the overhead of reflective operation calls can be found in [Chiba & Masuda 1993] and [Fabre & Perennou 1996].)

<table>
<thead>
<tr>
<th></th>
<th>Stobhill</th>
<th>Yardhope</th>
<th>Bowes</th>
<th>Glororan</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ method</td>
<td>56</td>
<td>3</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>Reflected method</td>
<td>100</td>
<td>6</td>
<td>100</td>
<td>16</td>
</tr>
<tr>
<td>Ratio reflected/C++</td>
<td>1.78</td>
<td>2</td>
<td>1.56</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Time Consumed by Reflected Methods and C++ Methods (μsec.)

Finally, Table 4 shows the separation of different programming concerns with respect to the design of metaobject protocols.

Table 4: Separation of Concerns Using Metaobject Protocols

7 Conclusion

There has been little work on experimental evaluation of software fault tolerance mechanisms in the context of object-oriented programming. We have conducted an initial experiment using two different implementation approaches: the C++ Library and the metaobject protocol in Open C++. Both implementation approaches impose acceptable run-time overheads and facilitate a clear separation of concerns in design and implementation stages. In particular, the metaobject approach provides ft-programmers with a clearer and simpler interface but with a slightly higher run-time overhead. However, when the cost of distributed communication (usually in the order of milliseconds) is taken into account, the overhead introduced by the reflective operation calls will not be of major concern.

Acknowledgement

This work was supported by the CEC-sponsored ESPRIT Basic Research Actions 3092 and 6352 on Predictably Dependable Computing Systems (PDCS and PDCS2), and by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa). Avelino Zorzo is also supported by CNPq (Brazil), grant n. 200531/95.6. Some ideas about reflection were discussed with our colleague Dr. Robert Stroud. We also thank Jean-Charles Fabre and Thanguy Perennou from LAAS for their constructive comments.
References


