



# *Aspect oriented software implemented node level fault tolerance*

Ruben Alexandersson, Peter Öhman, Martin Ivarsson

Dept. of Computer Engineering  
Chalmers University of Technology, SE-41296, Gothenburg, Sweden  
ruben@ce.chalmers.se  
Telephone: +46(0)31-7721685  
Fax: +46(0)31-7723663

## **REGULAR PAPER**

### **Abstract**

In this feasibility study it is shown by a number of reference implementations that the aspect oriented programming (AOP) paradigm is well suited for implementing node level fault tolerance.

By this, both systematic and application specific mechanisms can be implemented in a uniform way, and the fault tolerance code becomes completely separated from the primary function code. Furthermore, it is shown that any AOP language used for implementing node level fault tolerance must support monitoring of object data members which is presently not the case with AOP extensions to C++.

## **1 Introduction**

Software implemented fault tolerance is a well known technique for dealing with failures caused by both hardware and software faults. Compared to hardware implemented fault tolerance it has the advantage of being more flexible and cost efficient. Already today it is a widely used technique, and emerging application areas for cost efficient dependable systems such as automotive active safety systems will further increase the importance.

During the last decade reflection and metaprogramming techniques for building fault tolerant software have evolved [e.g. 1,2]. From a software engineering point of view the introduction of these techniques has lead to a number of major improvements. It supports efficient reuse of fault tolerance mechanisms (FTMs) and has, because of the separation of concerns in application source code, given less error prone code and simplified the task of program verification.

A fairly new approach to software implementation that has some of its roots in metaprogramming is aspect oriented programming (AOP) [3]. It has seen a rapid development over the last few years and is presently reaching the state where it can be considered a useful and working technique. In this paper it is shown that AOP provide an ideal base for implementing a wide range of FTMs spanning from application specific to systematic. The paper constitutes a simple aspect framework that includes the incremental recovery cache, time redundant execution, executable assertions, recovery blocks and control flow checking FTMs.

AOP has been discussed for some time in the domain of distributed fault tolerance [4,5,6,7] and has shown to be useful and also give performance benefits [8]. Prior to our work, no research has been addressing the question whether AOP can provide a base for implementing the lower level FTMs used within a node or in a non-distributed environment. In a somewhat related study AOP has been evaluated for the area of distributed real-time dependable systems [9].

## 2 AOP versus metaprogramming

The main difference between AOP and metaprogramming is in the underlying motivation. Metaprogramming and reflection is built on the observation that a program can in it self be represented as a data structure and made accessible to either itself or to another meta level program. Because of this both the program behavior and properties can be monitored and changed. Hence the focus is on supporting this general property of programs which in turn gives a technique that is both elegant and powerful but difficult to master [10].

AOP on the other hand is built on the observation that program system design and modularization is mainly conducted to reflect the primary function of the application. The effect is that all other functional and non-functional properties, called crosscutting concerns, can not be effectively modularized and the implementation of these becomes scattered throughout the program modules. AOP thus focuses on building a technique for modularizing crosscutting concerns and for separating concern specific code from the rest of the implementation [3].

Even though reflection and metaprogramming has been successfully used in the area of fault tolerance for achieving separation of concerns, it is not optimized for this specific purpose. AOP that focus only on crosscutting concerns gives more easily understandable language extensions since the intuitive model behind is easier to grasp. It has been argued that metaprogramming is easy to understand since it utilized the same language syntax as the base language. However, the argument fails to address the core problem of

understanding and mastering the semantic properties of the language and the underlying model. It is also argued by both [3] and [10] that AOP solutions generally give better performance since no runtime meta-protocol is used.

There is however also a strong relationship between AOP and metaprogramming. Most AOP implementations use some form of metaprogramming as its underlying technique. AOP can therefore be viewed as a better packaging of metaprogramming techniques for a specific purpose, namely that of modularizing crosscutting concerns. This leads to the question whether node level fault tolerance is a cross cutting concern in the AOP sense and if this packaging is suitable for fault tolerance needs. A later question is whether this difference in motivation actually leads to a better and or easier to use technology. We believe so but it is beyond the scope of this paper to conduct a rigorous comparative study.

### **3 AOP languages**

An aspect oriented implementation of a crosscutting concern consists of two parts, the actual implementation of the functionality associated with the concern, and the information on how that code should be integrated in the rest of the program. For the first part any traditional language like C, C++, or Java is well suited. But the traditional languages lack primitives for specifying how the concern specific code should be composed, or weaved, together to form the final system. An AOP language therefore defines a language for specifying rules for composing different implementation pieces together. The AOP language is then built as an extension to a traditional language to give

that language aspect oriented capabilities in the same way as C++ was build as an extension to C to provide object oriented capabilities to the C language.

An AOP language compiler is normally implemented as a source-to-source compiler that weaves the different concern specific sources together and then uses the base language compiler to compile the result into executable code. An alternative approach is taken by the AspectJ [11] language for Java which conducts the weaving on the compiled byte code. One view of an AOP compiler would be to say that it is simply a highly programmable compiler-compiler and this is why it is so well suited for implementing systematic FTMs (which are best suited for a compiler-compiler approach). The other view of an AOP language is that it is a programming language, or a well integrated extension to a programming language, that lets the programmer produce more or less application specific code. This is the reason why AOP is such a good candidate for a uniform way of building both systematic and application specific fault tolerance.

The main characteristics of an AOP language are what base language it extends and what joinpoints it supports. A joinpoint is an accessible point in the application source code where concern specific code can be inserted when conducting the system weaving. An AOP language that supports a large set of joinpoints is therefore the most powerful or flexible. Since AOP is mainly an effort for bringing a higher degree of modularization to OOP languages the efforts so far has been to construct languages that support joinpoints that give access to object members. That is methods, and variables that are part of the object state. An AOP language lets the programmer declare a pointcut that accesses a joinpoint. A pointcut can be linked to concern specific code segments called advices.

Currently there is a lot of effort put into the development of aspect oriented languages. Many projects [12] are still on the level of small research implementations but a few are reaching a mature state with stable languages and widely available tool support. The language that has reached the furthest in this regard is AspectJ [11] that is an extension to the Java language and it is currently in version 1.2. AspectJ is stable enough to be used in real life projects, and has become the reference language that other efforts are compared to. Another project that is well under way is AspectC++ [13] that will be available in version 1.0 later this year. These two languages are presently the most promising candidates for an efficient AOP platform for building software implemented fault tolerance and hence the focus of this study. There are also efforts being made to build AOP extensions to number of other programming languages including C [14,15]. However, these C projects are still on the level of small research projects.

The code examples of this paper is given in Java and AspectJ since it is more mature than AspectC++ that still lacks some of the AspectJ joinpoints. Because of this all the FTMs presented in this paper can not yet be implemented with AspectC++. The results presented should not however be considered as Java specific. The purpose of the code is to show how the FTMs can be implemented in a mature AOP language. Therefore any implementation details specific to Java or AspectJ will be pointed out and comparisons to the AspectC++ language will be made in the cases where AspectJ and AspectC++ differs.

## **4 Fault tolerance mechanisms**

To evaluate the feasibility of AOP languages a large number of the different FTMs were implemented. A subset of the implementations that demonstrates both the different benefits and limitations of present day AOP languages was selected for inclusion in this paper. The FTMs included are both fault detection and fault handling mechanisms. They range from systematic to application specific, and include mechanisms designed for both hardware and software fault tolerance.

### ***4.1 Incremental recovery cache***

A large number of FTMs relies on the concept of backward error recovery to return to a previously saved state. Normally it is part of the procedure to recover from a detected fault (as with recovery blocks [21,22]) but it can also be part of the fault detection mechanism (as with time redundant execution [19]). An efficient way of providing software implemented systematic support for backward error recovery is that of using an incremental recovery cache [16]. The idea behind the recovery cache is to not make a total snapshot of the system state when setting up a checkpoint but to start monitoring changes to system state variables. When a variable is changed for the first time after the checkpoint a copy of the old value is stored in the cache. Hence no memory or performance is lost to storing variables that are not changed. Previous implementations [16,17] rely on the concept of overloading the assignment operator to monitor changes to the variables that are part of the program state. They also require that the programmer explicitly declare which classes and variables should be monitored and stored in the cache.

```

import java.util.*;

public class RecoveryCache {

    private static Stack cacheStack = new Stack();
    private static int cacheDepth = 0;

    public static void establish(){
        cacheDepth++;
        cacheStack.push(new StackElement(new HashMap(), cacheDepth));
    }

    public static void discard(){
        cacheDepth--;
        if (cacheDepth == 0) cacheStack = new Stack();
    }

    public static void restore(){
        StackElement tempSE = ((StackElement) cacheStack.pop());
        restoreMap(tempSE.map);
        while (tempSE.depth > cacheDepth){
            tempSE = ((StackElement) cacheStack.pop());
            restoreMap(tempSE.map);
        }
        cacheStack.push(new StackElement(new HashMap(), cacheDepth));
    }

    private static void restoreMap(HashMap map){
        Iterator mapiterator = map.values().iterator();
        while (mapiterator.hasNext()) {
            MapElement tempME = ((MapElement) mapiterator.next());
            try {
                tempME.field.set(tempME.object, tempME.value);
            }
            catch (IllegalAccessException e){
                e.printStackTrace();
            }
        }
    }
}

```

**Figure 1: The RecoveryCache class**

The RecoveryCache class in figure 1 provides the three methods establish(), discard(), and restore() that is used for setting up, restoring to, and discarding checkpoints. These methods are accessible for any FTM that needs the functionality of the cache.

```

import java.lang.reflect.Field;

public privileged aspect RecoveryCacheAspect {

    pointcut w_point(): set(* *);
    pointcut internal_point(): within(RecoveryCache) || within(MapElement)
        || within(StackElement);
    pointcut ext_write_point(): w_point() && !cflowbelow(w_point())
        && !cflow(internal_point());

    before() : ext_write_point(){
        if (RecoveryCache.cacheDepth > 0){
            System.out.println(thisJoinPoint.getSignature().toString());
            String key = thisJoinPoint.getSignature().toString();
            if (!(StackElement)
                RecoveryCache.cacheStack.peek().map.containsKey(key)){
                String fieldName =
                    key.substring(key.lastIndexOf('.')+1, key.length());
                Object thisObject = thisJoinPoint.getThis();
                Object value = null;
                Field field = null;
                try{
                    field = thisObject.getClass().
                        getDeclaredField(fieldName);
                    field.setAccessible(true);
                    value = field.get(thisObject);
                }
            }
        }
    }
}

```



reason for this is that since C++ allows user defined operators it is not straight forward to define exactly what these joinpoints should match. A more comprehensive discussion on the set and get joinpoints can be found in section 5.

One problem with the *set* joinpoint of AspectJ is that it does not provide a reference to the variable being set. It has access to the value that is to be written to the variable but can not access the old value that needs to be saved to the cache. This is not in any way a fundamental problem with AspectJ and we propose this addition to the language. As long as this is not supported the recovery cache can still be implemented with the use of run-time reflection since the joinpoint provides both the name of the variable being set and a reference to the object to which it belongs. This is the approach used in the above implementation.

The aspect oriented implementation has a number of advantages compared to earlier ones. As with any AO implementation the recovery cache places no restriction on the target application. This means that the programmer do not need to declare which variables should be stored in the cache but can rely on that all changes to the system state will be reverted when a restore to the checkpoint is made. Because of this standard APIs can be used when writing fault tolerant software since the APIs are also affected by the aspect. In the case of Java it can even be applied to COTS components where the source code is not available, since the weaving is done on the compiled byte code. The fact that the implementation does not rely on overloading the assignment operator makes it applicable to languages that lack user defined operator capabilities, as is the case with

Java. Even though a large number of checkpoint implementations, even incremental ones [18], has been published for Java, the one presented here is the first published implementation of an incremental recovery cache for Java.

## 4.2 Time-redundant execution

One technique for detecting and masking transient errors is that of using time-redundant execution [19]. By executing a method two times and comparing the results an error originating from a transient fault can be detected. If the method is executed a third time the fault can be masked by voting between the three runs. The implementation in figure 4 is the basic version that detects an error and throws an exception to indicate that a fault has occurred. The error masking version follows the same structure but executes the method a third time and calls a voting algorithm instead of throwing an exception.

```
public abstract privileged aspect TimeRedundantExecution{
    abstract pointcut TimeRedundantMethod();
    Object around() : TimeRedundantMethod(){
        RecoveryCache.establish();
        Object r1 = proceed();
        RecoveryCache.restore();
        Object r2 = proceed();
        if(!r1.equals(r2)) throw new TransientFaultException;
        RecoveryCache.discard();
        return r1;
    }
}
```

**Figure 4:** Time-redundant execution implementation

The figure 4 code implements a general Time-redundancy aspect that can be applied to any method. The implementation makes use of the recovery cache to set up a checkpoint and return to the previous state before each run. It is very simple to apply this aspect to a target program since it is only a matter of declaring which method(s) should be executed in a time-redundant manner. This is done by instantiating the abstract pointcut as shown in figure 5.

```
public privileged aspect aTRE extends TimeRedundantExecution{
    pointcut TimeRedundantMethod():
```

```

    execution (anyType AnyPackage.anyClass.anyMethod (anyTypes) );
}

```

**Figure 5:** Time-redundant execution instantiation

The main advantage with this implementation is that it is implemented in a generic way and allows the programmer to simply apply it to any method. This is not possible in an OO only language that requires that you write wrapper methods to all methods individually. This and the fact that the fault tolerance code is completely separated from the primary function modules make it a very compelling approach to building this type of mechanisms.

### 4.3 Executable assertions

Executable assertions [20] are used to monitor the properties of values stored in variables or passed to and from methods. The technique is used to discovery erroneous or unintended system states that can origin from both software and hardware related faults. Executable assertions do not benefit from one common general AOP implementation as the structure of the AOP joinpoints and advice in itself gives an ideal environment for implementing each assertion. Assertions that are application specific can be implemented directly as a custom build aspect, and ones that are common, as e.g. assuring non null inputs to methods, can be implemented as a reusable aspect and applied to all applicable methods.

```

public privileged aspect ExecutableAssertion {
    pointcut assertionMethod(int i):
        execution (int Aclass.aMethod(int)) && args (i);

    before (int i): assertionMethod(i){
        if (i=0) throw assertionFailedException();
    }

    after (int i) returning (int r): assertionMethod(i){
        int r = proceed();
        if (r<0 && i>=0) throw assertionFailedException();
    }
}

```

**Figure 6:** Executable assertion example implementation

The execution joinpoint should be used to assert method arguments and results. In figure 6 the example code asserts that the argument value is non zero and that a non negative input should yield a non negative output. Also, the set joinpoint can be used to assert variable properties. The example in figure 7 asserts the counter property of a variable, i.e. that it is only reset or raised by one. Since this is not a generic implementation the old value of the variable can be accessed directly by name. To implement a generic version that can be reused and applied to any variable the same reflective approach as with the recovery cache must be used. See section 5 for a discussion about the availability of the set joinpoint.

```
public privileged aspect CounterAssertion {
    pointcut counter(AClass c, int i):
        set(int AClass.aMethod.aField) && args(i) && this(c);
    before (AClass c, int i): counter(i){
        if (i!=0 || i-c.aField!=1) throw assertionFailedException();
    }
}
```

**Figure 7:** Counter property assertion implementation

The advantages with using AOP for executable assertions is, apart from the separation of function and assertion code, the possibility for reuse of common assertions. In this context it can be noted that regardless whether AOP or any other technique is used for implementing executable assertions the code should either be generated from, or statically asserted against OCL (Object Constraint Language) specifications in the same way as class stubs are generated from UML class diagrams.

#### **4.4 Recovery blocks**

Recovery blocks [21,22] is a structured way of adding software fault tolerance to an application based on the concept of design diversity. A recovery block consists of an acceptance test that verifies the output of an algorithm and one or more alternative

implementations that is executed should the test fail. The Recovery block in figure 8 has one alternative algorithm but the implementations with more follows the same basic structure.

```
public abstract privileged aspect RecoveryBlock{
    abstract pointcut RecoveryBlockMethod();
    abstract boolean acceptanceTest(Object[] args, Object result);
    abstract Object alternativeImp(Object[] args);

    Object around() : RecoveryBlockMethod(){
        RecoveryCache.establish();
        Object r1 = proceed();
        if (acceptanceTest(thisJoinPoint.getArgs(), r1)){
            RecoveryCache.discard();
            return r1;
        }
        else{
            RecoveryCache.restore();
            Object r2 = alternativeImp(thisJoinPoint.getArgs());
            If (acceptanceTest(thisJoinPoint.getArgs(), r2)){
                RecoveryCache.discard();
                return r2;
            }
            else throw new SoftwareFaultException;
        }
    }
}
```

**Figure 8:** Recovery block implementation

The Recovery Block is applied to an algorithm (encapsulated in a method) by both instantiating the pointcut, and implementing the acceptance test and alternative algorithm as shown in figure 9.

```
public privileged aspect aRB extends RecoveryBlock{
    pointcut RecoveryBlockMethod():
        execution(anyType AnyPackage.anyClass.anyMethod(anyTypes));

    boolean acceptanceTest(Object[] args, Object result){
        // Implementation of acceptanceTest
    }

    Object alternativeImp(Object[] args){
        // alternative implementation of method.
    }
}
```

**Figure 9:** Recovery block instantiation

A recovery block is a highly application specific FTM and as such offers small opportunities for reuse. Hence it benefits less than most other FTMs from a general implementation and also, although the AOP approach separates it from the primary function syntactically, remains semantically tangled.

The advantages with using AOP for recovery blocks are still twofold. The syntactical separation is in it self is a good thing that adds flexibility and reusability. The other reason for using AOP is that recovery blocks relies on the recovery cache for doing backward error recovery and hence benefits strongly from the fact that this can be so effectively implemented with AOP.

## 4.5 Control flow checking

Control flow checking [23] is a very good example on both the advantages and limitations with present day AOP languages. The mechanism is used to detect errors in the program flow caused by illegal branches. Such a branch could e.g. be caused by transient faults that affect the program counter. The mechanism is built on the principle that if a program enters a branch free sequence of the code it must be the same sequence that it exits the next time it exits a sequence. An identifier that is unique to each sequence is placed in the beginning and in the end of the sequence. When entering a sequence the identifier is pushed on the stack and when exiting pulled and compared with the identifier placed at the end. If they don't match a fault has occurred.

```
privileged aspect ControlFlowChecking{

    pointcut ControlFlowExe(): execution(* *.*(..) && !execution(* *.main(..)
        &&!within(ControlFlowChecking));
    pointcut ControlFlowCall(): call(* *.*(..) && !call(* *.main(..)
        && !within(ControlFlowChecking));

    Stack s = new Stack();

    before() : ControlFlowCall(){
        s.push(thisJoinPoint.getSignature().toString());
    }

    before() : ControlFlowExe(){
        if (!(thisJoinPoint.getSignature().toString()).equals(s.peek())){
            throw new TransientFaultException();
        }
    }

    after() : ControlFlowExe(){
        if (!(thisJoinPoint.getSignature().toString()).equals(s.peek())){
            throw new TransientFaultException();
        }
    }

    after() : ControlFlowCall(){
```

```

        if (!(thisJoinPoint.getSignature().toString()).equals(s.pop())){
            throw new TransientFaultException();
        }
    }
}

```

**Figure 10:** Control flow checking implementation

Since present day AOP languages lack statement level joinpoints code can not be added to specific points within a method. Hence a mechanism as described above can not be implemented. What can be done is to construct a mechanism that do the verification on method level. The implementation in figure 10 verifies that the method entered is the one that was called. When reaching the end of a method body it verifies that it is the same method that was entered and finally after returning it verifies that it was the same method that was returned from. This will capture most of the errors found by the statement level mechanism but miss “small” illegal branches within the body of the method. This is actually the same problem as with the statement level mechanism that misses faults that cause a branch within the branch free sequence. Comparison of the effectiveness of the statement versus method level control flow checking is beyond the scope of this paper since the mechanism was presented here in order to illustrate the capabilities of AOP.

The main advantage with AOP when implementing control flow checking is that, since it is a strictly systematic FTM it has been implemented as a reusable aspect that can be applied to any target program without modification. The one limitation is the lack of statement level joinpoints.

## 5 Limitations

Although AOP is well suited for the implementing a large amount of FTMs there are a few limitations and concerns. One limitation, not with AOP as a concept, but with the

current implementation of AspectC++ is the lack of set and get joinpoints that make monitoring of object data members impossible. Although less efficient, checkpointing can still be implemented by storing object data at the checkpoint instead of monitoring changes. In this case the object initialization or constructor joinpoints can be used to allow access to all live objects. It is however impossible to implement executable assertions that assert variable properties since these require that the assertion is triggered by variable change. Another FTM that was not included in this paper due to space restrictions and that is dependent on both the set and get joinpoint is that of data redundancy. Since non of these FTMs can be efficiently implemented without monitoring object data members, any AOP language used for implementing node level fault tolerance must support the set and get joinpoints.

Another limitation illustrated by the control flow checking example is the lack of statement level joinpoints. This is a limitation shared with earlier metaprogramming techniques [1] and is not likely to be included in general purpose AOP languages. A general purpose AOP language could be extended with such joinpoints in order to suit fault tolerance needs but it is not necessarily a desirable solution. Statement level joinpoints breaks the encapsulation property of objects totally and could easily be misused.

## **6 Conclusion**

In this paper it is shown that aspect oriented programming (AOP) is well suited for implementing node level fault tolerance as it captures the crosscutting properties of non functional requirements. The presented AspectJ implementations of common fault

tolerance mechanisms (FTMs) demonstrate that the paradigm is feasible and results in a small amount of redundant code. This is further illustrated by the fact that the complete code for all presented FTMs are included within the size restriction of this paper. Also the primary functions modules are unaffected by the FTMs. Both systematic and applications specific FTMs benefits from the concept and can therefore be implemented in a uniform way by using AOP.

This paper shows that AspectJ is today feasible for implementing fault tolerance in Java applications. The work on AspectJ is presently focusing on extending the language with generic primitives and making it compatible with the Java 1.5 generics. This will further increase the flexibility of the language and make it an even more attractive platform.

In the case of AspectC++ it is shown that it is not presently an alternative to earlier metaprogramming platforms like OpenC++ [24] for implementing node level fault tolerance. However, when the ability to monitor object data members is added to the language it will have the same capabilities as AspectJ and hence be an excellent choice for the task.

Taking into account the potential performance benefits, the tool support and the easy to understand programming languages AOP has the potential to be the better choice when implementing node level fault tolerance.

## 7 References

- [1] J. Fabre, V. Nicomette, T. Perennou, R. J. Stroud, Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. *Proceedings of the 25th IEEE Symposium on Fault Tolerant Computing Systems*, 1995
- [2] M.L.B Lisboa. A New Trend on the Development of Fault-Tolerant Applications: Software Meta-Level Architectures. In *Proceedings of International Workshop on Dependable Computing and its Applications (IFIP'98)*, 1998.
- [3] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29-32, 2001.
- [4] A. Gal, O. Spinczyk, and W. Schröder Preikschat: On Aspect-Orientation in Distributed Real-Time Dependable Systems. In *Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 7-9, 2002, pp. 261-270.
- [5] J. Herrero, F. Sanchez, and M. Toro: Fault Tolerance AOP Approach. In *Proceedings of the International Workshop on Aspect-Oriented Programming and Separation of Concerns*, Lancaster University, UK, 24 August, 2001, pp.44-52
- [6] J. Fabry. *A Framework for replication of objects using Aspect-Oriented Programming*. Phd Thesis 1998. University of Brussel.
- [7] Herrero J.L., Sanchez F., Toro M.: Fault tolerance as an aspect using JReplia. In *Proceedings of the Eighth IEEE Workshop on Future trends of Distributed Computing Systems*, 31 Oct.-2 Nov, 2001, pp.201 – 207
- [8] Szentivanyi D., Nadjm-Tehrani S.: Aspects for improvement of performance in fault-tolerant software. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing*, 3-5 March, 2004, pp.283 - 291
- [9] A. Gal, O. Spinczyk, and W. Schroder Preikschat. On aspect-orientation in distributed real-time dependable systems. *The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, January 2002.
- [10] Sullivan, Gregory T.: "Aspect-Oriented Programming using Reflection", *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Oct. 14-18, 2001, Tampa Bay, Florida, USA
- [11] The AspectJ project homepage. <http://eclipse.org/aspectj/>

[12] The International Conference on Aspect-Oriented Software Development list of research projects developing AOSD technologies. <http://aosd.net/technology/research.php>

[13] The AspectC++ project homepage. <http://www.aspectc.org/>

[14] Remi Douence, Thomas Fritz, Nicolas Lorient, Jean-Marc Menaud, Marc Segura-Devillechaise and Mario Sudholt *An expressive aspect language for system applications with Arachne*, in Proceedings of the 4th international conference on Aspect-oriented software development, ACM Press, Chicago, USA, March 2005.

[15] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering, pages 88–98, Vienna, Austria, 2001.

[16] Rodgers, P.; Wellings, A.J.: “An Incremental Recovery Cache Supporting Software Fault Tolerance”, in *Reliable Software Technologies - Ada-Europe’99, Santander, Spain, June 7–11, 1999, Lecture Notes in Computer Science* 1622, pp. 385–396, 1999.

[17]. Rubira-Calsavara, C. M. F., Stroud, R. J.: Forward and Backward Error Recovery in C++. *Object-Oriented Systems*. **1**,1 (1994) 61-86

[18] J. Lawall and G. Muller. Efficient incremental checkpointing of Java programs. In *Proceedings of the International Conference on Dependable Systems and Networks*, New York, NY, USA, June 2000. IEEE.

[19] A. Damm, “The effectiveness of software error-detection mechanisms in real-time operating systems”, *FTCS Digest of Papers. 16<sup>th</sup> Annual International Symposium on Fault-Tolerant Computing Systems*, Washington, DC, USA, 1986.

[20] Saib S.H., “Executable Assertions – An Aid To Reliable Software”, *Conf. rec. 11th Asilomar Conference on Circuits Systems and Computers*, pp. 277-281, 1978

[21] Horning, J.J, et al.: “A Program Structure for Error Detection and Recovery”, in E. Gelenbe and C. Kaiser (eds.), *Lecture Notes in Computer Science* **16**, pp. 171–187, Springer, 1974.

[22] Randell, B.: “System Structure for Software Fault Tolerance”, *IEEE Transactions on Software Engineering* **SE-1**(2), pp. 220–232, 1975.

[23] N. Oh, P. Shirvani, and E. J. McCluskey, “Control-Flow Checking by Software Signatures,” Center for Reliable Computing, Stanford Univ., CA, CRC-TR-00-4 (CSL TR num 00-800), May 2000.

[24] The OpenC++ project homepage. <http://opencxx.sourceforge.net/>