Computational Reflection for Embedded Java Systems

Antonella Di Stefano[1], Marco Fargetta[1], and Emiliano Tramontana[2]

[1] Dipartimento di Ingegneria Informatica e delle Telecomunicazioni,
Catania University, Viale A. Doria, 6 - 95125 Catania, Italy,
{adistefa,mfargetta}@diit.unict.it,
[2] Dipartimento di Matematica e Informatica,
Catania University, Viale A. Doria, 6 - 95125 Catania, Italy,
tramontana@dmi.unict.it

Abstract. Although Java reduces the time to market of embedded systems, for some contexts developers are still forced to consider, beside application concerns, checks and handling activities for anomalous conditions that can occur on hardware devices. Typically, applications and handling of anomalous conditions are unrelated, and developers should be provided with means to treat them separately. Reflective systems have been successfully used to separate different concerns. However, special care is required when using them in embedded systems, due to the run time overhead that they can cause. In this paper, we propose what we call selective reflective behaviour, which aims at reducing the run time overhead of reflective systems. An efficient implementation of this behaviour is also proposed, which is suitable even for embedded Java systems. The paper also presents an example of a meta level that handles anomalous conditions for the embedded systems in a production cell.

1 Introduction

In the context of embedded systems, the interest for Java technology continuously increases, and today some embedded systems available include a JVM for running Java applications. This approach allows a fast software engineering process for increasingly complex functionalities that these systems are equipped with.

Many Java features explain its diffusion in embedded systems [6]. The first and the most important of these features is that Java limits the number of errors when developing applications. This goal is achieved since it is a strongly typed object-oriented language and by removing pointers. As a second feature, the code developed for one platform can be easily transferred to others, without any change, thanks to the portability of Java.

Although there are many projects that use Java in embedded systems, inserting a full JVM in a small device where there is not much memory nor high CPU performance is complicated, and different approaches have been followed to cope with these shortcomings. Sun Microsystems experienced that a standard

JVM was inadequate and developed several Java API subsets to be adopted for embedded systems [11]. These subsets are: Personal Java for big embedded systems, J2ME (Java 2 Micro Edition) for embedded systems and KVM (Kilo Virtual Machine) for very small embedded systems. Sun Microsystems also developed a special CPU, named PicoJava [12], which is able to execute only Java bytecode (without interpretation). A very efficient alternative is to compile Java applications into native code with a tool such as gcj [5], allowing them to execute without a JVM[1]. However, gcj does not support Java run time class loading.

Many embedded systems work in an environment where the same operations execute continuously, except for a fault or other sporadic operations that alter natural execution. This is frequent in control systems and in other devices used e.g. in a production cell for aided manufacturing where there are only a few operations to do, however there could be many exceptional situations. Since there are many possible faults, the code for handling faults may be bigger than the code for normal operations. This code could even be bigger than the available memory. Moreover, inserting into an application every check for exceptional events makes the application code more intricate and long. We propose a *reflective approach* to reduce the memory used by Java applications for embedded systems and avoid mixing application related and fault handling concerns into a class. Reflection overcomes these problems, since it allows the code handling faults to be separated from application classes, so that each can be loaded selectively, only when needed.

With a reflective approach, developers can structure systems by means of two levels [8,4] (see section 2.1), one for the application and another one for concerns such as faults handling [10], distribution [3], sinchronisation [14], etc. In our approach, we separate applications from the code handling faults related with hardware, such as memory overflow, wrong positions of robotic arms, overheated physical devices, etc. This fault handling code can be considered independent of application concerns and can be developed separately. Moreover, reflection allows applications for embedded systems to be customised with new functionalities or with the handling of exceptional conditions (either with an ad-hoc portion of code, developed by the application designer, or with code that normally equips embedded systems). As a further contribution we describe a novel reflective mechanism, called *Selective Reflective Behaviour*, that we have developed to reduce the overhead caused by reflective systems. This is especially useful for embedded systems where CPUs are usually not very powerful, so their time should not be wasted. By selectively trapping control from the application, the overhead of reflective systems is paid only when some fault occurs. As explained in section 5, this overhead amounts to two methods invocations (one is used to determine which class can solve the fault and another to recover the state by performing some operations) and a class loading (when the fault happens for the first time).

[1] This is helpful when real time features need to be inserted into the application since it is only the operating system that handles them. This is a better solution than submitting bytecode to the JVM where an additional scheduler is used.

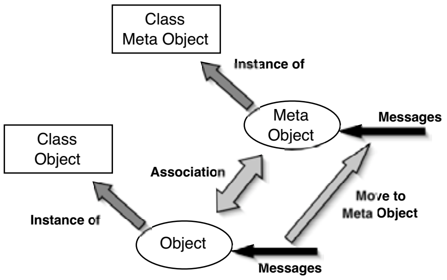

Fig. 1. Meta Object model

This paper is structured as follows. Next section introduces reflection and proposes a new approach for reducing its overhead. Section 3 describes the possible work of a meta level for embedded systems. Section 4 expresses how we have implemented a modified JVM supporting reflection. Section 5 shows a case study that we have developed using our modified JVM. Finally, section 6 draws some conclusions.

2 Supporting Reflection Efficiently

2.1 Reflection

Reflection is a programming paradigm whereby a system contains structures allowing to observe and control itself [8,4]. A reflective system can be represented by two levels: base level and meta level. The base level usually consists of an application, whereas the meta level holds the code that enriches (in some way) the application. The behaviour of object-oriented systems can be observed and controlled by means of *interception* and *inspection* of methods and fields. Interception is the mechanism used by reflective systems to capture control when e.g. methods of an object are invoked or fields are accessed. Inspection is the ability to check the state and structure of objects and classes. The twofold ability of intercepting and inspecting is called *reification* [1].

Reflection can be characterised by several models. In the *Meta Object* model, which is the one we use, application objects are associated with meta level objects, which are object instances of a special class called *Meta Object* or of its subclasses (see Fig. 1) able to intercept messages and so gain control before the associated application objects [8,4].

Some reflective Java extensions (e.g. Javassist [2], Kava [15]) use bytecode manipulation, either off-line or at load-time, to insert some statements into application classes and so allowing control to jump to the meta level when some operations are performed (e.g. a method of a class is invoked, an object is instantiated, etc.). These implementations impose a certain amount of performance degradation due to the handling of the meta level (bytecode manipulation, instantiation of meta objects, jumps to the meta level, etc.). When a meta object

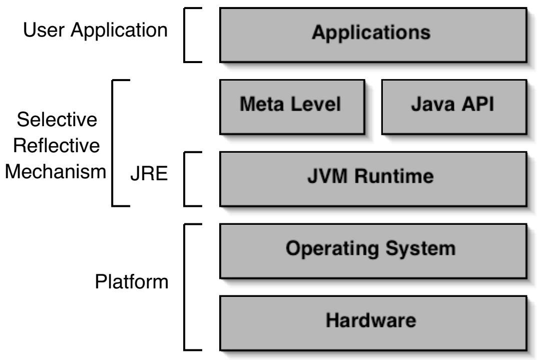

Fig. 2. JVM architecture with the selective reflective behaviour

has been associated with an object, it intercepts the operations performed on the object, causing control to jump to the meta level. As a consequence, interception slows execution down and especially when, due to system conditions, jumping to the meta level is useless this is considered an important drawback. This overhead would be reduced if jumps to the meta level were only performed when the work of the meta level was considered useful.

2.2 The Selective Reflective Behaviour

As said before, in the Meta Object model, messages sent to an application object are intercepted and redirected to the associated meta object. Then messages go from this meta object to an application object (the one invoked originally or another one) or to an appropriate meta level object. Passing control to several objects introduces some overhead.

A solution that we propose to limit this kind of overhead for reflective systems is what we call *Selective Reflective Behaviour*, which allows checking some conditions before trapping control to the meta level. We have achieved this selective reflective behaviour by modifying the version 1.2.2 of the Sun JVM[2] (see Fig. 2). The JVM is then able to decide when it is the case to enable interception, thus limiting the number of jumps to the meta level and so the overhead of reflective systems. This makes it possible to use reflective systems even in embedded environments, where CPU time cannot be wasted.

This selective reflective behaviour is useful for any reflective system, since it provides designers with means to set the degree by which interceptions have to be performed. As explained in the following section, designers communicate to the JVM when to intercept events by means of a file or a class annotation.

For embedded systems, we propose to use the meta level to handle hardware faults, thus, in normal conditions, the jump to the meta level is unnecessary. In

[2] An alternative implementation that we have produced to achieve this selective behaviour only modifies a JIT and is based on OpenJIT [9].

this case, having a selective reflective behaviour is very useful, since in normal conditions the JVM only invokes base level methods, avoiding the jump to the meta level. This reduces the overhead by one method call and the reflective system can be considered efficient for embedded systems. On the other hand, when some events move the system from its normal conditions, jumps to the meta level are enabled by the JVM. As far as memory use is concerned, in our approach application classes are smaller, since they do not implement any fault handling, which is instead implemented as meta level classes. These latter classes are only loaded into main memory when some fault has occurred and therefore in normal conditions the amount of memory used diminishes.

When developing software for embedded systems using the proposed approach, we can distinguish an application and a system designer. The application designer provides application classes implementing only those operations that have to be performed at run time, assuming that the execution does not meet anomalous conditions. The system designer provides the meta level classes implementing activities, unrelated to one particular application, that are necessary to detect anomalous conditions and to recover the system from such conditions. The system designer therefore specifies the conditions (for system variables) for which jumping to the meta level is necessary. The application designer can provide additional conditions (typically depending on application variables) that when satisfied determine a jump to the meta level. Moreover, the application designer can implement new meta level classes handling faults that were not considered by the system designer.[3]

Our use of the reflective mechanism could appear similar to the exception handling mechanism, since when a fault or other exceptional events occur the JVM executes a code handling the exception, however there are at least two main differences. The exception handling mechanism is used inside the application code by means of a `try-catch` block, instead when using the reflection mechanism the application code is not modified, and all the operations handling exceptional events are inserted into the meta level, which is detached from the application. Therefore, the reflective mechanism allows applications to be customised without being intrusive. Moreover, the reflection mechanism allows designing smaller classes, since the code handling exceptions is outside them and thus application classes for normal conditions take less time to be loaded and a smaller amount of memory once loaded. This is important for embedded systems where performance and memory are limited.

3 The Reflective Behaviour for Embedded Systems

3.1 Work of the Meta Level

Within the proposed meta level, the first operation would be to understand the conditions that caused interception. As Fig. 3 shows, this is achieved by

[3] Details on the meta level classes and their connection with application classes can be found in section 5.

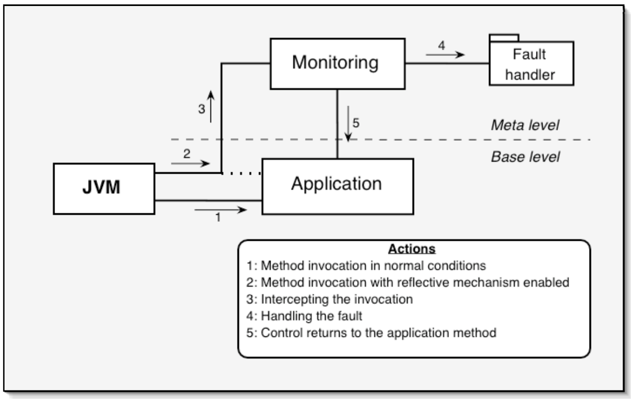
Monitoring
Fault handler
JVM
Application
Meta level
Base level
1
2
3
4
5
Actions
1: Method invocation in normal conditions
2: Method invocation with reflective mechanism enabled
3: Intercepting the invocation
4: Handling the fault
5: Control returns to the application method

Fig. 3. Selective reflective behaviour

a `Monitoring` meta object that checks the trapped object operation, the state of the application and hardware. The check aims at determining how the fault has to be handled and so the object that should receive control to recover the system. We expect the related class to be loaded by the JVM from the local file system, however it can also be downloaded from the network. Once this class is determined, an instance is created and control is passed to it. The mapping between anomalous condition and handling class is controlled by meta object `Monitoring` and a class library, on the basis of checks given by the meta level designer and of additional conditions that the application designer provides.

As an example, let us suppose that in a production cell a robotic arm has reached a bad position, then as soon as the JVM receives control, it detects such a condition and sends control to the meta level. The recovering code is implemented as meta level class `RecoveryArm` that meta object `Monitoring` knows and to which control is passed.

3.2 Enabling Interception

The selective reflective mechanism is activated whenever the JVM detects some anomalous condition. The description of the check that reveals anomalous conditions is inserted inside the JVM, however the application programmer can provide an additional check list, by means of a XML file. This file, located in the directory where application classes are found, describes: the conditions for which the meta level is allowed to trap control, the meta objects that are associated with application classes, and the operations that meta objects should trap.

In the XML file, we use the tag `<MetaClass>` to specify the name of the meta object class. The inner tag `<BaseClass>` allows the meta level class to be associated with an application class. Analogously, the tag `<Interception>` is used to specify which fields and methods have to be intercepted; and the tag

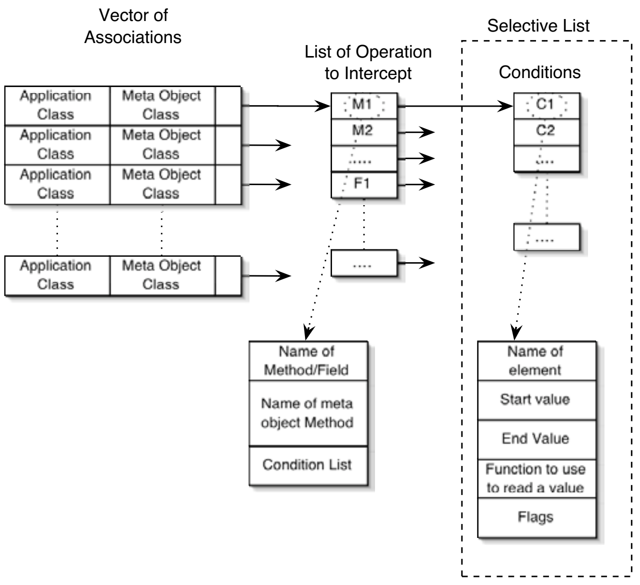

Fig. 4. Mapping data structure

`<Condition>` allows expressing the conditions that enable the meta level to trap control from the application (section 5 shows an example of such a file).

Being the XML specification separated from application and meta level classes, it can be easily modified to allow new connections between base level and meta level classes.

4 Modifying the JVM

The JVM needs extra information to support the selective reflective behaviour, in order to choose the appropriate object to give control to at run time. This extra information is organised as a list of system variables or application fields, together with the values to be checked when a switch to the meta level has to be enabled. Such a list, called `selective list`, includes system variables known by the modified JVM and information taken from a file.

As Fig. 4 shows, the `selective list` is part of a mapping data structure that allows the JVM to know for each class the corresponding meta object and the operations to be trapped. Such a structure consists of: an `association vector` that contains the association between application classes and their meta object classes; a `list of operations` that have to be intercepted; the `selective list` containing those conditions that have to be true to allow interception.

Because of its internal functioning, the JVM performs the check that may allow jumping to the meta level quickly. When an application carries out a field access or a method invocation (see lines from 4 to 17 in listing 4), the JVM searches for the location of the object in heap memory. At this point verifying the values of the fields enabling interception is very simple and fast. The list of conditions is then retrieved and if one condition is verified the method invocation is substituted (see lines from 25 to 33 in listing 4). This substitution is easily performed since the JVM holds the references to application objects and meta level objects. It is only necessary to prepare the parameters of the method that is actually called[4].

Moreover, values inside the hardware system (temperature sensors, contact sensor, etc.), or originated from the processing hardware (e.g. memory busy), are easily accessed by modifying some parts of the JVM. Generally, JVMs are implemented in C or C++ and by exploiting the features of these low level languages, they can be modified to achieve fast access to the state of some hardware devices. On the other hand, as Fig. 2 shows, Java applications cannot see the operating system nor the hardware directly and so access to the underlying hardware would be slow and difficult (except when an application uses JNI [13]).

4.1 Benefits When Modifying the JVM

The implementation of the Selective Reflective Behaviour can be efficiently achieved by either modifying the JVM or the JIT. We have produced a modified JVM, since the small amount of memory and the low CPU performance available in embedded systems makes it inadequate to use a JIT.

Some authors have proposed to use a JIT for embedded systems [7]. With such an approach instead of modifying the whole JVM, it would be possible to have a JIT that provides an application with the selective reflective behaviour. This allows the JIT to be completely removed when unnecessary, just by setting an environment variable, thus avoiding to perform any additional check for those applications that need so. We have implemented a modified version of OpenJIT that provides applications with selective reflective behaviour. However, we think that it is better to integrate this into JVMs for PCs.

By modifying a JVM to include reflective abilities, some significant benefits are achieved with respect to other approaches. Firstly, for the set up, the JVM needs only to instantiate some structures handling references to meta level classes and objects. The timeframe for such a set up is much smaller than the time needed to modify a class bytecode, which has to be performed each time a class is loaded (this approach is adopted by Kava and Javassist). Secondly, at runtime, switching to the metalevel is very fast, since control goes to the JVM whenever an application object invokes methods or access fields. Inside the JVM, executing a check to pass control to the associated meta object is faster then having two invocations (one for the application object and one for the meta object) as it happens with Kava and Javassist.

Finally, an ad-hoc version of the JVM could be installed into embedded systems to enrich them with a better support. Having a non-standard JVM

[4] The meta level method intercepting the call has a different list of parameters than the trapped method.

Listings 1.1. Modifications to `ExecuteJava`

```
bool_t ExecuteJava(unsigned char *initial_pc, ExecEnv *ee) {
  // ........
  switch (opcode) {
  case opc_invokevirtual:
    // Select the object whose method has to be called
    DO_INVOKER(FALSE);
    if (opcode == opc_invokevirtual) {
      // Select the method block and set program counter, etc.
      if (mb->fb.u.offset != 0)
        mb = mt_slot(obj_array_methodtable(o),mb->fb.u.offset);
      frame->returnpc = 0;
      optop -= args_size;
      // Jump to the code where the invocation is performed
      goto callmethod;
    } else {
        // Restore the stack
        SIZE_AND_STACK(0, 0);
    }
  // ........
  callmethod:
    sysAssert(o != 0);
    DECACHE_OPTOP();
    {
      // Retrieve the conditions whereby methods are called
      lst_condition = retrieveCondition(mb);
      // Check the list of the conditions
      while (lst_condition != NULL)
        if (check(lst, o, mb)) {
          // If this check is true the method invoked is
          // the one at the meta level
          mb = changeMethod(lst, mb, o);
          o = changeObject(lst, mb, o);
          break;
        }
      // Method invocation
      bool_t result = mb->invoker(o, mb, args_size, ee);
      // ........
}
```

should not be considered as a problem for these systems, since it is not expected to have a previous JVM installed with configured libraries, etc. nor to have applications that could be harmed by the inserted features (since applications for these embedded systems are known, compatibility tests can be run a priori). Moreover, applications are developed without using any feature of the modified JVM, in fact they are unaware of modifications, thus they would properly work on any JVM, given that faults are handled in another way.

5 Case Study

After that we have described how the selective reflective behaviour works, we use it for a case study in order to show how it can help the development of applications for embedded systems.

In modern factories there are assembly lines that are fully managed by robots. Each robot could be considered as an embedded system with its operating system and the application that controls its movements. The work of the robot is repetitive but many faults can occur and a software portion should handle them. The selective reflective behaviour is useful to simplify the development and maintenance of this software portion, thanks to the separation it enables between application and faults handling concerns.

Let us consider an application consisting of some classes that control the robot actions. The method classes are developed without needing to check the conditions of the environment or the application itself, instead this work is delegated to the meta level. In normal condition, i.e. when no problems occur to position arms, etc., the meta level is disabled and the application executes as expected. When these conditions change it is necessary to execute some code that handles the abnormal state and avoids malfunctioning.

In our example application, class `Actions` is dedicated to position the robot and includes methods: `moveArm()`, to control the arm movements; and `changePosition()`, to move the robot inside the production cell. These methods use two fields, `armPos` and `robotPos`, that hold the coordinates of arm and robot, respectively.

To handle the problems arising in exceptional conditions, a meta level is connected to the application. This meta level holds a class `Monitoring` that obtains control from the JVM when an exceptional condition is recognised. The `Monitoring` class checks the state conditions and forwards a request to a repository server specifying an identifier for the fault. The repository server returns a class able to handle the fault (see Fig. 3).

Listing 5 shows class `Monitoring`. This class implements the `MetaObject` interface that consists of two methods, `trapField()` and `trapMethod()`, which respectively intercept field access and method invocation. Two parameters are passed to `trapField()`, which identify the name of the field and the reference to the object from which control has been trapped. Method `trapMethod()` has two parameters, which determine the name of the trapped method and an array of objects, whose first element is the trapped object. The other elements of this array are the parameters passed to the original method.

To activate the selective reflective mechanism a file with the conditions to be checked is written. As shown in listing 5, in this file there could be different types of conditions enabling interception. Some conditions are used to check the arm position (expressed by means of tag `<range>`), which is given by a value inside the application class (specified by means of tag `<nameF>`). Other conditions refer to the memory and the CPU workload (expressed by means of tags `<memoryFree>` and `<workload>`, respectively). The first and second condition enable a field access to be intercepted, since they are specified within tag `<field>`, whereas

Listings 1.2. Meta level class `Monitoring`

```
class Monitoring implements MetaObject {
  Class cls;
  FaultIndex fidx;
  Object state[];
  public Monitoring(Object o[]) {
    cls = o[0].getClass();
    // Retrieve the list of fault identifiers
    fidx = Repository.getFaultIndex(cls);
    // Create a checkpoint for the object
    storeState(o);
  }
  public void trapMethod(String name, Object o[]) {
    Handle hand;
    FaultGen fg;
    // Detect the conditions that have generated the fault
    fg = checkFault(name, o);
    for(int i=0; i<fidx.length; i++)
      // Verify if this fault identifier is appropriate
      if (fg.compare(fidx[i]) {
        // Obtain handler from Repository and start execution
        hand = Repository.getHandler(fidx[i]);
        hand.startHandler(name, o);
      }
    return;
  }
  public void trapField(String name, Object o) {
    //....
  }
}
```

the third condition enables intercepting a method call, since it is specified within tag `<method>`.

When the application executes, if one condition is true, e.g. the CPU workload for the embedded system is over 30 processes in the run queue, the JVM on invocation of method `moveArm()` enables the reflective behaviour and provides control to `trapMethod()` of the `Monitoring` meta object. This meta object analyses the system conditions and invokes the `Repository` to obtain a class to handle the fault. Then control returns to the application and the meta level is disabled until a condition is true again.

6 Conclusions

The approach that allows reflection for embedded systems that we have proposed here provides at least three benefits. Firstly, application classes are not forced to include code to handle exceptional events, thus their development is easier

Listings 1.3. XML configuration file

```xml
<?xml version="1.0"?>
<MetaLevel>
  <MetaClass>
    <nameMO>Monitoring</nameMO>
    <BaseClass>
      <nameBO>Actions</nameBO>
      <Interception>
        <field>
          <nameF>arm</nameF>
          <nameFInterceptor>trapField</nameFInterceptor>
          <Condition>
            <range>
              <startValue>arm.x=30</startValue>
              <endValue>arm.x=90</endValue>
            </range>
              <!-- Enable interception for some arm values -->
            <memoryFree>1M</memoryFree>
              <!-- Enable interception if free memory is
                   less than 1MB -->
          </Condition>
        </field>
        <method>
          <nameM>moveArm</nameM>
          <nameMInterceptor>trapMethod</nameMInterceptor>
          <Condition>
            <workload>30</workload>
              <!-- Enable interception if workload is
                   over 30 processes -->
            <Sensor>
              <nameS>TemperatureSignal</nameS>
              <range>
                <startValue>50</startValue>
              </range>
                <!-- Enable interception if temperature is
                     over 50 C -->
            </Sensor>
          </Condition>
        </method>
      </Interception>
    </BaseClass>
  </MetaClass>
</MetaLevel>
```

than that in traditional approaches. Secondly, when the application is moved into a new environment, it does not need to be modified to insert the code handling possible faults. This allows moving an application in a new system in a shorter time, whereas with the traditional approach, applications have to be reengineered to insert checks. Thirdly, when a class handling a fault condition has to be changed, only the repository needs to be updated and the embedded system could execute without stopping. This simplifies updating an application and minimises downtime of the system.

The engineering approach that we have proposed reduces the effort when developing applications. We have achieved this by separating applications and fault handling by means of a novel reflective model. This separation can be considered similar to the one found in Aspect Oriented Programming, where components (which in our case are application classes) and aspects (code handling faults) are developed separately and then connected by an ad-hoc compiler called weaver. However, for the requirements of embedded systems, we needed to selectively enable at run time the code handling faults. This is not possible to achieve by simply using aspects that are always active. Moreover, we have lowered the memory used by the application by extracting faults handling from classes. The reflective mechanism ensures that this holds at run time, whereas what the run time condition of the memory is when using aspects depend on the implementation choices of the weaver used.

References

1. W. Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, 1998.
2. S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, 2000.
3. A. Di Stefano, G. Pappalardo, and E. Tramontana. Introducing Distribution into Applications: a Reflective Approach for Transparency and Dynamic Fine-Grained Object Allocation. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC'02)*, Taormina, Italy, 2002.
4. J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326, New York, NY, 1989.
5. Free Software Foundation, http://gcc.gnu.org/onlinedocs/gcj.ps.gz. *Guide to GNU gcj.*
6. V. Ivanovic and M. Mahar. Using Java in Embedded Systems. *Circuit Cellular Ink – The Computer Application Journal*, Issue 102, January 1999.
7. W. Kastner and C. Krügel. A New Approach for Java in Embedded Networks. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, Porto, Portugal, 2000.
8. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, volume 22(12) of *Sigplan Notices*, pages 147–155, Orlando, FA, 1987.

9. H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, F. Sohda, and Y. Kimura. OpenJIT Frontend System: an implementation of the reflective JIT compiler frontend. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer-Verlag, June 2000.
10. R. J. Stroud and Z. Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In C. Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.
11. Sun Microelectronics, http://java.sun.com/j2me. *Java 2 Platform, Micro Edition.*
12. Sun Microelectronics, http://www.sun.com/microelectronics/picoJava/. *picoJava^{TM}Microprocessor Core Overview.*
13. Sun Microelectronics, http://java.sun.com/docs/books/tutorial/native1.1/. *Trail: Java Native Interface.*
14. E. Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 59–78. Springer-Verlag, June 2000.
15. I. Welch and R. J. Stroud. Kava – A Reflective Java Based on Bytecode Rewriting. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 155–167. Springer-Verlag, June 2000.