User-Controlled Reflection on Join Points*

Piërre van de Laar
Embedded Systems Institute, Eindhoven, the Netherlands
Email: Pierre.van.de.Laar@esi.nl

Rob Golsteijn
NXP, Eindhoven, the Netherlands
Email: Rob.Golsteijn@nxp.com

***Abstract*—All aspect orientation languages provide a one-size-fits-all methodology for reflection on join points. However, the amount of resources necessary for this approach is too high to be applicable in the context of consumer products. In this industrial research paper, we describe a solution to this problem and prove via an experiment that it is suitable for our context. In particular, we advocate that in the context of consumer products the reflective information should be passed explicitly using dedicated reflection parameters. Furthermore, since reflective information should efficiently encode the relevant domain knowledge, the user must be in control of the type of the dedicated reflection parameters. We describe how we implemented user-controlled reflection on join points in our aspect-oriented framework AspectKoala [12] on top of the component model Koala [13]. We compare the resource consumption of different approaches to add reflective information on join points using this implementation. The difference in resource consumption clearly demonstrates the benefits of our solution for consumer products.**

***Index Terms*—Aspect orientation, reflection, join point, context, embedded system, consumer product.**

I. Introduction

This paper describes work done in the TRADER [16] project, of which NXP (formerly known as Philips Semiconductors) is the industrial partner. The research challenge of this project is to enhance reliability by improving the architecture and design of consumer products. Within TRADER, television is chosen as the representative of this kind of products.

A reliability concern is, for example, to handle calls to uninitialised software modules. This concern is scattered throughout the whole software stack as is shown in Fig. 1, which was made with AspectBrowser [17]. This scattering clearly indicates the added value of aspect orientation for our software stack. For our particular context a dedicated aspect-oriented framework, called AspectKoala [12], has been developed. It provides standard aspect orientation features like before, around, and after advices, and reflection on join points.

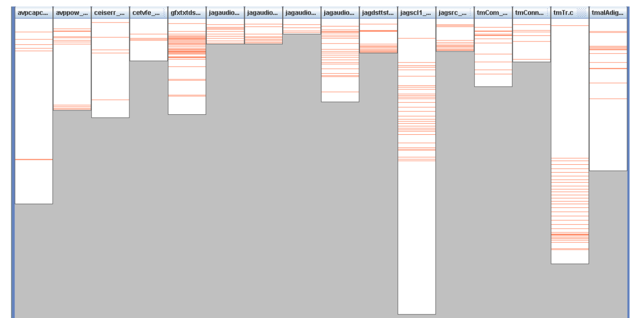

Figure 1. The scattering of a reliability concern over multiple files. Each source file is depicted as a block. Each source line dealing with this concern is colored.

Philips produces many kinds of consumer products, such as televisions, hard-disk DVD-recorders, and hi-fis. In this market, producers compete on consumer price. Due to the high volumes, a low bill of material is crucial. As a result, resources, such as memory (both ROM and RAM), CPU cycles, and bandwidth, are constrained. In this paper, we address a problem caused by these resource constraints.

We intend to use aspect orientation in consumer products. The aspect-oriented framework must support features, such as reflection on join points. Unfortunately, the standard one-size-fits-all methodology for reflection on join points ([6], [8], [15], [18]) requires more resources than affordable in consumer products. Since we wanted to apply aspect orientation in our context, we had the following objective: Develop a resource effective methodology for reflection on join points. This paper describes how we achieved this objective.

This paper is organised as follows: We start with related work in Section 2. We illustrate the importance of reflection on join points in Section 3. The architectural decisions of AspectKoala relevant for this paper are discussed in Section 4. In Section 5, we describe the background, rationale, and design of AspectKoala, and we apply it to measure in a real-world example the overhead of different alternatives for reflection on join

This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik03021 program.

Large part of this work was done while Piërre van de Laar worked at Philips Research.

© 2007 ACADEMY PUBLISHER

points. We end with a discussion, conclusions and future work in Sections 6 and 7.

II. Related Work

Reflection adds an amount of overhead in performance and resources. The overhead in performance, such as CPU consumption, has received considerable attention (see e.g., [1] and [2]). The overhead in resources, such as the consumption of ROM and RAM memory, has so far not received much attention. For consumer products both kinds of overhead are important.

Aspects can be weaved into a system either statically or dynamically. The performance loss factor of current dynamic weavers is large [4]. According to Schröder-Preikschat *et al.* [10] a dynamic weaver for embedded systems[1] seems possible in principle. However, currently all dynamic weavers targeted for embedded systems miss access to join point context information by the advice code [10]. So currently, we can only use static weaving in consumer products when we want to apply reflection on join points too.

III. Reflection On Join Points

We illustrate the importance of reflection on join points by an example dealing with tracing. The pieces of code of this example are written in the well-known languages Java [14] and AspectJ [15]. The pieces of code interact at the so-called join points. The exposed set of join points is a choice of the aspect language. AspectJ introduces, amongst others, the following join points: method execution and call, field read and write access, and object initialization.

We intend to trace method calls in the following Java class:

```
public class MyClass
{
    public void f() { /* some code */ }

    public void g() {
        f();
    }

    public static void main(String[] args) {
        MyClass mc = new MyClass();
        mc.f();
        mc.g();
    }
}
```

Figure 2. An example of a Java class.

For tracing, we write the following AspectJ aspect:

```
1      public aspect MyAspect {
2        pointcut MyClassMethodCall():
3          call (* MyClass.* (..));
4        before(): MyClassMethodCall()
5        {
6          System.out.println(
7            thisEnclosingJoinPointStaticPart
8            + " calls "+thisJoinPoint);
9        }
10     }
```

Figure 3. An aspect that traces the Java class `MyClass`.

[1] Consumer products are embedded systems.

We limit our explanation to this example. For more details on AspectJ, we refer to [15]. On line 1 of Fig. 3, an aspect called `MyAspect` is declared. It contains two elements called pointcut and before advice. Lines 2 and 3 define the pointcut, i.e., a selection of join points. This pointcut, named `MyClassMethodCall`, selects all method calls to any method in `MyClass`. Lines 4 to 9 define the before advice. This before advice specifies that before any of the join points selected by the pointcut `MyClassMethodCall`, we want to execute the code at lines 5 to 9. The before advice accesses reflective information via the two predefined objects `thisEnclosingJoinPoint-StaticPart` and `thisJoinPoint`. This reflective information serves two purposes. One, a single method can be called from multiple locations. The object `thisEnclosingJoinPointStaticPart` gives information about the current calling context. Two, a pointcut can use wildcards and hence match multiple join points. The object `thisJoinPoint` contains the information about the currently matched join point.

Compiling and running the class and aspect produces the following output:

```
1  execution(void MyClass.main(String[]))
calls call(void MyClass.f())
2  execution(void MyClass.main(String[]))
calls call(void MyClass.g())
3  execution(void MyClass.g()) calls call(void
MyClass.f())
```

Figure 4. The output of running the Java class `MyClass` with the tracing aspect `MyAspect`.

The aspect consists of only ten lines. Its output is comprehensible and effective thanks to the reflective information. As this example demonstrates, reflective information is crucial for tracing using aspect orientation. In fact, when aspect orientation is used, reflective information is always needed for identification, e.g., for error isolation, caller localization, or debugging purposes. We experienced the problem with the one-size-fits-all methodology for reflection on join points in exactly such a situation: we wanted to measure the maximum duration of each critical section in the system and, hence, we had to be able to efficiently identify each critical section.

IV. Architectural Decisions

In this section, we answer three questions necessary to add reflective information on join points in the context of consumer products. The answers are mainly driven by reliability and overhead requirements for this context.

A. How to Pass Reflective Information?

How should we pass reflective information on a join point to the advice? Two alternatives exist: implicit and explicit reflection parameters. AspectJ [15] is an example of the former, and Eos [8] and JBoss AOP [18] are examples of the latter. See also Fig. 5. The two alternatives behave different with respect to overhead, reliability, and conceptual consistency.

The overhead of passing a reflection parameter includes stack space, CPU cycles, and ROM footprint. With explicit reflection parameters, the overhead is only

© 2007 ACADEMY PUBLISHER

incurred when it is actually used by an aspect in an inheritance hierarchy. In Fig. 5 `MyAspectX`, `MyAspectY`, and `MyAspectZ`, must have an explicit reflection parameter, since reflection is used in their inheritance hierarchy by `MyAspectY`. With implicit reflection parameters, the overhead will be always incurred in general. Since, in general, aspects must be substitutable (e.g., to enable dynamic weaving or dynamic linking) and the system can consist out of multiple compilation units, thus any sub-aspect, defined in another compilation unit, could use the reflection parameter, and the advice's implicit signature must thus contain the reflection parameter to ensure compatibility at the binary level. See also Fig. 5: `MyAspectX`'s implicit signature must include the reflection parameter, since otherwise subtype `MyAspectY`, whose implicit signature includes the reflection parameter, will be incompatible at the binary level. However, by limiting a system to a single compilation unit, a weaver can analyse all aspects for the actual usage of the reflection parameter and make the overhead identical to the overhead with explicit reflection parameters. If aspects also do not need to be substitutable, as in AspectJ [15], the overhead can be made even smaller than with explicit reflection parameters. We will illustrate this by using the example aspects of Fig. 5. Given that aspects need not to be substitutable, the aspect `MyAspectZ` (and its base aspect) can be analysed in isolation. The function `f` of aspect `MyAspectZ` implements the before advice for all join points in the pointcut `MyPointCut`. Since this function `f` does not use reflection, it will not have a reflection parameter in the implicit case, yet it will have one in the explicit case. Of course, in all cases the function `f` of `MyAspectY` will have a reflection parameter.

Reliability benefits when the intentions can be verified too. With implicit reflection parameters, the intention to use reflection cannot be expressed, while the presence of an explicit reflection parameter in a function's signature specifies the intention to use reflection[2]. Verification covers both illegal usage of a reflection parameter and a reflection parameter not being used[3]. These kinds of verification for parameters are present in all modern compilers. Hence, an explicit reflection parameter allows for verification of intention to use and thus increases reliability.

For conceptual consistency, no difference between aspects and classes should exist [8]. Advices and methods become identical when implicit reflection parameters of an advice are made explicit.

An advantage of explicit over implicit reflection parameters is that only for explicit reflection parameters the type can be put under the control of the user. Whether this advantage is useful will be discussed later on in this section.

Based on these differences, we prefer explicit reflection parameters to implicit reflection parameters for consumer products. In other words, our conclusion is that reflective information on join points should be passed explicitly. Yet, what information should be passed to an advice via these explicit reflection parameters?

B. What Reflective Information to Pass?

The reflective information on the join points can range from general-purpose, i.e. for all aspects and all join points identical, to dedicated, i.e. specific for a particular set of aspects and join points. AspectJ [15], for example, uses instead of one general-purpose reflection type, three more dedicated reflection types: `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPoint-StaticPart`. AspectJ analyses the advices to determine the necessity of these three reflection parameters. Based on the result of the analysis, AspectJ limits the overhead to roughly the necessary reflective information. The overhead is still considerable, since reflection allows extracting information in the verbose human readable form, like in Fig. 4.

We compare general-purpose and dedicated reflective information with respect to overhead, reliability, and evolve-ability.

A dedicated reflection parameter needs only to satisfy a single purpose, while a general-purpose reflection parameter must serve all possible purposes. A dedicated parameter can leave out information that is not needed for its purpose and apply optimizations that are possible given its specific purpose. Hence, the overhead of memory, CPU cycles, and ROM footprint for a dedicated reflection parameter will never be larger than for a general-purpose reflection parameter. In particular, many consumer products have only the requirement to support off-line or post-mortem reflection. To satisfy this requirement, only a unique identifier per join point, comparable to IDREF in AspectCobol [6], is sufficient at run-time. Off-line, the unique identifier can be linked via a look-up table to any static reflective information, including information in verbose human readable form.

A general-purpose reflection parameter specifies the intention to use reflective information. A dedicated reflection parameter also specifies the particular part of the reflective information that is intended to be used. Hence, a dedicated reflection parameter compared to a general-purpose parameter allows for more verification and thus increases reliability.

A dedicated reflection parameter only evolves when the related system requirements change. A general-purpose reflection parameter also evolves when the environment changes. It evolves, for example, due to changes in the language and compiler, such as the addition of custom attributes, and due to changes in the reflection type based on requirements of other systems.

Based on this comparison, we prefer dedicated reflection parameters to general-purpose reflection parameters in the context of consumer products. Hence, our conclusion can only be that reflective information on join points should be passed explicitly using dedicated reflection parameters. Yet, who controls the type of these dedicated reflection parameters?

[2] The reflection parameter can be used by the aspect itself or by aspects in its inheritance hierarchy.

[3] Reflection parameter not being used is only informative when the complete inheritance hierarchy is taken into consideration.

© 2007 ACADEMY PUBLISHER

Implicit	Explicit

`abstract aspect MyAspectX`
`{`
`    abstract pointcut MyPointCut();`
 	
`    [before MyPointCut]`
`    void f (){`
`       …`
`    }`
`}`
 	
`aspect MyAspectY:MyAspectX`
`{`
`    …`
`    [before MyPointCut]`
`    void f (){`
`       Debug.WriteLine (thisJoinPoint);`
`    }`
`}`
 	
`aspect MyAspectZ:MyAspectX`
`{`
`    …`
`    [before MyPointCut]`
`    void f (){`
`       …`
`    }`
`}`
	`abstract aspect MyAspectX`
	`{`
	`    abstract pointcut MyPointCut();`
	
	`    [before MyPointCut]`
	`    void f (JoinPoint thisJoinPoint){`
	`       …`
	`    }`
	`}`
	
	`aspect MyAspectY:MyAspectX`
	`{`
	`    …`
	`    [before MyPointCut]`
	`    void f (JoinPoint thisJoinPoint){`
	`       Debug.WriteLine (thisJoinPoint);`
	`    }`
	`}`
	
	`aspect MyAspectZ:MyAspectX`
	`{`
	`    …`
	`    [before MyPointCut]`
	`    void f (JoinPoint thisJoinPoint){`
	`       …`
	`    }`
	`}`

Figure 5. Implicit versus explicit reflective parameters.

```
interface IComponentAspect
{
    bool JoinPointHasBeforeAdvice(JoinPoint jp);

    //PreCondition: JoinPointHasBeforeAdvice(jp)
    bool BeforeAdvice_UsesReflection(JoinPoint jp);

    //PreCondition: BeforeAdvice_UsesReflection(jp)
    Type BeforeAdvice_TypeOfReflectionParameter(JoinPoint jp);

    //PreCondition: BeforeAdvice_UsesReflection(jp)
    bool BeforeAdvice_IsSetupNeededForArgumentToReflectionParameter (JoinPoint jp);

    //PreCondition: BeforeAdvice_IsSetupNeededForArgumentToReflectionParameter(jp)
    Code BeforeAdvice_SetupOfArgumentToReflectionParameter (JoinPoint jp);

    //PreCondition:BeforeAdvice_UsesReflection(jp)
    Code BeforeAdvice_ArgumentToReflectionParameter(JoinPoint jp);

    //PreCondition: JoinPointHasBeforeAdvice(jp)
    Code BeforeAdvice_Code(JoinPoint jp, string reflection);

    //similar for Around and After Advice
    ...

    //Factor out commonality of Before, Around, and After advice to save resources

    //PreCondition: BeforeAdvice_UsesReflection(jp)||
    //              AroundAdvice_UsesReflection(jp)|| AfterAdvice_UsesReflection(jp)
    bool IsCommonSetupNeededForArgumentToReflectionParameter (JoinPoint jp);

    //PreCondition: IsCommonSetupNeededForArgumentToReflectionParameter(jp)
    Code CommonSetupOfArgumentToReflectionParameter(JoinPoint jp);
}
```

Figure 6. The part of the `IComponentAspect` interface relevant for reflection.

C. How to Control Reflective Information?

The type of reflective information is under control of either the user or the aspect-oriented framework, e.g., via predefined or configurable types. In the first case, the user must not only specify the type but also the value of the reflection parameter as function of the join point when the advice is called, since the framework is not knowledge-able of the reflective information. Of course, a library can support the user herein with for example

© 2007 ACADEMY PUBLISHER

string compression; bit encoding for collections; and bits concatenation.

In the case that the type of reflective information is under control of a generic aspect-oriented framework[4], domain knowledge cannot be exploited to efficiently encode the reflective information. In this case, for example, it is not possible to tune the size of the reflective type based on global product knowledge, such as the number of join points, or to let the reflection type contain Boolean fields that indicate:

- The begin or end of an interval delimited by a function pair, like begin/end critical section, enter/leave menu, open/close file, and acquire/release semaphore; or
- Whether the join point is in the application or platform part of the software.

Since, for consumer products, the reduction of overhead outweighs the additional user effort, we decided to pass reflective information in our framework explicitly using dedicated reflection parameters which types are controlled by the user. Yet, how do we implement that?

V. AspectKoala

In this section, we describe the background, rationale, and high-level design of AspectKoala. We elaborate on the design of reflection on join points in AspectKoala, and we apply AspectKoala to measure in a real-world example the overhead of different alternatives for reflection on join points.

A. Background

Software development benefits from separation of concerns, i.e., the mental ability to deal with the difficulties, the obligations, the desires, and the constraints one by one [3]. Especially, when the modularization of the reasoning can be reflected in the modularization of the software. The software in Philips televisions is currently modularised using the component model Koala [13]. Koala has many similarities with other component models, such as OMG's CORBA Component Model [11] and Microsoft's COM [9]. In Koala:

- Interfaces and components are specified in an architecture description language.
- Components can only interact with each other via interfaces.
- A component provides functionality for which it may require functionality from its environment.
- A component can provide and require multiple interfaces of the same type, since a component refers to its interfaces by instance name and not by type.
- A third party instantiates components and connects their interfaces

[4]For example, AspectC++ tailors down the reflective information according to the individual requirements of the actual advice [19] by using a super-type of the generic reflection type.

B. Rationale

Unfortunately, the modularization of the reasoning cannot always be reflected in the modularization of the software into components, as can be observed in the current software and was already shown in Fig. 1:

- Many (non-functional) concerns are not localised in one component but are scattered throughout the software; and
- Multiple concerns are tangled in one component.

To better support the modularization of the reasoning, Philips has looked at more effective modularization techniques for software, such as aspect orientation. For this purpose AspectKoala [12], an aspect-oriented framework on top of Koala, has been developed.

C. High-Level Design

AspectKoala is built on top of the abstract syntax tree of Koala, much like SourceWeave.Net [5] is built on top of CodeDOM: the abstract syntax tree of .Net. The architectural description of all components and interfaces is contained in the abstract syntax tree of Koala. AspectKoala weaves aspects into the product based on these architectural descriptions, and it is independent of the programming languages used to implement the components. The join points of AspectKoala are the execution of and calls to interface functions. Aspects in AspectKoala are compiled .Net classes that implement the `IComponentAspect` interface. This interface contains functions to specify the pointcuts, to implement the advices in the C programming language, and to declare dependencies, such as usage interfaces [7]. Hence, unlike AspectJ that provides regular expressions to specify pointcuts, in AspectKoala functions of the `IComponentAspect` interface must be implemented for this purpose. Of course, these functions can use .Net's regular expressions to specify pointcuts.

Currently, adding an aspect to Koala's component-based software consists of the following steps:

1. Write and compile an aspect[5], i.e., a class in .Net that implements the `IComponentAspect` interface.
2. Run AspectKoala to weave the aspect with a particular Koala configuration. This results in a new Koala configuration where the aspect is added.
3. Compile this new Koala configuration as usual.

In the future, we want to integrate AspectKoala with the Koala compiler.

D. Design of Reflection on Join Points

We describe the part of the `IComponentAspect` interface relevant for user-controlled reflection on join points, see also Fig. 6. The function `JoinPoint-HasKindAdvice` specifies that a join point is part of a pointcut and has a particular `Kind` of advice, i.e., before, around, or after advice. If a join point has a particular kind of advice, the function `KindAdvice_UsesReflection`

[5] Our framework is not limited to one aspect, since an aspect can instantiate multiple other aspects and control their precedence.

© 2007 ACADEMY PUBLISHER

explicitly specifies the presence of a reflection parameter at that join point for that kind of advice. When a reflection parameter is present, `KindAdvice_TypeOfReflectionParameter` specifies its user-defined type. Since the user defines the type of the reflection parameter, it also becomes his responsibility to provide an argument for it. Due to a limitation of the C programming language, i.e., C does not accept struct literals as arguments, the interface contains three functions[6] to specify the argument to the reflection parameter for a kind of advice: The function `KindAdvice_IsSetupNeededForArgumentToReflectionParameter` specifies that a setup (for a struct literal) is needed; the function `KindAdvice_SetupOfArgumentToReflectionParameter` specifies that setup when needed; and the function `KindAdvice_ArgumentToReflectionParameter` provides the actual reflection argument, which can refer to the elements introduced in the setup when present. The advice is implemented by the function `KindAdvice_Code`. The advice can access the reflective information, since its parameter called `reflection` contains the instance name of the reflection parameter in C.

```
public class MyAspect : ComponentAspect
{
    public override bool
     JoinPointHasBeforeAdvice(JoinPoint jp)
     { return true; }

    public override bool
     BeforeAdvice_UsesReflection(JoinPoint jp)
     { return true; }

    public override Type
     BeforeAdvice_TypeOfReflectionParameter
                                  (JoinPoint jp)
     { return typeof(int); }

    public override Code
     BeforeAdvice_ArgumentToReflectionParameter
                                  (JoinPoint jp)
     { return
          Code.IntValue(LookUpTableJP2UID(jp));
     }

    public override Code
     BeforeAdvice_Code
              (JoinPoint jp, string reflection)
     { return Code.Skip; }

    //identical for After Advice
    ...
}
```

Figure 7. Example aspect in AspectKoala that uses a unique id per join point as reflective information

Aspects programmed in AspectKoala usually do not implement interface `IComponentAspect` directly, but inherit from the abstract `ComponentAspect` class. This class returns `false` on all Boolean methods of `IComponentAspect`. As a result only the features that are used by an aspect must be implemented by overriding the appropriate methods. This considerably simplifies the implementation of aspects.

Fig. 7 contains an example aspect written in AspectKoala. This aspect specifies that all join points have a before advice in `JoinPointHasBeforeAdvice`. In `BeforeAdvice_UsesReflection` and `BeforeAdvice_TypeOfReflectionParameter` it specifies that reflection is used in the before advices of all join points and that the type of the reflection parameter is always integer, respectively. The implementation of `BeforeAdvice_ArgumentToReflectionParameter` specifies that the argument to the reflection parameters depends on the particular join point: The integer value as generated by the lookup-table function that assigns a unique identifier to each join point. Since the implementation of `AfterAdvice_ArgumentToReflectionParameter` is identical, the before and after advices associated with the same join point will be called with the same integer value of the reflection parameter. Finally, the aspect specifies in `BeforeAdvice_Code` that the before advice of all join points is equal to the skip-statement.

E. Overhead

AspectKoala enables us to experiment not only with the presence of a reflection parameter but also with the type of this parameter. We wrote four aspects to add tracing[7] before and after the execution of every join point in an existing configuration. The first aspect adds tracing without reflection and acts as our baseline. The second aspect adds tracing with a unique identifier per join point and is (partly) shown in Fig. 7. The third aspect adds tracing with only the function, interface, and component name per join point. The fourth aspect adds tracing with a general-purpose reflection type parameter comparable to AspectJ's `thisJoinPointStaticPart`.

We spent considerable effort to minimise the overhead of these aspects. For example, for the last two aspects as mentioned in the previous paragraph, we used look-up tables to factor out commonalities and thus to save resources. However, we only looked at one specific but representative product in this experiment. And, we did not include custom attributes in the last aspect. The numbers we present are thus not hard values, but mere indications.

[6] Since our framework is targeted at a resource limited environment, the `IComponentAspect` interface also contains two functions to factor out the commonalities in the setups of the reflection argument for the before, around, and after advices: `IsCommonSetupNeededForArgumentToReflectionParameter` and `CommonSetupOfArgumentToReflectionParameter`.

[7] Only the infrastructure needed to trace is added, no tracing information is collected. I.e., the advice contains no statements.

© 2007 ACADEMY PUBLISHER

TABLE I. THE PERCENTAGES OF INCREASE IN CPU LOAD AND ROM FOOTPRINT DUE TO DIFFERENT TYPES OF REFLECTION PARAMETERS.

reflection type	Increase	
	CPU load	ROM footprint
unique identifier	4.7 %	1.8 %
names	5.5 %	10 %
general-purpose	5.5 %	36 %

With these aspects, we measured the increase in CPU load and ROM footprint (i.e., the size of the binary code and the constant/static data) due to reflection. See also Table I. Note that this increase is on top of the increase already introduced by tracing for a release build as measured for the baseline, our first aspect. Since we consider an increase above 15% in ROM footprint as not acceptable for consumer products, we cannot apply a general-purpose reflection parameter in our context. Furthermore, as shown in Table I, the difference between "unique identifier" and "names" is considerable. This justifies that we put the user in control of the reflective information such that the user can incorporate domain knowledge to save memory, CPU cycles, and bandwidth for his particular application.

VI. DISCUSSION

For many applications, we see advantages of using unique identifiers as part of the reflective information. To give two examples:

- Applications that trace static reflective information over a large time-window, and
- Remote diagnostic and control applications with limited bandwidth.

So, we are surprised that so few other aspect-oriented frameworks (to our knowledge only AspectCobol [6]) are currently offering unique identifiers as part of the reflective information.

In our experiment, the unique identifiers were handed out at compile time. For products composed of binary COTS components this cannot be done so efficiently at compile time, e.g., in COM [9] GUIDs are 128 bits. For these products, smaller unique identifiers can be handed out at load, initialization, or even run-time.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we indicated, for the context of consumer products, that resources are constrained and that aspect orientation has added value. We also illustrated that reflection on join points is crucial for aspect orientation. Unfortunately, the standard one-size-fits-all methodology for reflection on join points ([6], [8], [15], [18]) requires more resources than are affordable for consumer products. Since we wanted to apply aspect orientation in our context, we had the following objective: Develop a resource effective design for reflection on join points. We achieved our objective by explicitly passing dedicated reflection parameters. Furthermore, we enabled the user to control the reflective information, such that the user can incorporate domain knowledge to save memory, CPU cycles, and bandwidth for his particular application. We showed that, with user-controlled reflection on join points, we are able, for a given application, to reduce resource usage to an affordable level without sacrificing the necessary expressive power.

In the future, we might develop our own aspect language to move from compilation to interpretation of aspects. Yet, we foresee our framework AspectKoala to remain hybrid supporting both compilation and interpretation. We expect this since domain knowledge should not be part of the framework, while reflective information should be generated such that it efficiently encodes the relevant domain knowledge. Putting the user in control of the encoding of the reflective information is a fundamental difference between AspectKoala and all other frameworks known to us.

ACKNOWLEDGEMENT

We would like to thank Michael Borth, Joris Dobbelsteen, Teun Hendriks, Koen van Langen, Mathijs Opdam, Maarten Pennings, Bedir Tekinerdogan, Aleksandra Tesanovic, and the anonymous referees for their useful remarks on earlier versions of this paper.

REFERENCES

[1] Cazzola, W., *SmartReflection: Efficient Introspection in Java*, Journal of Object Technology, 117-132, Vol. 3 (11), 2004.

[2] Chiba, S., *Implementation Techniques for Efficient Reflective Languages*, University of Tokyo, Technical Report 97-06, 1997.

[3] Dijkstra, E. W., *A Discipline of Programming*, ISBN 0613924118, 1976.

[4] Haupt, M., and Mezini, M., *Micro-Measurements for Dynamic Aspect-Oriented Systems*, In *NetObjectDays (NODe '04)* (Erfurt, Germany), vol. 3263 of LNCS, 81-96, 2004.

[5] Jackson, A., and Clarke, S., SourceWeave.NET: Cross-Language Aspect-Oriented Programming, In *Generative Programming and Component Engineering* (Vancouver, Canada), 115-135, 2004.

[6] Lämmel, R., and De Schutter, K., What does aspect-oriented programming mean to Cobol?, In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)* (Chicago, Illinois, USA), 99-110, 2005.

[7] Lieberherr, K., Lorenz, D., and Mezini, M., *Programming with Aspectual Components*, Technical Report, NU-CSS-99-01, March 1999. Available at http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html

[8] Rajan, H., and Sullivan, K. J., Classpects: Unifying Aspect- and Object-Oriented Language Design, In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, Missouri, USA), 2005, 59–68.

[9] Rogerson, D., *Inside COM: Microsoft's Component Object Model*, Microsoft Press, ISBN 1-572-31349-8, 1997.

© 2007 ACADEMY PUBLISHER

[10] Schröder-Preikschat, W., Lohmann, D., Scheler, F., Gilani, W., and Spinczyk, O., *Static and Dynamic Weaving in System Software with AspectC++*, Proceedings of the 39th Hawaii International Conference on System Sciences, 2006.
[11] Siegel, J., *CORBA 3: Fundamentals and Programming*, OMG Press, ISBN 0-471-29518-3, 2000.
[12] van de Laar, P., *Combining component-based and aspect-oriented software development in a resource constrained environment*, Philips, Technical Note PR-TN 2006/00648. Available at http://www.extra.research.philips.com/publ/rep/nl-ur/PR-TN2006-00648.pdf
[13] Van Ommering, R., *Building Product Populations with Software Components*, PhD Thesis, University of Groningen, The Netherlands, ISBN 90-74445-64-0, 2004. Available at http://irs.ub.rug.nl/ppn/27516956
[14] Arnold, K., and Gosling, J., *The Java Programming Language*, Addison-Wesley, ISBN 0201634554, 1996.
[15] AspectJ Home Page. http://www.aspectj.org
[16] TRADER Home Page. http://www.esi.nl/trader
[17] Griswold, W., Kato, Y., and Yuan,J., *AspectBrowser: Tool Support for Managing Dispersed Aspects*, CS1999-0640, 1999. Available at http://citeseer.ist.psu.edu/griswold99aspect.html
[18] JBoss AOP Home Page. http://labs.jboss.com/jbossaop
[19] Spinczyk, O., Lohmann, D., and Urban, M., *Advances in AOP with AspectC++*, In *Software Methodologies, Tools and Techniques (SoMeT 2005)* (Tokyo, Japan), 33-53, 2005.

Piërre van de Laar was born in Bavel, the Netherlands, in 1971. Between 1989 and 1998 he studied at the Catholic University of Nijmegen and was awarded his master degree (cum laude) in both theoretical and computational physics in 1994 and his PhD on 'Selection in Neural Information Processing' in 1999.

From 1998 until 2006, he was employed by Philips Research in Eindhoven and spent two years following the course 'from doctor in science to computational scientist'. From 2000 onwards, he investigated the exploitation of architecture description languages for product families, visualization, verification, aspect-orientation, and dependability.

Since 2006, he has been employed by the Embedded Systems Institute as a Research Fellow. After performing a study for DaimlerChrysler, he became a member of the Darwin project. Darwin is a collaborative research project between the Embedded Systems Institute, Philips Medical Systems, Philips Research, and five Dutch universities (Delft, Eindhoven, Groningen, Twente, and the Free University of Amsterdam), and aims to improve the evolvability of complex systems.

Rob Golsteijn was born in Sittard, the Netherlands, in 1972. Between 1991 and 1997 he studied at the Eindhoven University of Technology and was awarded his master degree in Computing Science. In 1999 he completed the two-years postgraduate Software Technology program at the Stan Ackermans Institute, the Netherlands.

Since 1999, he has worked at NXP, formerly known as Philips Semiconductors, and specialised in embedded software development of television platforms, products, and product families. He was a member of the infrastructure software group that focuses on operating system and drivers. His roles in this group included software engineer, team leader, and chairman of the change control board.

Currently, Rob is investigating the use of aspect-orientation for monitoring, visualizing, and analysing complex run-time system behaviour as member of the TRADER project. TRADER is an industrial research project in which the Embedded Systems Institute, NXP, TASS, DTI, IMEC and three Dutch universities (Delft, Leiden, Twente) participate and which focuses on reliability of resource-constrained consumer devices.