NOTICE: This is the author's version of a work accepted for publication by World Scientific. Changes resulting from the publishing process, including peer review, editing, corrections, structural formatting and other quality control mechanisms, may not be reflected in this document. A definitive version was subsequently published in the International Journal of Software Engineering and Knowledge Engineering, Volume 21, Issue 7, pp. 891–929, November 2011.

International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

THE DSAW ASPECT-ORIENTED SOFTWARE DEVELOPMENT PLATFORM

FRANCISCO ORTIN*
LUIS VINUESA†
Computer Science Department, University of Oviedo
Calvo Sotelo s/n, Oviedo, 33007, Spain

JOSE M. FELIX‡
Principality of Asturias, Computer Science Department
C/ Coronel Aranda 2, Oviedo, 33005, Spain

Received (9 October 2009)
Revised (19 July 2010)
Accepted (7 April 2011)

Aspect-Oriented Software Development (AOSD) provides systematic means to modularize crosscutting concerns in software development. Common AOSD benefits are a higher level of abstraction, concern reuse, better legibility, and software maintainability improvement. In AOSD, static weaving implementations commonly obtain better runtime performance, whereas dynamic weaving provides runtime application adaptiveness and a valuable aid in software development. Since both approaches provide benefits, we have developed a Dynamic and Static Aspect Weaving (DSAW) platform that supports both kinds of weavers: a full dynamic one to offer high dynamic adaptiveness and a static one to obtain better runtime performance when the application is deployed. Furthermore, both weaving techniques can even be used simultaneously in the same application. Depending on the adaptiveness requirements and the life cycle stage, the programmer could change from one type of weaving to the other without performing any modification in the source code of either components or aspects. Therefore, DSAW provides the separation of the *dynamism* (weaving-time) concern in the aspect-oriented software development process. Moreover, our platform also supports a wide set of join-points and is language and platform neutral. A detailed assessment has revealed that the DSAW platform provides a competitive alternative to develop aspect oriented software.

Keywords: Aspect-Oriented Software Development, Static Weaving, Dynamic Weaving, Separation of Concerns, Dynamism, Computational Reflection, Language Neutrality

* *ortin@lsi.uniovi.es*
http://www.di.uniovi.es/∼ortin

† *vinuesa@uniovi.es*
http://www.di.uniovi.es/∼vinuesa

‡ *jmanuelfr@princast.es*

1. Introduction

Aspect Oriented Software Development (AOSD) [1] is a concrete approach of the *Separation of Concerns* (SoC) principle [2]. AOSD offers a direct support to modularize different concerns that cut across system software. The modularization of crosscutting concerns prevents tangling of the application source code, making it easier to debug, maintain and modify [3]. Typical examples of cross-cutting concerns are persistence, authentication, logging and tracing [4].

The process of integrating aspects into the main application code is called *weaving* and an *aspect weaver* (or simply *weaver*) is the tool that performs it [1]. The weaving process can be performed statically (compile time or load time) or dynamically (at runtime).

Numerous programming environments that support AOSD only employ static weavers. Once the final application has been compiled, woven and executed, it is not possible to add new aspects or remove existing ones at runtime. However, there are specific scenarios where it is necessary to adapt running applications in response to runtime emerging requirements. For example, dynamic weaving has been used in handling *Quality of Service* (QoS) requirements in CORBA distributed systems [5], managing web cache prefetching [6], balancing the load of RMI applications [7], and changing the control policy of distributed systems [8]. In these cases, dynamic weavers are a powerful tool for building runtime adaptable software.

Both dynamic and static weavers have pros and cons. One of the main benefits of static weaving is runtime performance. Since the combination of components and aspects is performed prior to application execution, there is little performance cost compared to traditional object-oriented development [9, 10]. In contrast, runtime weaving (and unweaving) performed by dynamic AOSD tools commonly implies a runtime performance penalty, but it provides higher flexibility in the development of software. Applications can be adapted at runtime by dynamically adding or removing aspects that customize the application behavior. Moreover, the dynamic adaptation mechanism is preferable while developing aspect-oriented applications, because it facilitates interactive debugging following an *edit-and-continue* development. The *edit-and-continue* (also known as *fix-and-continue* [11]) debugging scheme refers to the ability to detect an error at runtime, modify the original code of aspects, recompile them, weave the running application, and continue the execution of the adapted system [12].

Previous works have identified the appropriateness of integrating both kinds of weavers in the same development environment [13, 9, 14], obtaining the benefits of both approaches in the software development process. Moreover, if the AOSD platform is appropriately designed, aspects could be changed from dynamic to static and vice versa, without changing their source code. Dynamic weaving could be used when the system is being tested to make software development easier [12]. Upon deployment, aspects that do not need to be adapted at runtime can be woven statically to improve the runtime performance of the whole system. Those that

require dynamic (un)weaving can be used together with static ones.

In this paper we present DSAW (Dynamic and Static Aspect Weaver), an aspect platform that supports both static and dynamic weaving. DSAW offers the runtime performance of static weaving, and the dynamism and interactive development of full dynamic weaving. The main contributions of this work are:

(1) The creation of an AOSD platform that offers the benefits of both dynamic and static weaving in a transparent way.
(2) An extension of the *Common Aspect Semantics Base* (CASB) [15] to specify a hybrid dynamic and static aspect weaving system.
(3) A qualitative and quantitative evaluation of existing static- and dynamic-weaving aspect-oriented platforms.

In DSAW, aspects and components can be developed regardless of the dynamism they require, making possible the transition from dynamic to static weaving (and the other way around), without any change in their source code. At first stages of the software development process, dynamic weaving could be used to facilitate interactive application debugging. At deployment, if no runtime adaptation is required, aspects can be statically woven to obtain a better runtime performance. Dynamic aspects can also be used in the released application if the system requires dynamic adaptiveness. Therefore, DSAW applies the *Separation of Concerns* principle: the *dynamism* concern has been separated from the aspect-oriented software development process.

DSAW is a language neutral system, allowing the programmer to use any high-level programming language. It is also platform independent, because its weaver does not rely on specific features of a particular operating system. In addition, we specify its semantics in order to describe its behavior independently of any implementation issue.

The set of join-points offered by DSAW is wider than most existing dynamic weaving tools, and its runtime performance is competitive in both static and dynamic weaving scenarios. Finally, our platform is based on the standard specification of a virtual machine. This makes DSAW portable, being able to be run on any standard implementation.

The rest of this paper is structured as follows. In the next section we present the basis of Aspect-Oriented Software Development and dynamic AOSD is described in Section 3. The semantics of DSAW is depicted in Section 4 and its architecture is presented in Section 5. Afterwards, the system design is described (Section 6). DSAW is qualitatively and quantitatively evaluated in Section 7, and the related work is analyzed Section 8. Finally, Section 9 presents conclusions and future work.

2. Aspect-Oriented Software Development

In many cases, significant concerns in software applications are not easily expressed in a modular way. Examples of such concerns are transactions, security, logging or

persistence. With the classical object-oriented paradigm, the code that addresses these concerns is often spread out over many parts of the application. The *Separation of Concerns* principle [3, 2] manages the complexity of software development, separating concerns whose implementation would otherwise be scattered over several modules and tangled with the code of other concerns. Major benefits of the *Separation of Concerns* principle are a higher level of abstraction, concern reuse, higher legibility of each concern in isolation, and software maintainability improvement [2].

Figure 1 shows a credit card processing application that will be used as a motivating example throughout the paper. This figure illustrates part of the initial source code, where different concerns are tangled with the core application. Three different concerns are identified: the core functionality (the payment, which first validates the card and then performs the transfer), a logging concern (using Apache log4net [16]), and a profiling concern (that measures the time needed to execute the method). Each concern is shown in a different color. In this object-oriented program, source code of the logging and profiling concerns is tangled with the business logic, and spread out over many parts of the application –e.g. the profiler code that calculates how long it takes to execute a method is repeated in many methods.

```
public bool payment(CreditCard card, double ammount) {
  double startTime = DateTime.Now.Ticks;
  ILog logger = LogManager.GetLogger(
           MethodBase.GetCurrentMethod().DeclaringType);
  logger.Info("Entering the payment method");
  logger.Debug("Arguments:  {card="+card+",
           ammount="+ammount+"}");
  bool correct = validateCard(card.Number, card.ExpDate,
           card.CardType);
  if (correct) {
    CardCompany company=CardCompany.getCardCompany(card.CardType);
    correct = company.transfer(card, this.myAccount, ammount);
  }
  logger.Debug("The payment has been "+
           (correct?"successful": "erroneous"));
  logger.Info("Exiting the payment method");
  profiler.measure(MethodBase.GetCurrentMethod().Name,
            (DateTime.Now.Ticks-startTime) /
             TimeSpan.TicksPerMillisecond );
  return correct;
  }
```

Fig. 1. Source code with three different concerns tangled.

AOSD [1] provides explicit language support for modularizing application concerns that cut across the application code. AOSD is an approach to obtain a better separation of concerns than object orientation. Aspects express functionality that

cuts across the system in a modular way, thereby allowing the developer to design a system out of orthogonal concerns, providing a single focus point for modifications. The modularization of crosscutting concerns avoids application source tangling, being easier to debug, maintain and modify [3].

In AOSD, final applications are built by means of its components plus their specific crosscutting concerns. This task is performed by the aspect weaver. Figure 2 shows the difference between the traditional modularization scheme of the object-oriented paradigm and the AOSD one. With the aspect-oriented approach, each module can represent a separate aspect, overcoming the code tangling and code spreading disadvantages mentioned above. Aspect weavers perform the transition from aspect modularization to the traditional one.

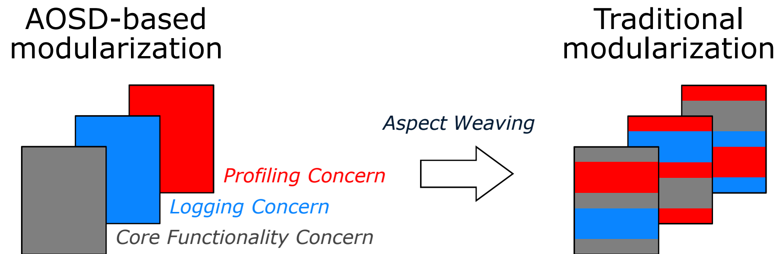

Fig. 2. AOSD-based and traditional modularization of concerns.

Considering when aspects are woven, AOSD platforms can be classified into static and dynamic. Static tools perform aspect weaving prior to application execution (at compile-time or load-time), whereas dynamic tools provide application aspectation at runtime.

3. Dynamic AOSD

3.1. *Existing Dynamic AOSD Systems*

Dynamic AOSD tools perform aspect weaving at runtime, achieving dynamic adaptation of application aspects. Many AOSD tools do not support aspect adaptation at runtime, because they use static (or load-time) weavers. Once the final application has been generated (woven), the system will not be capable of adapting its aspects at runtime. There are certain cases in which the adaptation of application concerns should be done dynamically, in response to changes in the runtime environment –e.g. some examples were given in Section 1 .

Unlike static weaving systems, dynamic ones offer some kind of runtime adaptation because aspects can be (un)woven while applications are running. Well-known examples are PROSE [8, 17], JBoss AOP [18] and JAsCo [19]. However, dynamic AOSD systems commonly have some limitations:

(1) **Limited runtime adaptiveness**. Many dynamic AOSD systems do not offer a high level of adaptiveness at runtime, requiring the static specification of which join-points the aspects are going to use at runtime (e.g., Rapier-LOOM.Net [20] and Spring AOP [21]). Others offer dynamic addition of new aspects but they do not support dynamic deletion (e.g., Wicca [22]).
(2) **Limited set of join-points**. The set of join-points that most dynamic weavers offer is significantly smaller than that offered by static ones [13, 23]. As an example, JAsCo is a dynamic weaving platform that offers an excellent runtime performance, but it only allows interception of the `methodcall` join-point.
(3) **Limited portability and interoperability**. Some implementations of dynamic weavers obtain an extraordinary runtime performance (quite near to static ones) by modifying the implementation of a virtual machine. An example is the Steamloom approach that modifies the Jikes research virtual machine to detect join-point shadows and perform runtime advice weaving [24]. However, the modification of a specific virtual machine involves portability and interoperability limitations [25]. This means that it is not possible to use, for example, any standard implementation of the Java virtual machine.
(4) **Platform dependency**. There are dynamic weavers that can only be used in a specific operating system or hardware. The Arachne dynamic weaver for C applications is an example of this kind of systems. Arachne rewrites binary code of executable files at runtime as long as these files conform to the mapping defined by the Unix standard between C and the x86 assembly language [26]. Although the runtime performance obtained is extraordinary high, this dynamic weaving technique makes Arachne dependent on a specific platform [14].

Our platform overcomes these limitations (see Section 7). It implements a full dynamic weaver that allows runtime addition and deletion of aspects, even at join-points that were not woven before application execution. Both the static and the dynamic weavers offer the same number of join-points, quite similar to AspectJ. Additionally, DSAW is platform and language independent.

3.2. *Static weaving where possible, dynamic weaving when needed*

Although existing implementation of static weavers commonly offer runtime performance benefits, they also have some limitations. As an example, if the programmer develops a statically woven application that requires logging or testing aspects, the application must be compiled, woven, run and debugged. The runtime context to debug should be reproduced and, afterwards, all the information generated by the aspects should be analyzed. If some error occurs, the application should be modified, recompiled, rewoven and rerun [9, 27]. In contrast, dynamic weaving allows the programmer to add and remove aspects in exact points of execution, producing less information to be analyzed. Moreover, application execution should not be stopped in case we want to modify an aspect –it can be modified, recompiled and once again rewoven at runtime. This has been referred to as *edit-and-continue* development

[12].

Different scenarios motivate the use of static weaving where possible and dynamic weaving when needed [14, 9, 28]. A tool that supports both techniques should define aspects independently of their dynamism (weaving-time). This approach will benefit from both static and dynamic weaving in the same system.

DSAW is a homogeneous static and dynamic weaving aspect-oriented platform. Its main aim is to achieve weaving-time neutrality, and language and platform independence. Both kinds of weavers are offered and source code of neither aspect nor components depends on their weaving time. The application should not be changed if the programmer needs to convert an aspect from static to dynamic, and vice versa. This way, it is possible to use aspect-orientation for rapid prototyping (dynamic weaving) and later, upon deployment, optimize the application (static weaving) without performing any change to its source code. The programmer should finally indicate the trade-off between performance and flexibility in the system requirements [14].

4. Formal Semantics of DSAW

There are different approaches to describe the semantics of aspect oriented languages [29, 30, 31]. However, most of them provide a semantics as a whole but do not isolate the specific features of aspect languages. The formalization of the semantics of DSAW is based on the *Common Aspect Semantics Base* (CASB) semantics proposed by Djoko *et al.* [15]. CASB describes aspects semantics independently from the base language, introducing the minimal constructions of the base language necessary to plug aspects in. We present a variant of CASB in order to describe our both static and dynamic weaving system.

4.1. *The Base Language Semantics*

The base language semantics is described in terms of a small-step semantics, formalized through the $\rightarrow_b$ reduction on configurations made of a program (C) and a state (Σ). A program C is a sequence of basic instructions terminated by the empty instruction ε, where : denotes the concatenation of two instructions:

$$C ::= i : C \mid \varepsilon$$

A configuration is a tuple (C, Σ) where Σ represents the state of the interpreter. Σ is kept as abstract as possible. It may contain environments, stacks, heaps, or whatever element depending on the semantics of the considered language, and the details of its implementation.

A reduction step of the base language semantics is written:

$$(i : C, \Sigma) \rightarrow_b (C', \Sigma')$$

Intuitively, i represents the current instruction and C the following ones. The final configuration has the form (ε, Σ).

4.2. *Static Aspect Weaving*

Aspects semantics is represented with a function ψ that, applied to the current instruction i, returns a tuple of triples with all the aspects woven at the i instruction. Each triple contains an advice ϕ, a type t (*before*, *after* or *around*) denoting the kind of aspect, and w indicating when the aspect has been woven (*static* or *dynamic*). An *advice* is a method-like construct used to define additional behavior at join-points [32]. *Join-points* are those elements of the programming language semantics which the aspects coordinate with [1].

In CASB, ϕ is a function that takes Σ and returns an advice. For the sake of simplicity, we will rather assume that aspects are directly returned as executable instructions (i.e., an advice is a sequence of instructions).

$$\psi(i) = ((\phi_1, t_1, w_1) \ldots (\phi_n, t_n, w_n))$$
$$\text{where } t \in \{before, after, around\} \text{ and } w \in \{static, dynamic\}$$

The ψ function can be seen as a way to decide which join-points are woven. Static ones are woven before the application is executed, and they do not change while the program is running; dynamic ones can be added and removed at runtime (Section 4.3). The ψ function returns ε when no aspect has been woven at the instruction passed as an argument. Note that this formalization allows weaving any instruction in the base language, not only function (or method) calls.

The semantics of weaving is described in terms of the $\rightarrow$ reduction on configurations, which includes the semantics of the base language ($\rightarrow_b$). The NoAdvice rule executes the current i instruction when it has no aspect woven.

$$\frac{\psi(i) = \varepsilon \qquad (i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(i : C, \Sigma) \rightarrow (C', \Sigma')} \quad (\text{NoAdvice})$$

4.2.1. *Before, After and Around Aspects*

To describe the semantics of *before*, *after* and *around* aspects we first depict the behavior when one single aspect is woven at an instruction. Afterwards, we generalize this scenario with multiple different aspects intercepting the same join-point.

In order to formalize around aspects, the base language is enhanced with an additional `proceed` instruction which can be used in the code of an around advice. Around aspects execute their advice instead of the current instruction. The advice code may execute the original instruction by using the `proceed` instruction –the advice may also terminate without executing the current instruction if no `proceed` instruction is run. In general, an around advice may contain several proceeds resulting in multiple executions of the instruction matched by the around aspect.

To represent the behavior of around aspects a special stack P, called the *proceed stack*, is introduced. This stack is used to describe the semantics of the `proceed` instruction. The Around rule inserts the advice code followed by a pop_p instruction, and pushes the current instruction i in the proceed stack so that it can be possibly

executed by a `proceed`. The pop_p instruction simply removes the top of the proceed stack to restore the stack to its original state.

$$\text{(AROUND)}\ \frac{\psi(i) = (\phi, around, w)}{(i : C, \Sigma, P) \rightarrow (\phi : pop_p : C, \Sigma, \bar{i}{:}P)} \qquad \text{(POP)}\ \frac{}{(pop_p : C, \Sigma, i : P) \rightarrow (C, \Sigma, P)}$$

In order to prevent an instruction i to be matched, it is introduced the notion of tagged instructions (written $\bar{i}$). A tagged instruction $\bar{i}$ has exactly the same semantics as i except that it is not subject to weaving. Formally:

$$\forall\ (i, C, \Sigma),\ (i : C, \Sigma) \rightarrow_b (C', \Sigma') \Rightarrow (\bar{i} : C, \Sigma) \rightarrow (C', \Sigma')$$
$$\forall\ i,\ \psi(\bar{i}) = \varepsilon$$

The rule PROCEED executes the instruction on top of the proceed stack. This instruction is removed because i may be the code of an enclosing around aspect whose `proceed` would refer to the top of the stack P (not i). After proceeding, i is pushed again (PUSH), since the advice may contain other proceeds.

$$\text{(PROCEED)}\ \frac{}{(\texttt{proceed} : C, \Sigma, i{:}P) \rightarrow (i : push_p i : C, \Sigma, P)} \qquad \text{(PUSH)}\ \frac{}{(push_p i : C, \Sigma, P) \rightarrow (C, \Sigma, i{:}P)}$$

Note that aspects can be woven at any instruction, not only at function (method) calls. We also assumed that the advice of an around aspect could be matched by another around aspect and that there can be nested `proceeds` –AspectJ prevents this case by syntactic restrictions.

After defining the semantics of *around*, we can describe the semantics of *before* and *after* by means of a function γ, translating any aspect tuple into an equivalent tuple of around aspects:

$$\begin{array}{lcl} \gamma(\phi, before, w) & = & (\phi : \texttt{proceed}, around, w) \\ \gamma(\phi, after, w) & = & (\texttt{proceed} : \phi, around, w) \\ \gamma(\phi, around, w) & = & (\phi, around, w) \end{array}$$

A before aspect is translated into an around one that inserts the advice before it proceeds with the next aspect. Symmetrically, an after aspect is translated into an around one that places the advice after the next aspect is executed.

4.2.2. *Multiple Aspects in the same Join-Point*

After formalizing the execution of aspects when only one can be woven at a join-point, we now consider the weaving of several aspects at the same join-point. Although several aspects of different kinds (*after*, *before* and *around*) can be woven at the same join-point, we showed how a function γ can be used to translate all of them into around aspects. Therefore, we simply tackle with potentially multiple around aspects woven at each join-point.

The way aspects woven at the same join-point are sorted is not a trivial task. For instance, AspectJ by default sorts these triples in the order *before*, *after* and

around, although the programmer can modify it by the use of `declare precedence`. To generalize this, the use of a new α function has been previously considered [33].

$$\alpha(\Sigma, (\phi_1, t_1, w_1), \ldots, (\phi_n, t_n, w_n)) = ((\phi'_1, t'_1, w'_1) \ldots (\phi'_n, t'_n, w'_n))$$

This function sorts the set of triples in a join-point, introducing the possibility to perform dynamic scheduling based on the dynamic context (Σ is passed as a parameter), advice code (ϕ), the type of the aspects (t), and their weaving time (w). Section 6.8 describes how we have defined the α function in our current implementation.

We can now specify the Around* rule that performs weaving of all the aspects in a single join-point.

(Around*)

$$\frac{\begin{array}{c}\psi(i) = ((\phi_1, t_1, w_1) \ldots (\phi_n, t_n, w_n)) \\ \alpha(\Sigma, (\phi_1, t_1, w_1), \ldots, (\phi_n, t_n, w_n)) = ((\phi'_1, t'_1, w'_1) \ldots (\phi'_n, t'_n, w'_n)) \\ \gamma(\phi'_1, t'_1, w'_1) = (\phi''_1, around, w'_1) \ldots \gamma(\phi'_n, t'_n, w'_n) = (\phi''_n, around, w'_n)\end{array}}{(i : C, \Sigma, P) \to (\phi''_1 : \ldots : \phi''_n : pop_p : C, \Sigma, \bar{\bar{i}}{:}P)}$$

The semantics of multiple weaving in a single join-point differs from AspectJ in how the `proceed` instruction works [15]. In AspectJ, the first advice is executed and the rest of advice are pushed in the proceed stack. This means that, if the code of the first advice proceeds, the second one will be executed and so on. In our approach, every advice is executed in place of the i instruction. When one of them proceeds, the i instruction is then run. It is worth noting that the AspectJ semantics can be obtained in our system by weaving around aspects to another (*before*, *after* or *around*) aspect, establishing a chain of woven aspects.

4.3. *Dynamic Aspect Weaving*

Since dynamic weaving may modify the set of aspects woven at a join-point while a program is being executed, we enhance the configuration tuple with the ψ function. The $\to$ reduction will now describe how to add/remove aspects to/from a specific join-point at runtime.

Dynamic weaving is offered with two new instructions: `weave` and `unweave`. The former adds a new dynamic aspect to the i instruction, specifying the advice (ϕ) and the type of the aspect (t). $\psi[i \mapsto (\phi, t, dynamic)]$ denotes the function ψ updated for the instruction i to return the aspect $(\phi, t, dynamic)$, meaning that the new aspect $(\phi, t, dynamic)$ will for now on be woven at the i instruction. Therefore, the next time the Around* semantic rule is executed, the new dynamically woven aspects will be taken into account.

$$\frac{\begin{array}{c}\psi(i) = ((\phi_1, t_1, w_1) \ldots (\phi_n, t_n, w_n)) \\ \psi' = \psi[i \mapsto ((\phi_1, t_1, w_1) \ldots (\phi_n, t_n, w_n)(\phi, t, dynamic))]\end{array}}{(\texttt{weave}\ i, \phi, t : C, \Sigma, P, \psi) \to (C, \Sigma, P, \psi')} \quad \text{(Weave)}$$

The `unweave` instruction does the opposite, removing a previously woven aspect at an specific instruction[a]. Notice that aspects woven and unwoven at runtime are always dynamic.

(UNWEAVE)

$$\frac{\psi(i) = ((\phi_1, t_1, w_1)\ldots(\phi_j, t_j, dynamic)^{j\in[1,n]}\ldots(\phi_n, t_n, w_n)) \quad \psi' = \psi[i\mapsto((\phi_1, t_1, w_1)\ldots(\phi_{j-1}, t_{j-1}, w_{j-1})(\phi_{j+1}, t_{j+1}, w_{j+1})^{j\in[1,n]}\ldots(\phi_n, t_n, w_n))]}{(\texttt{unweave}\ i, \phi_j, t_j^{j\in[1,n]} : C, \Sigma, P, \psi) \rightarrow (C, \Sigma, P, \psi')}$$

5. System Architecture

The architecture of DSAW is depicted in Figure 3. Although each module is detailed in Section 6, this section briefly describes the responsibilities of each subsystem.

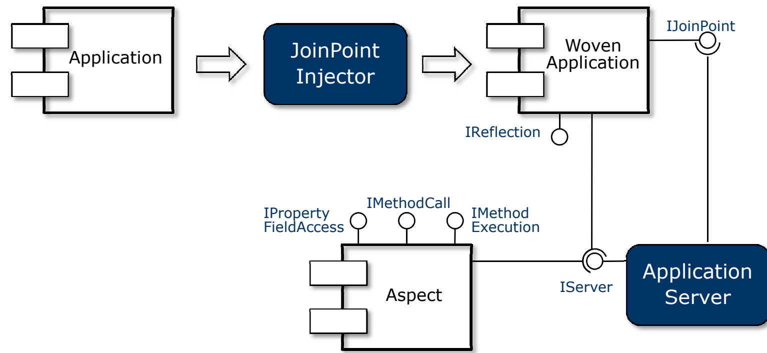

Fig. 3. Architecture of DSAW.

Any existing .NET application, regardless of the programming language used to develop it, can be adapted by DSAW. The Join-Point Injector (JPI) takes the application binary code (*assembly*) and, prior to its execution, performs instrumentation of the code in memory. If the weaving is static, a pointcut specification file must be passed as a parameter (a *pointcut* is a set of join-points plus, optionally, some of the values in the execution context of those join-points [32]). In that case, the application is modified with calls to specific interfaces (`IPropertyFieldAccess`, `IMethodCall` and `IMethodExecution`) of the appropriate aspect specified in the pointcut specification file. This functionality is the ψ function described in Section 4.2 when no `weave` or `unweave` instructions have been executed, indicating which join-points have been statically woven.

[a]We consider the collection of aspects woven at the same join-point as a set. This simplification limits the possibility to weave the same advice code, with the same aspect kind, to the same join-point more than once. To avoid this limitation, a unique identifier would have to be added to each aspect.

In case dynamic weaving is required, the JPI also instruments the application with more code to perform dynamic weaving. Since the JPI does not know in which join-points the developer of a dynamic aspect could be interested in, it instruments the application so that any join-point can be intercepted at runtime –notice that, in our formalization, *any* join-point could be intercepted. The activation of these join-points is offered by the `IJoinPoint` interface implementation added to the application by the JPI (Figure 3). It is also injected an implementation of the `IReflection` interface that allows aspects to reflective access applications at runtime. This is particularly powerful in the case of around aspects.

The Application Server (AS) in Figure 3 coordinates applications and dynamic aspects, providing the aspect-oriented adaptation of programs at runtime. This subsystem allows the execution of the dynamic `weave` and `unweave` operations described in Section 4.3. By means of the `IServer` interface, it allows aspects to be woven with an specific application at certain join-points (`IJoinPoint`). The `IServer` interface is also used to register and deregister aspects and applications at runtime. The JPI instruments applications making them automatically register and deregister at application startup and finalization, respectively.

Aspects implement code that may be woven with applications by implementing three interfaces: `IPropertyFieldAccess`, `IMethodCall` and `IMethodExecution`. These interfaces should be implemented depending on the type of the join-points to be adapted. The advice code represented with the ϕ function in Section 4.2 is the aspect code implementing these interfaces. The type of advice (*before*, *after* and *around*) is specified in the pointcut description files (Section 6.4) that aspects pass to the AS by means of the `IServer` interface.

6. System Design

6.1. *Applications*

DSAW has been developed over the .NET platform following its standard reference [34] neither modifying nor extending its semantics. This guarantees a complete language and platform independence, allowing the deployment of our system over any .NET implementation (such as Mono, SSCLI and DotGNU) [35]. It is not only possible to use any .NET language to develop aspect-oriented applications, but it is also feasible to create each aspect in a different programming language (Figure 4).

DSAW performs software adaptation at the byte-code level of the virtual machine –libraries and executable files. This means that our weaver does not require the source code of aspects or components, and it is language independent. At the same time, it is not necessary to implement specific interfaces or inherit from any given classes. Any existing application or library can be used in DSAW without changing its implementation –we follow the *POJO* idea of the *Java Persistence API* [36]. C# code in Figure 5 shows the core component of our credit card payment example. Using DSAW, no other concern needs to be included –they are now implemented as aspects.

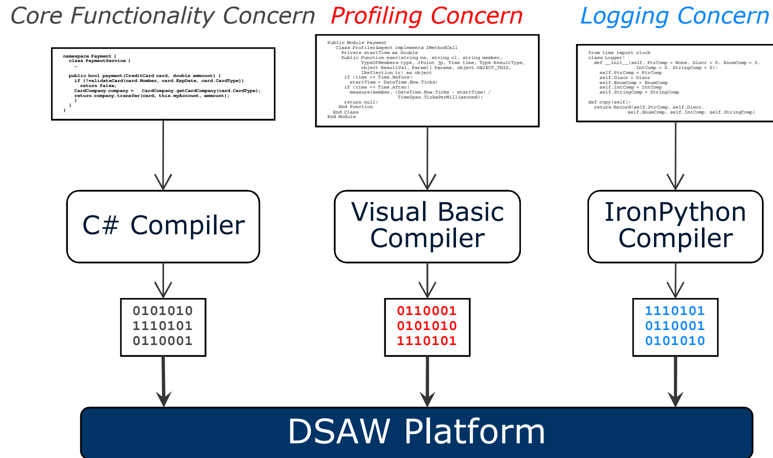

Fig. 4. DSAW language neutrality.

```
namespace Payment {
  class PaymentService {
    public bool payment(CreditCard card, double ammount) {
      if (!validateCard(card.Number, card.ExpDate, card.CardType))
        return false;
      CardCompany company = CardCompany.getCardCompany(card.CardType);
      return company.transfer(card, this.myAccount, ammount);
    }
  }
}
```

Fig. 5. C# implementation of the core functionality in the DSAW platform.

6.2. *Join-Point Injector*

The Join-Point Injector (JPI) is the part of the DSAW platform that performs byte-code instrumentation to incorporate any existing .NET binary application into our aspect-oriented system. .NET byte-code is the source language of the .NET virtual machine, being language and platform independent [34].

Prior to its execution, the application to be adapted is processed in memory by the JPI, adding the code that allows its dynamic adaptation by aspects at runtime (Figure 6). The JPI also performs static weaving; this feature is described in Section 6.6.

One limitation of existing runtime weavers, compared to static ones, is commonly a more reduced set of join-points. This is mainly due to the complexity of implementing runtime adaptation [13]. One of the features that have been considered in the design of DSAW is the set of join-points to be provided. The collection of join-

points DSAW offers is near to the one supplied by AspectJ [32] (see Section 7.2). We currently support the following static and dynamic join-points: method and constructor execution, method and constructor call, and field and property read and write. We also provide the before, after, and around advice.

In order to implement join-points, the first functionality the JPI adds to the program binaries is a Meta-Object Protocol (MOP) [37]. A MOP is a reflective technique that offers dynamic adaptation of running applications [38]. The injected MOP provides runtime modification of the program semantics such as message passing or field access. This way, it is possible to adapt running applications with dynamically woven aspects.

The JPI analyzes byte-code to detect join-point shadows. A join-point shadow is the mapping between join-points and the points in the program code where the compiler actually operates [39]. When a join-point shadow is detected, new byte-code is added to implement the MOP. This new code checks at runtime if any aspect has been subscribed to that join-point; if so, subscribed aspects will be called when the join-point is reached. An implementation of the `IJoinPoint` interface is also added to allow aspects to register for join-point activation (Figure 6).

The JPI also injects into the application other functionalities of the platform such as application registration at startup, accessing the `IServer` interface of the AS subsystem. It is also included a deregistration routine at application exit, and the publication of .NET reflective information to permit aspects to inspect (and invoke) application structure at runtime (the `IReflection` interface in Figure 6). The application is now ready to be executed.

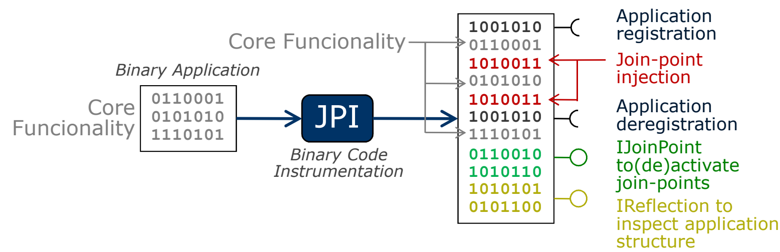

Fig. 6. Byte-code instrumentation performed by the JPI.

6.3. *Aspects*

Aspects (both dynamic and static) can be developed in any .NET programming language. DSAW does not need the source code of aspects, and hence it could be possible to take third-party binary software and make them work as if they were aspects.

An aspect can be developed following two approaches. The first one, which

provides better runtime performance, requires the programmer to implement at least one of the three following interfaces (all of them have one single `exec` method):

(1) `IMethodCall`. It is used for intercepting message passing. This interface can be used to run at the `after`, `before` and `around` *times*. Its parameters are (regarding to the join-point): the name of the namespace, the name of the class, the name of the member (method or constructor), the type of join-point (`call` or `execution`), the join-point time (`after`, `before` or `around`), the result type, the result value, the types and values of parameters, the `this` reference in the application, and a reference to `IReflection` –see Section 6.2. It can be applied to both methods and constructors.
(2) `IMethodExecution`. This interface intercepts the execution of methods and constructors. The `before`, `after` and `around` *times* are provided, and its parameters are the same as above.
(3) `IPropertyFieldAccess`. Interception of fields and properties accesses can be done implementing this interface. The parameters of its unique `exec` method are the names of the namespace, class and member, the type of member (`field` or `property`), the type of join-point (`reference` for reading and `set` for writing), the join-point time (`after`, `before` or `around`), the member value, and the `this` reference in the application.

Following our example, we can suppose that the programmer is now interested in adapting the application at runtime. Let us assume that, in a certain point of execution, the credit card payment application performance seems to be poor. Under these circumstances, a profiling aspect can be added at runtime, and removed later when the application performance is recovered. The C# source code of this dynamic aspect is shown in Figure 7. It is worth noting how the profiling concern has been clearly separated from the core functionality shown in Figure 5.

The other way of creating an aspect is using an existing application or library on .NET, even if the source code is not available. It could be a third-party component or a static aspect that the programmer is interested in weaving it dynamically. This approach makes it possible to convert an aspect from static to dynamic and vice versa, without changing its implementation. This is how our platform offers a transparent separation of the dynamism concern.

With this approach, DSAW takes the aspect binaries and an XML document describing the aspect advice, and creates the appropriate implementation of at least one of the interfaces shown above. In order to do that, an XML document describing aspect advice [32] should be written (the structure of this XML document is described in sections 6.4 and 6.7). These two elements (aspect code plus advice type) comprise the two first elements of the formalization presented in Section 4: (ϕ, t, w) –the w element, *weaving time*, in this case is *dynamic*.

```
namespace Payment {
   public class ProfilerAspect : IMethodCall {
     private double startTime=0;
     public object exec(string ns, string cl, string member,
            TypeOfMembers type, JPoint jp, Time time,
            Type ResultType, object ResultVal, Param[] Params,
            object OBJECT_THIS, IReflection ir) {

      if (time == Time.Before)
        startTime = DateTime.Now.Ticks;
      if (time == Time.After)
        measure(member,(DateTime.Now.Ticks-startTime)
                     /TimeSpan.TicksPerMillisecond);
      return null;
    }
  }
}
```

Fig. 7. C# implementation of the dynamic profiling aspect in the DSAW platform.

6.4. *Pointcut Specification*

We have seen how any existing application or library can be used as components or aspects in DSAW. However, it is necessary to describe the mapping between join-points and aspects by means of pointcuts. In DSAW, pointcuts are specified by means of XML documents that describe the mapping between join-points and aspects. The schema of these XML documents is an evolution of the one used by the Weave.Net platform [40]. As described in Section 7.2, we have developed a Visual Studio plug-in that automatically generates these XML documents, making aspect-oriented programming in DSAW easier.

Describing mappings between components and aspects in separate XML files provides a complete separation (no coupling) between them, improving the reutilization of both aspects and components. In fact, aspects can also be treated as components. They may be adapted by other aspects, statically or dynamically, regardless of their programming language.

Figure 8 shows the pointcut description file of our dynamic profiler. We use both `before` and `after` times of the `methodcall` join-point. We are interested in any return type (regular expressions can be used) and only those methods that are `public` (all the member flags used in the CLI [34] are supported). Our example adapts all the methods (`qualified_method_name`) in every class (`class` and `identifier_name`) in the `Payment` namespace (`namespace` and `type_name`), but the `Main` method (`name`, `not`, `identifier_pattern`, and `identifier_name`).

```
<?xml version="1.0" encoding="UTF-8"?>
<aspect_definitions xmlns="urn:gramaticapointcuts-schema" … >
<pointcut_definition>
  <time>before</time>  <time>after</time>
  <joinpoint_type>     <methodcall>
    <method_signature>
      <return_type><type_name>*</type_name></return_type>
      <method_flags><public/></method_flags>
      <qualified_method_name>
        <qualified_class>
          <namespace><type_name>Payment</type_name></namespace>
          <class><identifier_name>*</identifier_name></class>
        </qualified_class>
        <name> <not><identifier_pattern><identifier_name>Main
        </identifier_name></identifier_pattern></not></name>
      </qualified_method_name>
    </method_signature> </methodcall>
  </joinpoint_type>
</pointcut_definition>
</aspect_definitions>
```

Fig. 8. XML pointcut description for the dynamic profiling aspect.

6.5. *System Execution*

When dynamic weaving is required, system execution is controlled by the AS (Application Server). The AS coordinates components and dynamic aspects, providing the aspect-oriented adaptation of programs at runtime. The AS acts as the system registry of running applications. It offers aspects the list of active applications to facilitate their dynamic adaptation.

Following with the credit card payment example, this is how the application, the dynamic aspect profiler, and the AS work together at runtime (illustrated in Figure 9):

(1) The application, once processed by the JPI, is executed. At startup it registers itself into the AS with a globally unique identifier (GUID), following the OSF/DCE specification [41]. This GUID (and registration code) was previously injected by the JPI during code instrumentation, and it is used to identify the application in the system.
(2) When the developer detects low runtime performance, the profiling aspect is run to dynamically adapt the application. The aspect calls the AS (`IServer`) passing the pointcut XML document shown in Figure 8 and the GUID of the application.
(3) The AS parses the XML document finding the pointcuts passed by the aspect.

The join-points that match these pointcuts are activated in the application by means of the MOP injected by the JPI (`IJoinPoint`). This activation implies aspect invocation at runtime. This action is the `weave` i, ϕ, t instruction formalized in Section 4.3, where i and t are, respectively, the join-point and advice type described in the XML document, and ϕ is the aspect code.

(4) When the credit card application execution reaches a woven join-point (e.g., calling the `payment` method), it calls the profiling aspect through `IMethodCall`. Then, the application sends the aspect information regarding the join-point and a reference to the own application. The profiling aspect now saves the execution time to measure runtime performance.

(5) Although it is not used in this example, the aspect may use the application reference to access the application by means of reflection. The JPI adds the `IReflection` interface for this purpose. Therefore, the aspect could inspect and modify the values of any field or property of the application, and invoke any of its methods (not only the intercepted one).

(6) When the aspect execution is about to finish, the AS deactivates the application join-points previously turned on by the aspect (if no other aspect uses those join-points). This action is what we formalized as the `unweave` i, ϕ, t instruction in Section 4.3.

(7) Finally, when the application finishes its execution, the code added by the JPI notifies the AS that the application has exited.

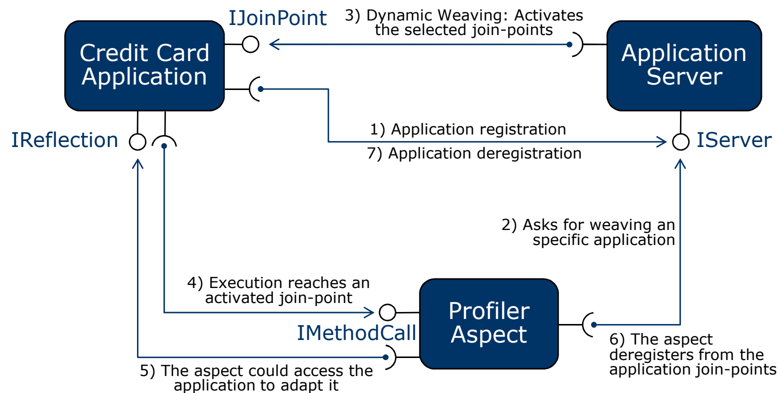

Fig. 9. Dynamic application adaptation at runtime.

It could be necessary to modify the pointcuts used by an aspect at runtime. This operation is also offered by the AS. In this case, the aspect should send a new XML document specifying the new pointcut document. The AS will then analyze this XML file, activating new join-points and deactivating existing ones in the running application.

The AS acts as a *mediator* between aspects and applications [42]. This mediation is only performed when join-points are woven or unwoven. Once these operations have been performed, the application and the aspects interact directly. The application calls the aspect when an activated join-point is reached, and then the aspect may inspect the application.

With this design, applications do not need to know their runtime-woven aspects before its execution. At the same time, aspects can be applied to any application, or even aspects, without any static dependency. This behavior reduces coupling and promotes both aspect and component reuse.

We have used .NET remoting (now part of the *Windows Communication Foundation* framework) to intercommunicate the AS, aspects and applications. .NET remoting is a standard service over the .NET platform and is channel (protocol) independent, allowing DSAW to run over distributed environments.

6.6. *The JPI as a Static Weaver*

Although dynamic weaving facilitates interactive software development following the AOSD approach, it also involves a common runtime performance cost. At the same time, although there are scenarios where the dynamic adaptation of aspects is appropriate, many others do not require that level of dynamism. This is the reason why we have designed DSAW to support both approaches at the same time, obtaining the benefits of both.

We have applied the *Separation of Concerns* principle to DSAW. In particular, we have separated the *dynamism* (weaving-time) concern to facilitate the use of AOSD in multiple scenarios. This process has been performed transparently, reducing the impact of changing the dynamism concern in the application source code. The programmer could use dynamic weaving for interactive development and, once the application has been tested, use static weaving to obtain better performance. An existing dynamic aspect can also be easily converted to a dynamic one. Moreover, both kinds of aspects, dynamic and static, can be simultaneously applied to the same application.

We have seen how the JPI instruments the byte-code to obtain dynamic adaptation of applications. In addition, if an XML document describing pointcuts is passed as an argument to the JPI, it performs static weaving between components and aspects. Therefore, as shown in Figure 10, the JPI not only instruments applications to be adapted at runtime, but it also weaves them statically.

Following with our example, we can reuse an existing logging aspect taken from another aspect-oriented program. Maintaining the dynamic profiling aspect, we can add a new static one to perform logging tasks. The source code in Figure 11 shows the last concern of our example. Notice that the method signature is exactly the same as the `exec` method of the dynamic aspect (Figure 7).

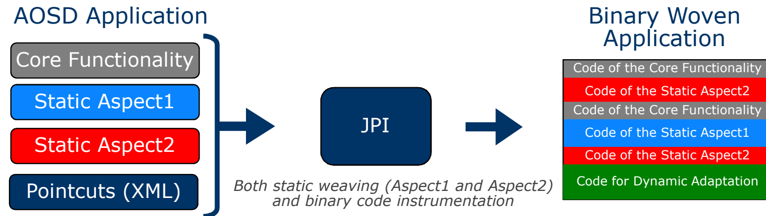

Fig. 10. Static weaving performed by the JPI.

```
namespace Payment {
   public class LoggerAspect {
     static public object exec(string ns, string cl,
           string member, TypeOfMembers type, JPoint jp,
           Time time, Type ResultType, object ResultVal,
           Param[] Params, object OBJECT_THIS,
           IReflection ir) {
      ILog logger = LogManager.GetLogger(
                MethodBase.GetCurrentMethod().DeclaringType);
      if (time == Time.Before) {
        logger.Info("Entering the " + member + " method");
        StringBuilder sb = new StringBuilder("Arguments: {");
        for (int i = 0; i < Params.Length; i++) {
          sb.Append(Params[i].name + "=" + Params[i].val);
          sb.Append(i < Params.Length - 1 ? ", " : "}");
        }
        logger.Debug(sb.ToString());
      }
      if (time == Time.After) {
        logger.Debug("The payment has been " +
                 ((bool)ResultVal?"successful":"erroneous"));
        logger.Info("Exiting the " + member + " method");
        }
      return ResultVal;
    }
  }
}
```

Fig. 11. C# implementation of the static logging aspect in the DSAW platform.

6.7. *Pointcut and Advice Description for Static Weaving*

As mentioned, the JPI performs static weaving when an XML document is passed as an argument from the command line. This document describes pointcuts, but also requires advice annotation. In DSAW, an advice indicates which methods in an aspect must be called when join-points (described by pointcuts in the XML file) are reached. This is what we formalized as the ψ function in Section 4.2.

Figure 12 shows the XML document used to statically weave the logging aspect of our example with the rest of the application. Pointcuts are defined first, the same way as described in Section 6.4, and then comes the advice specification (`advice_definition`). After the name of the aspect (`StaticLoggerAspect`) its assembly is identified (`static.logger.dll`); then, we indicate the class name including its namespace (`Payment.LoggerAspect`), the method to be called at join-point interception (`exec`), an optional priority (it is explained in the following section), and a reference to its corresponding pointcut description. In this example, whenever a public method of any class in the `Payment` namespace is reached, the `exec` method of the `Payment.LoggerAspect` class will be called.

```
<?xml version="1.0" encoding="UTF-8"?>
<aspect_definitions xmlns="urn:gramaticapointcuts-schema" … >
  <pointcut_definition id="DynamicMethodCall">
    <time>before</time>  <time>after</time>
    <joinpoint_type> <methodcall>
      <method_signature>
        <return_type><type_name>*</type_name></return_type>
        <method_flags><public/></method_flags>
        <qualified_method_name>
          <qualified_class>
            <namespace><type_name>Payment</type_name></namespace>
            <class><identifier_name>*</identifier_name></class>
          </qualified_class>
          <name><identifier_name>*</identifier_name></name>
        </qualified_method_name>
      </method_signature>
    </methodcall></joinpoint_type>
  </pointcut_definition>
  <advice_definition>
         <name>StaticLoggerAspect</name>
         <assembly>static.logger.dll</assembly>
         <type>Payment.LoggerAspect</type>
         <behaviour>exec</behaviour>
         <priority>1</priority>
         <pointcut_definitionRef idRef="DynamicMethodCall"/>
  </advice_definition>
</aspect_definitions>
```

Fig. 12. XML pointcut and advice description for the static logging aspect.

As mentioned in Section 6.4, dynamic aspects could also use advice XML documents. In that case, it is not necessary to implement the `IMethodCall`, `IMethodExecution` or `IPropertyFieldAccess` interfaces; DSAW creates these implementations taking into account advice descriptions. This makes possible the transition from dynamic to static, and the other way around, without modifying the implementation of aspects.

6.8. *Conflict Resolution*

In DSAW, it is possible to create applications with statically woven aspects woven together with aspects that are later added at runtime. Therefore, it is necessary to define a conflict resolution mechanism [43]. A conflict between aspects is produced when two different aspects are woven at the same join-point. A conflict resolution algorithm should make it possible to specify a flexible strategy to determine which aspect should be called first when two or more aspects are woven at the same join-point.

In Section 4.2.2 we formalized the conflict resolution strategy as a function α that could be implemented in different ways, taking the dynamic environment information into consideration. Our current implementation of the α conflict resolution function, although straightforward, takes into account four variables: the aspect dynamism, its advice type, its priority, and when it was woven.

(1) Taking into account its dynamism, DSAW gives priority to dynamic aspects. Since a dynamic aspect is not intercepted by the application throughout its whole execution, DSAW gives priority to these type of aspects.
(2) Considering the advice type, aspect code is executed following the *before*, *around* and *after* order. As mentioned in Section 4.2.2, if the programmer wants around aspects to adapt the rest of aspects woven at the same join-point (following the AspectJ semantics), the around aspect should weave the existing aspects instead of the application.
(3) For each kind of aspect, the programmer may set its priority with a number between 1 and 100. The higher this value is, the sooner the aspect is executed. This mechanism is quite similar to the `declare precedence` construction of AspectJ. However, DSAW establishes precedence between pointcuts, whereas AspectJ uses aspects. This makes DSAW capable of solving more than one conflict at the very same aspect.
(4) Finally, aspects with the same dynamism and priority are executed following a FIFO policy strategy.

Although our current implementation of conflict resolution is simplistic, we have separated the conflict resolution mechanism from aspect implementation to facilitate the addition of new conflict resolution strategies in the future. The first approach would be extending the XML advice description file to allow the programmer to specify a method to be called for dynamic conflict resolution. This approach is similar to JAsCo *connectors* [19], but it would be language neutral in our case. Another approach to improve our conflict resolution mechanism is to include declarative rules that, making use of *composition operators* [44], would allow the user to reorder or nest the aspects involved in a conflict. Following steps could include more advanced solutions such as *stateful aspects* [45] to improve dynamicity of conflict resolution (taking into account the history of computation), or even semantic conflict detection and correction [46].

7. Evaluation

In this section we evaluate different AOSD platforms, comparing them with DSAW. Our experimental methodology is outlined first. Afterwards, qualitative and quantitative evaluations are performed. Finally, we present a discussion regarding to the evaluation obtained, and a description of two real applications developed in DSAW.

7.1. *Methodology*

We measure both qualitative and quantitative features of some well-known AOSD systems. The qualitative features are those mentioned throughout the paper (detailed in Section 7.2). Quantitative characteristics are runtime performance and memory consumption.

Quantitative assessment gives us an estimate of what are the benefits of static weaving versus dynamic weaving. It also can be used to contrast runtime performance and memory consumption between different platforms. We have developed a micro-benchmark over different join-points to assess the cost of aspect-oriented primitives in each platform. Two real applications are also evaluated.

In order to compare DSAW with existing systems, we first analyze those systems that support both dynamic and static weaving. The only one that seemed to be mature enough was LOOM.Net. However, LOOM.Net applies different weavers for static and dynamic scenarios (Gripper and Rapier respectively). Therefore, we have also selected those advanced AOP systems that appear to be used in the development of real applications (AspectJ, Spring for both Java and .NET, and JBoss). Finally, we assessed two systems that offer a high level of dynamism (see Section 7.2): PROSE and JAsCo. No native platform-dependent approach has been evaluated because we consider platform independence a key feature. These are the specific implementations:

- AspectJ 1.6.9 [32]. It is probably the most widely used aspect-oriented tool in software development. It is a seamless aspect-oriented extension to the Java programming language. Although it offers dynamism with pointcuts such as `cflow` or `within`, its weaver is not launched at runtime. We have used the `ajc` compiler in the evaluation.
- Spring Java 3.2.0 [21]. Spring is a layered Java/J2EE application framework that supports Aspect-Oriented Programming (AOP). Spring Java runs over Java 1.4+. It offers an API to add and remove advice at runtime, supporting both load-time and dynamic weaving. Spring Java uses AspectJ to support static weaving.
- Spring .Net 1.3.0 [47]. Spring.NET is an open source application framework to make enterprise .NET applications development easier. The design of Spring.Net is based on the Java version of the Spring Framework. It implements the Spring.AOP to support AOSD. An aspect library provides predefined aspects for transactions, logging, performance monitoring, caching, method retry and

exception handling. Spring.Net 1.1.2 runs on .NET frameworks over 1.2.

- LOOM.Net (Rapier-LOOM.Net 2.2. and Gripper-LOOM.Net 0.92) [48]. The LOOM.Net project aims to investigate and promote the usage of AOP in the context of the Microsoft .NET framework. In order to do that, two AOP tools have been developed, implementing two different weavers. Gripper-LOOM.Net is a static weaver that makes the result of the weaving process permanent. It is language-neutral because it operates on binary .NET assemblies. Rapier-LOOM.Net is the original runtime aspect weaver. Although aspects are woven at runtime, it is not possible to add a new aspect (or replace an existing one) once the application is running.
- PROSE 1.4.0 [17]. PROSE (PROgrammable extenSions of sErvices) allows aspect-oriented Java programs to be modified at runtime. PROSE supports dynamic weaving and unweaving of aspects, even if they are unknown at compile time. We have evaluated the JVMDI/JVMTI event notification based weaver for the JDK 1.5 Windows XP release.
- JAsCo 0.8.7 [19]. A dynamic AOP language originally tailored for the component-based field. The JAsCo technology excels at providing dynamic integration and removal of aspects with a minimal performance overhead. The JAsCo language is an aspect-oriented extension of Java. It requires a Java 1.5 compatible virtual machine.
- JBoss AOP 2.1.8.GA [18]. JBoss AOP is a 100% pure Java aspect-oriented framework usable in any Java programming environment, or tightly integrated with an application server. It offers a prepackaged set of aspects that are applied via annotations, pointcut expressions, or dynamically at runtime. Java 1.5 is required.

These implementations have been compared with the DSAW platform using the .NET Framework 2.0 build 50727 for 32 bits, over a Windows XP SP 3 operating system. All tests have been carried out on a lightly loaded 3.2 GHz iPIV hyper-threading system with 1 GB of RAM. We developed all the tests in Java and C# programming languages.

To evaluate runtime performance, we have instrumented the code with hooks to record the value of the processor time stamp counter. We have measured the difference in the value between the beginning and the end of each benchmark to obtain the total execution time of each program. To suppress the cost of native code generation by the JIT compiler, we first make a single use of each join-point primitive. This first invocation is not taken into account in the evaluation. Therefore, this assessment ignores the time required to dynamically generate native code by the JIT compiler of the virtual machine.

All the benchmarks have been executed utilizing the Windows XP performance monitor. We have measured the maximum size of working set memory used by the process since it started (the `PeakWorkingSet` property). The working set of a process is the number of memory pages currently visible to the process in physical

RAM memory. These pages are resident and available for an application to use without triggering a page fault. The working set includes both shared and private data. The shared data comprises the pages that contain all the instructions that the process executes, including instructions from the process modules and the system libraries.

To evaluate average percentages and ratios, we use the geometric mean.

7.2. *Qualitative Evaluation*

Analyzing those features that were considered in the design of DSAW, we have established a qualitative comparison. The objective of this comparison is not to evaluate which AOSD platform is better, but to clarify what scenarios they have been developed for. More exhaustive AOSD evaluations could be found in [49], [50] and [27]. A global assessment of these features is displayed in Table 1.

Table 1. Features offered by different AOSD platforms.

Req.	**AspectJ**	**DSAW**	**PROSE**	**JAsCo**	**Spring Java**	**Spring Net**	**LOOM .Net**	**JBoss AOP**
1		*Yes*	*Partially*	*Partially*				
2		*Yes*			*Partially*	*Partially*	*Partially*	*Partially*
3		*Yes*						*Yes*
4		*Yes*				*Yes*	*Yes*	
5	*Yes*	*Yes*	*Partially*	*Yes*	*Partially*	*Partially*	*Partially*	*Yes*
6	*Partially*	*Yes*	*Partially*	*Partially*	*Partially*	*Partially*		*Partially*
7		*Yes*	*Partially*		*Yes*	*Yes*	*Yes*	*Yes*
8	18	18/16	4	3	3	3	11	18

(1) **Full dynamic weaving**. This feature is commonly taken into account when dynamic weaving systems are analyzed [19]. Unlike many dynamic AOP approaches, unweaving and reweaving during runtime should be possible, even at join-points that were not woven before. That means that there must be no coupling between aspects and components at all. An aspect must be able to adapt a running application, even if the former was created after running the latter.

Both implementations of the Spring framework require the programmer to specify an XML file declaring advice (*advisors*). At the same time, runtime aspect weaving and unweaving must be explicitly stated in the applications' source code. In the case of JBoss, it is required to specify an XML document with pointcuts and advices. If so, aspects could be used later, once the application has been launched. However, aspects that were not specified in this XML file could not be woven together with the application at runtime.

Although JAsCo, PROSE and DSAW offer a higher level of dynamism than the rest of systems, both JAsCo and PROSE show a limitation if re-weaving

is required. In the *fix-and-continue* debugging scheme it is necessary to weave an aspect at runtime, unweave it later because the runtime behavior is not the expected one, and finally re-weave it with the debugged aspect. Both JAsCo and PROSE permit aspect unweaving, deactivating aspects at runtime. However, if the aspect implementation is replaced by a new one, its new functionality is not reflected at runtime when the aspect is rewoven.

(2) **Both dynamic and static weaving**. Both kinds of weavers are supported in order to obtain better runtime performance and dynamic adaptiveness. If full dynamism is mandatory to fulfill this requirement, only DSAW offers this feature. JBoss, Spring and LOOM.Net partially achieve it.

(3) **Separation of the dynamism concern**. This feature implies the conversion of a static aspect into a dynamic one (and vice versa) without changing its implementation. Both JBoss and DSAW provide this feature. LOOM.Net and the Spring framework use different approaches for both kinds of weaving. Therefore, they do not really separate the dynamism (weaving-time) concern.

(4) **Language neutrality**. This feature implies the development of applications and aspects in any programming language. All the systems but LOOM.Net, Spring.Net and DSAW only support the Java programming language.

(5) **Source code is not required**. Although the LOOM.Net weaves components with aspects at the virtual machine level, it imposes specific requirements on applications: methods need to be `virtual`, `public` methods of classes must be extracted in separate interfaces, and the use of the operator `new` is restricted. This might be interpreted as a source code modification requirement. Something similar happens to PROSE. Although the weaver does not require the source code, the aspect manager should be explicitly included in the applications' source code. Both versions of the Spring AOP framework also require either the explicit load of the advice (*advisor*) document or the inclusion of code that programmatically defines them.

(6) **Integration of existing applications**. This feature represents the idea of taking any existing program or library and use it as a component or aspect. Therefore, it could not be imposed the implementation of specific interfaces or particular naming conventions on component or aspects.

    Concerning to components, only LOOM.Net establishes specific requirements –described in the previous point. Regarding to aspects, all the platforms but DSAW imposes important restrictions over the aspects.

(7) **Separation of pointcuts and aspects**. Both AspectJ and JAsCo include pointcut descriptions in aspects, extending the syntax of the Java programming language. In the case of PROSE, pointcuts are represented by explicit calls to system methods. Therefore, these platforms show some coupling between pointcuts and aspects, making it difficult to reuse aspects.

(8) **Join-point set**. A precise analysis must be done to evaluate the set of join-points offered by each system. This assessment is depicted in Table 2, showing whether or not each system implements the {`constructor`, `method`}{`call`,

`execution`} and `field`{`get`, `set`} join-points. AspectJ offers more join-points but, since they are not provided by the rest of the systems, we have not taken them into account. Join-points have been analyzed with the `before`, `after` and `around` times. In each column, we can see how many join-points each platform implements. This number is also displayed as the eighth row in Table 1. JBoss AOP has just included the `before` and `after` times in its last version.

Table 2. Join-point set offered by different AOSD platforms.

	AspectJ	**DSAW Static**	**DSAW Dynamic**	**PROSE**	**JAsCo**	**Spring Java**	**Spring .Net**	**Rapier**	**Gripper**	**JBoss AOP**
Method Call	*All*	*All*	*All*	*Before, After*	*All*	*All*	*All*	*All*	*All*	*All*
Method Execution	*All*	*All*	*All*							*All*
Constructor Call	*All*	*All*	*Before, After*					*All*	*All*	*All*
Constructor Execution	*All*	*All*	*Before, After*							*All*
FieldGet	*All*	*All*	*All*	*Around*				*Before, After*	*Before, After*	*All*
FieldSet	*All*	*All*	*All*	*Around*				*All*	*All*	*All*
Number of Join-Points	18	18	16	4	3	3	3	11	11	18

Another issue that we are currently working in is the easy-of-use of the AOSD platform. We have recently developed a Visual Studio plug-in (Figure 13) to facilitate the development of aspect-oriented applications in DSAW. The appearance of the IDE is similar to the AspectJ Development Tools (AJDT) for Eclipse. Current features include visual pointcut and advice connection (upper center window), plus XML autocomplete (bottom center window), to facilitate the creation of the pointcut specification documents described in this paper. For each pointcut, the IDE shows the programmer the join-points that the pointcut activates and the woven advice (bottom left window). Selecting an advice, it is shown the adviced pointcuts and join-points. Using reflection, the structure of every component and aspect is shown regardless its programming language (upper left window). Existing projects do not need to be modified to be included in the DSAW platform (the solution explorer window on the right). It is only necessary to add a new DSAW project in any existing solution to make use of AOSD.

7.3. *Quantitative Evaluation*

To obtain an evaluation of runtime performance and memory consumption, we have assessed the cost of the join-points shown in Table 2. We have evaluated the

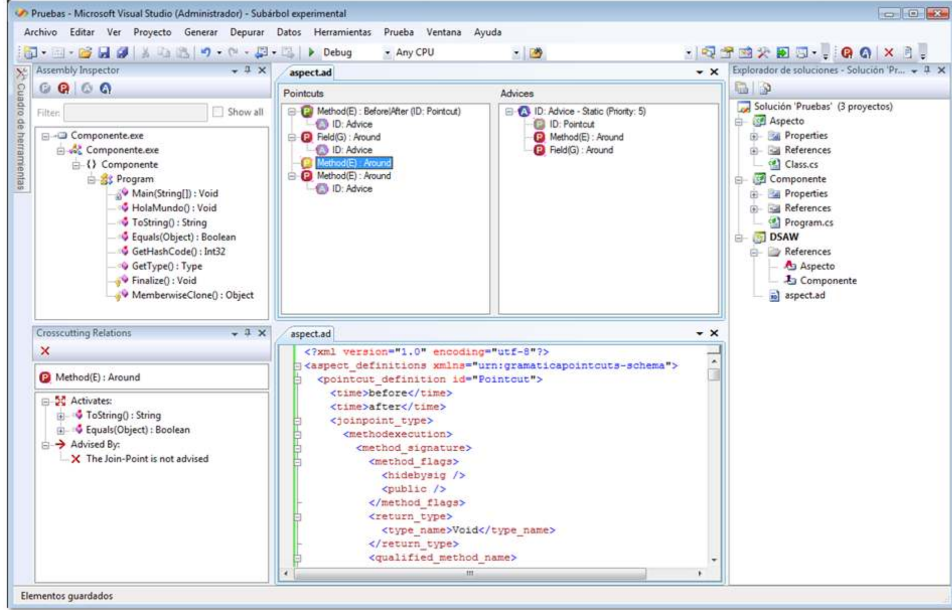

Fig. 13. DSAW Visual Studio plug-in.

execution of 6 join-points in the 10 different AOSD platforms mentioned before.

Table 3: Execution time and memory consumption of join-points.

Join-Point	Time	AspectJ	DSAW Static	DSAW Dynamic	PROSE	JAsCo	Spring Java	Spring .Net	Rapier	Gripper	JBoss AOP
Meth. Call	Before	10,532	10,814	37,363	2,550,900	24,071	30,017	24,527	38,805	38,782	27,794
		13,328	5,940	12,784	9,916	16,288	21,944	11,556	11,972	7,988	21,436
	After	10,510	10,515	37,893	2,564,100	23,814	29,901	24,161	37,836	38,516	28,980
		13,368	5,948	12,772	9,944	15,744	21,896	11,596	12,012	7,976	21,384
	Around	10,382	10,470	37,599	N/A	23,708	27,708	14,912	38,313	41,461	30,208
		13,336	5,796	12,728		15,768	21,876	11,500	12,020	7,960	21,376
Meth.Exec.	Before	11,398	10,814	42,813	N/A	N/A	N/A	N/A	N/A	N/A	27,213
		13,220	7,760	14,556							21,572
	After	11,261	11,289	39,933	N/A	N/A	N/A	N/A	N/A	N/A	29,166
		13,224	5,924	14,444							21,660
	Around	11,293	13,798	42,171	N/A	N/A	N/A	N/A	N/A	N/A	29,851
		13,224	6,628	12,764							21,656
Cons. Call	Before	6,622	11,065	44,086	N/A	N/A	N/A	N/A	67,429	52,137	33,231
		13,320	5,784	12,828					11,960	7,996	21,516
	After	8,181	10,947	41,498	N/A	N/A	N/A	N/A	65,540	52,567	32,180
		13,332	5,772	12,788					12,060	7,964	21,612
	Around	9,870	268,747	N/A	N/A	N/A	N/A	N/A	66,476	55,028	27,530
		13,340	6,312						12,064	8,004	21,524
Cons.Exec.	Before	13,450	10,856	39,384	N/A	N/A	N/A	N/A	N/A	N/A	24,294
		13,216	5,936	12,816							21,664
	After	13,432	10,845	39,210	N/A	N/A	N/A	N/A	N/A	N/A	24,374
		13,212	5,776	12,760							21,668
	Around	15,575	274,176	N/A	N/A	N/A	N/A	N/A	N/A	N/A	26,058
		13,256	6,972								21,740
Field Get	Before	7,352	3,198	31,212	N/A	N/A	N/A	N/A	27,914	27,634	12,613
		13,296	5,772	12,740					11,272	7,448	22,028
	After	6,874	3,178	30,215	N/A	N/A	N/A	N/A	28,474	28,386	12,184
		13,276	5,776	12,760					11,340	7,452	22,064
	Around	6,821	3,305	31,423	1,382,100	N/A	N/A	N/A	N/A	N/A	14,205
		13,296	5,768	12,736	11,568						22,076
Field Set	Before	8,834	3,162	30,680	N/A	N/A	N/A	N/A	28,163	27,605	14,141
		13,284	5,780	12,732					11,320	7,444	22,096
	After	8,854	3,187	29,904	N/A	N/A	N/A	N/A	28,386	27,653	14,154
		13,364	5,744	13,276					11,388	7,412	22,176
	Around	8,680	3,381	32,886	41,417,400	N/A	N/A	N/A	28,255	27,537	15,977
		13,260	5,744	12,748	11,576				11,320	7,424	22,228

For each join-point, we have measured the `before`, `after` and `around` times. The aspect code used in our micro-benchmark accesses the signature and target values (plus argument values in case of methods and constructors execution/call) using the reference to the join-point received (e.g., in AspectJ, `thisJoinPoint` for methods and constructors and `thisJoinPointStaticPart` for fields). In the case of `around` constructor call/execution, the `proceed` method is also called to create the corresponding instance; we also invoke `proceed` in the `around` field set.

Each join-point interception has been run in a loop of 10 million iterations. The first call has not been included to rule out the cost of JIT compilation. Table 3 shows the results; execution time (first row of each join-point evaluation) is expressed in milliseconds and memory consumption (second row) in Kbytes.

A summary of the results is displayed in Figures 14 and 15. All values are shown relative to AspectJ (values were divided by the values of AspectJ). It is worth noting that not every platform implements all the join-points. The average assessment shown in both figures is the geometric mean of each platform values (for *before*, *after* and *around* times).

7.3.1. *Discussion*

From the comparison illustrated in Figures 14 and 15, three major discussions could be identified. The first one is related to the cost of dynamic weaving. We can see how runtime performance of the three full dynamic platforms is significantly lower than the static ones: on average, dynamic DSAW, PROSE and JAsCo are 2.89, 616 and 1.28 times slower than AspectJ. The cost of dynamic weaving in DSAW is obtained by comparing its two weaving implementations, where static weaving is 2.6 times faster than dynamic weaving.

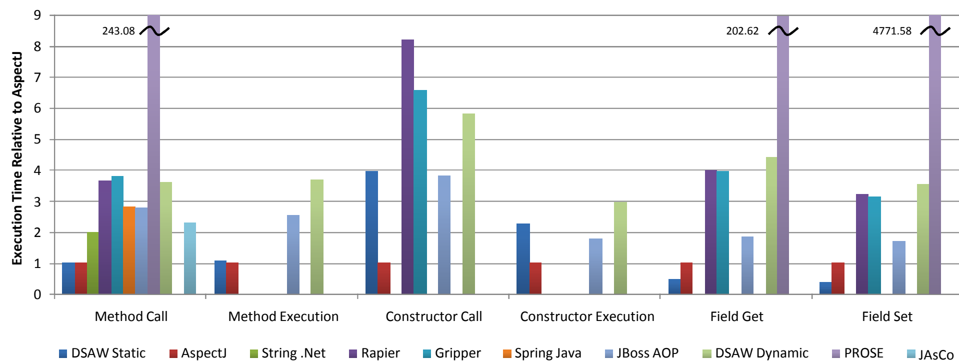

Fig. 14. Execution time relative to AspectJ.

Regarding to memory consumption, only DSAW seems to require more memory resources when weaving is performed at runtime (1.14 times). The rest of dynamic

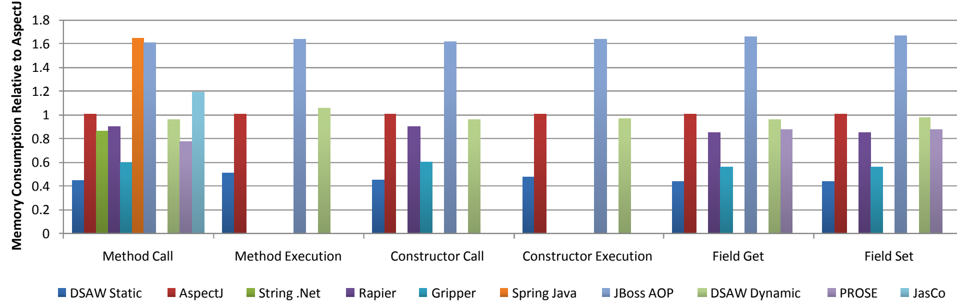

Fig. 15. Memory consumption relative to AspectJ.

weavers do not consume more memory than static ones. This difference between DSAW and the rest of systems must be due to the join-point injection technique we use to implement dynamic weaving.

The second comparison to be established is between static weaving systems (including those that do not obtain the full dynamism degree described in Section 7.1). On average, AspectJ is the fastest one being only 8.16% faster than DSAW, which is the second one. The rest of static weaving platforms require 197% (String .Net), 229% (JBoss AOP), 278% (Spring Java), 418% (Gripper-LOOM.Net) and 424% (Rapier-LOOM.Net) the time used by AspectJ to run the same code.

With regard to memory consumption, DSAW is the best one (45.47% the memory used by AspectJ). Gripper-LOOM.Net, Spring .Net and Rapier-LOOM.Net consume less memory than AspectJ (57.86%, 86.56% and 87.59% respectively). JBoss AOP and Spring Java require 63.68% and 64.15% more memory than AspectJ. Therefore, we can see how the good runtime performance of AspectJ is counteracted with a higher memory consumption. In fact, DSAW obtains the highest performance per memory consumption ratio, being more than order of magnitude better than AspectJ (the second-best one).

Finally, those dynamic weaving platforms that offer a real decoupling between applications and aspects (Dynamic DSAW, PROSE and JAsCo) are also compared. JAsCo offers the best runtime performance, being 1.27 times slower than AspectJ. Dynamic weaving in DSAW is 71.09% slower than JAsCo and more than 157 times faster than PROSE. It is worth noting that the good runtime performance of JAsCo is obtained implementing only one join-point (see Table 7.2). JAsCo utilizes the Java agents for program instrumentation included in the `java.lang.instrument` package of the Java 1.5 release. Although this technology permits the optimal replacement of code at runtime, it seems to be difficult to use it for the implementation of the remaining 5 join-points. The low runtime performance showed by PROSE is explained by the mechanism used to interconnect aspects and components. PROSE uses the JVMDI/JVMTI event notification API that lowers the overall performance of the system.

Measuring the average performance per memory ratio, DSAW uses 38.09% less memory than JAsCo to obtain the same runtime performance, whereas PROSE requires 189 times more memory than DSAW.

7.3.2. *The Cost of Weaving*

The benefits of using AOSD also involve a cost of runtime performance. Hilsdade and Hugunin showed that AspectJ adds a maximum 20% performance overhead relative to hand-coded implementations [51]. We have reproduced the experiment described by them where AspectJ involved a 3% performance penalty [51], obtaining a 1.13% performance cost using the static weaver of DSAW.

At the same time, we have previously seen how dynamic weaving commonly implies a performance cost as well. DSAW shows an average performance penalty of 260.37% with regard to static weaving. We have measured this cost, breaking it down into the following percentages. The code injected by the JPI, when no aspect has been woven, implies a 31.2% of the total cost. The remaining performance penalty (68.8%) is produced by the dynamic dispatching of messages from applications to components. This dynamic dispatch selects the advice to be called when many aspects intercept the same join-point (the α function described in Section 4.2.2). To perform the dynamic invocation of advice, DSAW creates a new method stub at runtime using CodeDOM. This strongly typed method invocation avoids the important performance penalty caused by the usage of reflection in the .NET platform [52].

7.4. *Real Applications*

We[b] have used both the static and dynamic weavers of DSAW to develop security issues of distributed systems [53]. DSAW has been used to implement access control, data flow and encryption of transmissions in two different scenarios. Details of both implementations are presented in [53].

The first scenario is based on distributed systems made up by mobile devices, where network topologies and communication channels may dynamically change. If the user is connected to a distributed system and it is detected that the communication channel is not secure any more, encryption of transmissions may be required. Therefore, a dynamic encrypting aspect is woven with the application that uses the distributed system while the system is running. The aspect is even able to forward the channel to another secure one if the mobile device allows it. Any kind of encryption or forwarding aspect can be woven at runtime, because the DSAW dynamic weaver does not impose any coupling between aspects and components. Finally, if the mobile device returns to a trusted environment, the encrypting aspect is unwoven to avoid the unnecessary overhead of encryption.

[b]In collaboration with the Liverpool John Moores University.

In the second scenario, we tackle vulnerabilities caused by the flow of data through the network. Each node in the network has an authorization level. The security policy of the distributed system dictates that a node with an authorization level can only send and receive information from those nodes with greater or equal authorization level [54]. Figure 16.a shows an example. Nodes 1 and 4 can send information to any other node because the *confidential* level is the lowest one. Node 2 can only send information to node 3, since the *secret* authorization level is lower than *top secret*. Finally, node 3 cannot send information to anyone because it has the highest authorization level.

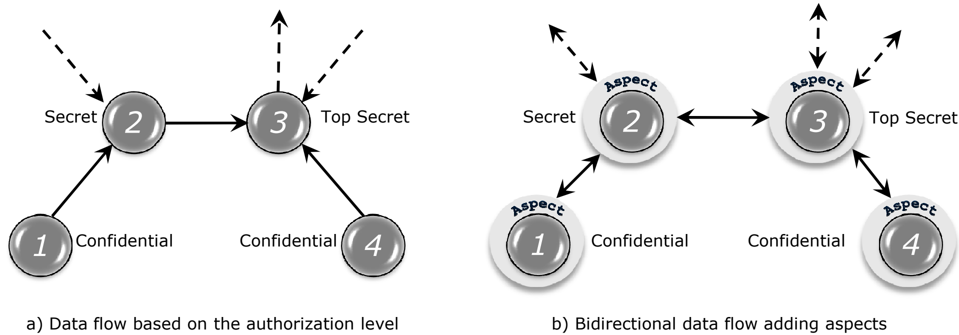

Fig. 16. Using aspects to modify the data flow.

The traditional implementation only considers one-to-one relationships [55], implying restrictions on data flow in point-to-point networks with changing topologies. For example, nodes 1 and 4 in Figure 16.a have the same access level, but they cannot exchange information because node 3 cannot relay messages to nodes 2 and 4.

We have used the static weaver of DSAW to implement a distributed system with this security policy that guarantees the secure transmission of information over changing topologies, tagging data with the authorization levels of nodes. Applications are built relying on the classical *send* and *receive* operations, and aspects intercept these two messages to include the following functionalities:

(1) Encryption of information to avoid unauthorized access to it.
(2) Authentication to grant the user the appropriate authorization level.
(3) Data tagging to determinate how information flows across the network and to control the access to it.

As shown in the right part of Figure 16, all nodes can now exchange information between them regardless of their authorization level, because aspects control the data flow and restrict the access to data. As a result, nodes 1 and 4 can securely exchange data through nodes 2 and 3.

7.4.1. *Performance and Memory Cost*

We have assessed the performance and memory costs of using AOSD in these two systems. The aspect-oriented implementation was compared with an equivalent object-oriented program that offers the same functionalities, but with the corresponding aspect code tangled throughout the application. In the static weaving scenario, aspects involved a 3.89% and 1.41% cost of runtime performance and memory consumption respectively. Costs did not depend on the number of messages sent.

For the encryption scenario, we used dynamic weaving. In this case, the runtime performance penalty was 59.18% and the increase of memory consumption was 55.96%. These results show how the assessment presented above (Section 7.3) represents the maximum cost of weaving. Since we measured the costs of join-point interception, the maximum performance penalty is obtained when *every* join-point is woven. These two systems show the overall performance and memory costs of two real aspect-oriented applications developed in DSAW.

8. Related Work

There exist many static weaving AOSD tools, and there are also some dynamic ones. However, there are few that offer both approaches.

Wicca is one example of a dynamic and static aspect-oriented system [56]. Wicca has been developed over the .NET platform making use of the Phoenix framework – a back-end compiler infrastructure [57]. Wicca performs static weaving by means of code instrumentation. They achieved dynamic weaving using the debugging API of the CLR. Dynamic weaving is released in an alpha version, and it does not support dynamic aspect deletion yet. The static and dynamic weavers are not equivalent; static weaving is more expressive than the dynamic one [22]. Current runtime performance of Wicca is not competitive because applications should be executed in debugging mode, enabling the *edit-and-continue* support of the CLR [22].

AOP.NET, formerly called NAop, is another dynamic and static weaving proposal –no implementation has been released yet [13]. Its design follows a proxy-based component decoration. This proxy is used in both static and dynamic scenarios. The weaver uses a proxy class instead of each component class. The proxy adapts the behavior of its decorated class. Depending on the pointcuts, the proxy delegates its functionality on the original class or it calls the registered aspects. The static weaver performs this process prior to application execution, whereas the dynamic weaver does it at runtime.

The LOOM.NET project provides dynamic and static weaving over the same core implementation, using the .NET platform [48]. Rapier LOOM.NET is the dynamic weaver. Pointcuts in aspects are expressed by means of custom attributes in .NET. At load-time, the application is woven together with aspects. Applications and aspects should be linked prior to their execution. A static weaver, called Gripper-LOOM.NET (in version 0.92), is currently being developed [58]. The syntax

of the pointcut language description is not the same for both weavers. This makes it difficult to convert aspects from static to dynamic. The dynamic weaver needs the source code of applications and does not provide an ample set of join-points [48, 59]; dynamic aspect deletion is not supported either [20].

Regarding aspect formalization, most existing semantics consider object oriented base programs [60, 31, 61, 62], while others consider functional languages [29], as well as process calculi [63]. The *Common Aspect Semantics Base* (CASB) specifies the aspect semantics independently from its base language [15]. For each aspect feature, it is introduced a minimal construction of the base language necessary to plug aspects in. The CASB semantics was previously used to discuss the benefits of scheduling aspects at runtime [33]. The static scheduling semantics of AspectJ was extended with the concept of *aspect group*, the aspects scheduled for the current join-point, with the opportunity to access them from the advice body.

9. Conclusions

There are many tools that support AOSD. Some of them offer application weaving before its execution, while others provide this adaptation at runtime. Although the static approach is suitable in many cases, dynamic adaptation may also be required when the application should respond to runtime emerging contexts and requirements. Static weaving supports efficient AOSD, whereas dynamic weaving involves runtime adaptiveness. DSAW is an AOSD platform that obtains the benefits of both sorts of weavers, combining the features of different existing platforms into one system.

The design of DSAW has been performed following the *Separation of Concerns* principle, so that the weaving-time concern does not interfere in the aspect-oriented development process. Aspects can be changed from static to dynamic and vice versa depending on the life cycle stage, and both type of weavers can be used in the same application. This facilities both *edit-and-continue* development (at first stages of software development) and efficient static weaving (when the application is about to be released). That is, DSAW completely separates the weaving-time concern. The programmer may modify the flexibility/performance trade-off during the development life cycle.

DSAW has been designed over the .NET platform, taking advantage of its features. Both weavers use byte-code instrumentation, making DSAW be language neutral. This permits the adaptation of legacy applications, and promotes aspects and components reuse. The system offers a wide set of join-points, language and platform neutrality, and full dynamic program adaptation.

The assessment of runtime performance has shown that the static weaver of DSAW is the second fastest, being on average 8.16% slower than AspectJ. Comparing full dynamic weavers, DSAW is 71.09% slower than JAsCo and more than 157 times faster than PROSE. Considering the performance per memory ratio, DSAW has obtained the best measurements for both dynamic and static weaving. Two

real applications developed in DSAW have entailed a performance cost of 3.89% and 59.18%, respectively comparing static and dynamic weaving with the traditional object-oriented development.

Future work will be focused on designing a conflict resolution mechanism, including composition operators [44] and semantic conflict correction techniques [46]. We are also extending the developed Visual Studio plug-in to facilitate the dynamic (un)weaving of aspects.

Current implementation of DSAW can be freely downloaded from its Web page at `http://www.reflection.uniovi.es/dsaw`

Acknowledgements

This work was supported by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation, projects TIN2008-00276 and TIN2011-25978; it has also been partially funded by the University of Oviedo, with the project entitled *AOSD System to support both Dynamic and Static Weaving in a Language and Platform Neutral way*, UNOV-08-MB-15.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *ECOOP: European Conference on Object-Oriented Programming*, (Berlin), pp. 220–242, Springer Verlag, 1997.

[2] W. Hürsch and C. Lopes, *Separation of Concerns, Technical Report NU-CCS-95-03*. Boston: Northeastern University, 1995.

[3] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, December 1972.

[4] F. Ortin, B. Lopez, and J. B. G. Perez-Schofield, "Separating adaptable persistence attributes through computational reflection," *IEEE Software*, vol. 21, no. 6, pp. 41–49, 2004.

[5] J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural support for quality of service for corba objects," *TAPOS*, vol. 3, no. 1, pp. 55–73, 1997.

[6] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. L. Lawall, "Web cache prefetching as an aspect: towards a dynamic-weaving based solution," in *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 110–119, ACM, 2003.

[7] A. Stevenson and S. MacDonald, "Dynamic aspect-oriented load balancing in java rmi," in *PDPTA: Parallel and Distributed Processing Techniques and Applications*, pp. 485–491, 2008.

[8] A. Popovici, T. Gross, and G. Alonso, *Dynamic Homogenous AOP with PROSE, Technical Report*. Department of Computer Science, ETH Zürich, 2001.

[9] K. Böllert, "On weaving aspects," in *Proceedings of the Workshop on Object-Oriented Technology*, (London, UK), pp. 301–302, Springer-Verlag, 1999.

[10] M. Haupt and M. Mezini, "Micro-measurements for dynamic aspect-oriented systems," in *Net.ObjectDays*, (Berlin), pp. 81–96, 2004.

[11] M. Dmitriev, "Applications of the hotswap technology to advanced profiling," in *USE2002: First International Workshop on Unanticipated Software Evolution*, (Berlin), Springer, 2002.

[12] M. Eaddy and S. Feiner, *Multi-Language Edit-and-Continue for the Masses*. New York: Technical Report CUCS-015-05, Department of Computer Science, Columbia University, 2005.

[13] M. Blackstock, "Aspect weaving with C# and .NET." `http://www.cs.ubc.ca/michael/publications/AOPNET5.pdf`, 2004.

[14] W. Gilani, F. Scheler, D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat, "Unification of static and dynamic aop for evolution in embedded software systems," in *Software Composition*, pp. 216–234, 2007.

[15] S. D. Djoko, P. Fradet, and D. L. Botlan, *CASB: Common Aspect Semantics Base, Deliverable 5*. AOSD-Europe, EU Network of Excellence in AOSD, 2006.

[16] Apache Software Foundation, "Apache log4net, logging services." `http://logging.apache.org/log4net/index.html`.

[17] A. Nicoară and G. Alonso, "Dynamic aop with prose," in *1st International Workshop on Adaptive and Self-Managing Enterprise Applications*, pp. 125–138, 2005.

[18] JBoss Community, "Jboss AOP homepage." `http://labs.jboss.com/jbossaop/`.

[19] D. Suvée, W. Vanderperren, and V. Jonckers, "Jasco: an aspect-oriented approach tailored for component based software development," in *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 21–29, ACM, 2003.

[20] A. Frei, P. Grawehr, and G. Alonso, *A Dynamic AOP-Engine for* .NET*, Technical Report 445*. Zürich: Department of Computer Science, ETH, 2004.

[21] The Spring Framework, "Spring java/j2ee application framework, reference documentation, version 2.5.5." `http://static.springframework.org/spring/docs/2.5.x/spring-reference.pdf`.

[22] Computer Science Department, Columbia University, "Wicca v2 home page." `http://www1.cs.columbia.edu/~eaddy/wicca/`.

[23] L. Vinuesa and F. Ortin, "A dynamic aspect weaver over the .net platform," in *Metainformatics*, pp. 197–212, 2003.

[24] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini, "Adapting virtual machine techniques for seamless aspect support," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, (New York, NY, USA), pp. 109–124, ACM, 2006.

[25] F. Ogel, G. Thomas, and B. Folliot, "Supporting efficient dynamic aspects through reflection and dynamic compilation," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, (New York, NY, USA), pp. 1351–1356, ACM, 2005.

[26] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt, "An expressive aspect language for system applications with arachne," in *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 27–38, ACM, 2005.

[27] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger, "Debugging aspect-composed programs," in *SC2007: International Symposium on Software Composition*, 2007.

[28] W. Schroder-Preikschat, D. Lohmann, F. Scheler, W. Gilani, and O. Spinczyk, "Static and dynamic weaving in system software with aspectc++," in *HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, (Washington, DC, USA), p. 214.1, IEEE Computer Society, 2006.

[29] D. Walker, S. Zdancewic, and J. Ligatti, "A theory of aspects," in *ACM International Conference on Functional Programming*, (Uppsala), Aug. 2003.

[30] R. Jagadeesan, A. Jeffrey, and J. Riely, "Typed parametric polymorphism for aspects," *Sci. Comput. Program.*, vol. 63, no. 3, pp. 267–296, 2006.

[31] R. Lämmel, "A Semantical Approach to Method-Call Interception," in *Proceedings of*

the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), (Twente, The Netherlands), pp. 41–55, ACM Press, Apr. 2002.

[32] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP: European Conference on Object-Oriented Programming*, (Berlin), pp. 327–353, Springer Verlag, 2001.

[33] A. Assaf and J. Noyé, "Dynamic aspectj," in *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, (New York, NY, USA), pp. 1–12, ACM, 2008.

[34] European Computers Manufacturing Association, *Standard ECMA-335, Common Language Infrastructure (CLI), 4th edition*. ECMA - European Computers Manufacturing Association, 2006.

[35] D. Lafferty and V. Cahill, "Language-independent aspect-oriented programming," in *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, (New York, NY, USA), pp. 1–12, ACM, 2003.

[36] Sun Microsystems, *Java Specification Request (JSR) 220. Enterprise Java Beans, version 3.0. Java Persitence API*. Java Community Process, 2006.

[37] G. Kiczales and J. D. Rivieres, *The Art of the Metaobject Protocol*. Cambridge, MA, USA: MIT Press, 1991.

[38] F. Ortin, J. M. Cueva, and A. B. Martinez, "The reflective nitro abstract machine," *SIGPLAN Not.*, vol. 38, no. 6, pp. 40–49, 2003.

[39] H. G. Andchris, H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," in *Compiler Construction, volume 2622 of Springer Lecture Notes in Computer Science*, pp. 46–60, Springer, 2003.

[40] Weave.NET homepage, "Trinity college dublin." `http://www.dsg.cs.tcd.ie/index.php?category_id=193`.

[41] S. Miller, *DEC/HP Network Computing Architecture Remote Procedure Call Run Time Extensions version OSF TX1.0.11.* , Open Software Foundation, Cambridge, MA, 1992.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[43] I. Nagy, L. Bergmans, and M. Aksit, "Composing aspects at shared join points," in *Conference proceedings : Erfurt, Germany*, Lecture Notes in Informatics 69, pp. 19–38, 2005.

[44] É. Tanter, "Aspects of Composition in the Reflex AOP Kernel," in *Software Composition, volume 4089 of Lecture Notes in Computer Science*, pp. 98–113, Springer, 2006.

[45] R. Douence, P. Fradet, and M. Südholt, "Composition, reuse and interaction analysis of stateful aspects," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 141–150, ACM, 2004.

[46] P. Durr, T. Staijen, L. Bergmans, and M. Aksit, "Reasoning about semantic conflicts between aspects," in *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005.

[47] The Spring Framework, "Spring.net reference documentation, version 1.2.0 m1." `http://www.springframework.net/docs/1.2.0-M1/reference/pdf/spring-net-reference.pdf`.

[48] W. Schult and P. Trögger, "Loom.net — an aspect weaving tool," in *Workshop on Aspect-Oriented Programming, European Conference on Object-Oriented Programming*, (Darmstadt, Germany), 2003.

[49] J. Brichau and M. HauptSurvey, *Survey of Aspect-oriented Languages and Execution*

Models. Document VUB-01, AOSD-Europe, 2005.

[50] L. Vinuesa, *Dynamic Separation of Aspects by means of Language and Platform Neutral Computational Reflection, Ph.D. Dissertation*. University of Oviedo, Spain, 2007.

[51] E. Hilsdale and J. Hugunin, "Advice weaving in AspectJ," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 26–35, ACM, 2004.

[52] F. Ortin, J. M. Redondo, and J. Baltasar García Perez-Schofield, "Efficient virtual machine support of runtime structural reflection," *Science of Computing Programming*, vol. 74, no. 10, pp. 836–860, 2009.

[53] M. Garcia, D. Llewellyn-Jones, M. Merabti, and F. Ortin, "Applying Dynamic Separation of Aspects to Distributed Systems Security, Technical Report." `http://www.reflection.uniovi.es/publications/2010/DSS.pdf`, 2010.

[54] N. C. S. C. NCSC, "Trusted network interpretation environments guideline," 1990.

[55] U. Lang and R. Schreiner, *Developing secure distributed systems with CORBA*. Artech House Publishers, 2002.

[56] M. Eaddy, "Wicca 2.0: Dynamic weaving using the .NET 2.0 debugging api," in *AOSD 2007: Aspect-Oriented Software Development*, 2007.

[57] Microsoft Research, "Phoenix academic program." `http://research.microsoft.com/phoenix`.

[58] HPI (Hasso Plattner Institut), "Gripper-loom.net." `http://www.dcl.hpi.uni-potsdam.de/research/loom/gripper_loom.htm`.

[59] K. Köhne, W. Schult, and A. Polze, *Design by contract in* .NET *Using Aspect Oriented Programming, Technical Report*. Hasso Plattner Institut, Universität, Potsdam, Germany, 2005.

[60] R. Jagadeesan, A. Jeffrey, and J. Riely, "A calculus of untyped aspect-oriented programs," in *European Conference on Object-Oriented Programming*, pp. 54–73, 2003.

[61] R. Douence and L. Teboul, "A pointcut language for control-flow," in *Generative Programming and Component Engineering (GPCE)*, pp. 95–114, 2004.

[62] M. Wand, G. Kiczales, and C. Dutchyn, "A semantics for advice and dynamic join points in aspect-oriented programming," *ACM Transactions on Programming Languages and Systems*, vol. 26, pp. 890–910, September 2004.

[63] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely, "μabc: A minimal aspect calculus," in *Concurrency Theory, volume 3170 of Lecture Notes in Computer Science*, pp. 209–224, Springer, 2004.