Journal of Universal Computer Science, vol. 17, no. 5 (2011), 742-776
submitted: 8/2/10, accepted: 4/11/10, appeared: 1/3/11

An Aspect-Oriented Framework for Weaving Domain-Specific Concerns into Component-Based Systems

Frédéric Loiret, Romain Rouvoy, Lionel Seinturier, Daniel Romero
Kévin Sénéchal
(INRIA Lille – Nord Europe, France
University of Lille 1 - LIFL, France
{firstname.lastname}@inria.fr)

Aleš Plšek
(Computer Science Dept.
Purdue University, West Lafayette, USA
aplsek@purdue.edu)

Abstract: Software components are used in various application domains, and many component models and frameworks have been proposed to fulfill domain-specific requirements. The general trend followed by these approaches is to provide ad-hoc models and tools for capturing these requirements and for implementing their support within dedicated runtime platforms, limited to features of the targeted domain. The challenge is then to propose more flexible solutions, where components reuse is *domain agnostic*. In this article, we present a framework supporting compositional construction and development of applications that must meet various extra-functional/domain-specific requirements. The key points of our contribution are: *i)* We target development of component-oriented applications where extra-functional requirements are expressed as annotations on the units of composition in the application architecture. *ii)* These annotations are implemented as open and extensible component-based containers, achieving full separation of functional and extra-functional concerns. *iii)* Finally, the full machinery is implemented using the *Aspect-Oriented Programming* paradigm. We validate our approach with two case studies: the first is related to real-time and embedded applications, while the second refers to the domain of distributed context-aware middleware.
Key Words: component-based frameworks, domain-specific software engineering, aspect-oriented software architectures.
Category: D.2.2, D.2.11, D.2.13

1 Introduction

Component-Based Software Engineering (CBSE) promotes software architectures by decomposing systems into logical modules, which can be easily packaged and composed. CBSE has therefore emerged as an elegant technology for the rapid assembly of flexible systems, fostering reuse and separation of concerns. Although CBSE is already applied in IT systems using mainstream component technologies, such as *Enterprise Java Beans* (EJB) or *CORBA Component Model* (CCM), its benefits increase drastically when designing *Domain-Specific Component Frameworks* (DSCF) in various application domains [LT09, LW07], from dynamic adaptability [BCL+06] and distribution support [SVB+08] to embedded applications imposing strict constrains in terms of performance [CHP06]. A DSCF typically consists of a component model isolating the relevant domain-specific abstractions and a toolkit for the generation of a dedicated execution platform providing all the extra-functional services required by the application.

However, the plethora of DSCF propositions conflicts with the principles established by CBSE, since each DSCF tends to provide its own abstractions and non-extensible platforms, which therefore prevents any reuse or improvement of legacy solutions when generating the dedicated frameworks. This raises the challenge of proposing more flexible solutions, where components can be designed as pure logical modules, which can be independently deployed in various execution contexts depending on these domain-specific/extra-functional requirements. According to our experience in this field [LNBL09, PSCD06, PLMS08, SMF+09], we believe that DSCFs actually share many architectural concepts, design patterns, and principles that are applied when implementing the domain-specific parts of these execution platforms.

To solve these issues, the contributions exposed in this article revisit the coupling of the *Aspect-Oriented* (AO) paradigm with an *Architecture Description Language* (ADL). More specifically, we are interested in an approach where aspects are implemented as architectural fragments woven within component containers. The idea of relying on AO techniques to inject domain-specific/extra-functional features into an ADL has already been addressed in the state-of-the-art. Nonetheless, the originality of our approach resides in the following two contributions: *First*, the weaving mechanisms provided to the developer for expressing the composition of aspects from the ADL typically rely on *pointcut expressions* and on *advice implementation languages* [aos]. Existing approaches, such as CAM/DAOP [PFT05], FAC [PSCD06], FuseJ [SFV06], or the AO component and composition model defined by the AOSD-Europe project [Pro08] rely on low-level and verbose languages capturing all the composition rules between the ADLs and their advice implementations. These solutions compose business components and aspect connectors in a single software entity, thus partially violating the obliviousness principle of aspect orientations. Within our approach, we rather promote a higher-level specification language for expressing the composition between these two levels based on an *annotation weaving process*. Annotations are used as concise domain-specific notations, which are easier to write, reuse, and analyze to check potential conflicts. These annotations can be associated to business components *a posteriori* via pointcut expressions, which select the target architectural artefacts to be woven. *Second*, we consider the way advices are implemented and the platform resulting from the weaving process of aspects on the core architecture. The existing approaches sharing with our proposal the capability to implement aspects as architectures [Pro08] or those which propose an

Aspect-Oriented Architecture Description Language, such as AO-ADL [PF07], AspectualACME [GCB^{+}06], or AspectLEDA [MTM09], do not rely on a concise platform model. Indeed, in these approaches, the weaving process results in a flatten component-based architecture preventing any distinction at runtime between business and non-functional components. We rather believe that the weaving process should preserve the separation of concerns, and we propose to exploit the concept of *container* to isolate the non-functional concerns. In particular, the containers we consider are implemented as composite components extending business components with their domain-specific behaviors. In this sense, a container acts as a concise weaving infrastructure, which is based on architectural patterns specifying the composition rules between the component contents and the selected aspects.

Therefore, this article introduces an incremental weaving process combining aspects and components homogeneously from the application down to the platform. At the application level, the system is designed as a software architecture centered on the business logic and is incrementally specialized using domain-specific annotations. These annotations act as meta-information reified at the architectural level, thus expressing the domain-specific requirements of the application. The expression of these annotations is uncoupled from the base architecture, allowing the tailoring of the latter to various execution contexts by means of an annotation injection process. At the platform level, each woven annotation is reified as an aspect, which is implemented as a fine-grained component-based architecture. Containers are then used as a base infrastructure, in which domain-specific concerns can be injected and composed to generate dedicated containers, which perfectly conform to the domain-specific requirements of the application.

The remainder of this article is organized as follows. Section **2** introduces the background of this work and clarifies our proposal. Sections **3** and **4** expose our approach at application and platform levels, respectively. Section **5** provides some insights on implementation issues. Section **6** evaluates our approach on two case studies. Finally, Section **7** discusses the related work, while Section **8** concludes the article.

2 Hulotte: Component Model & Design Process

2.1 Background

A *Domain-Specific Component Framework* (DSCF) is composed of a component model and the associated toolkit enabling the assembly, deployment, and execution of specific applications [BHM09]. In particular, the DSCF component model defines the relevant architectural abstractions close to the *problem domain*, according to the requirements of the targeted application domain (*e.g.*, to address the distribution support or real-time constraints). According to [CL02], a recognized methodology for developing DSCF is composed of two main steps:

First, the *domain-specific component model* is used to develop the functional concerns of an application: the *applicative components*. Typically, the applicative components encapsulate the business logic of the application and are specified using the key abstractions for the considered domain.

Second, the *domain-specific framework toolkit* is employed to create a *dedicated runtime platform*. Most of component frameworks provide a runtime platform for hosting and running components following the container idiom (*e.g.*, the EJB container). In this case, containers provide, in a quasi-transparent manner, platform-wide services to application components, thus relieving the developer from dealing with domain-specific requirements and implementing the associated execution support.

Although one of the main benefits expected in using the component paradigm is reuse [Cle02], it has been argued [DHT01] that the vast and increasing number of proposals to address these domain-specific requirements does not encourage reuse. In particular, these container-oriented platforms are often statically configured to support a fixed set of extra-functional services, which limits the integration of additional domain-specific concerns. From this observation, we proposed HULOTTE [LMP+09]: a framework for the specification and implementation of arbitrary domain-specific concerns in a unified way, which is easily extendable towards different application domains. In the following section, we present the generic component model on which this framework relies, the mechanisms proposed to extend it towards different domains, as well as the roles involved within its design process. The latter is based on aspect-oriented weaving techniques introduced in Section 2.3.

2.2 Hulotte Component Model

HULOTTE relies on a component model based on general CBSE principles [Cle02], and is inspired by the reflective FRACTAL component model [BCL+06]. In particular, HULOTTE identifies as core architectural artifacts the concepts of `Component` (either `Primitive` or `Composite`), `Attribute`, `Interface`, and `Binding`. The behavior of a primitive component is implemented by the underlying programming language supported by our framework (Java or Scala in the context of this article) and is reified by a `Content` artifact. An architecture is then specified as a set of interconnected components (at an arbitrary level of encapsulation by using the composite design pattern) via oriented relationships between required and provided `Interfaces`. Finally, an `Interface` type is specified by a signature specified within an *Interface Description Language* (IDL) file (*e.g.* a Java interface specified in a `.java` file in the context of this article). An example of a HULOTTE architecture is given Figure 2.

According to the two above described steps for developing DSCF, we distinguish two roles involved in the HULOTTE development process: the *application developer* and the *platform developer*. The *application developer* is responsible for the development of *applicative components* and the specification of domain-specific requirements. She/he uses the HULOTTE model concepts to design the component-based application, which is annotated by `Domain Specific Annotations` afterwards. These annotations mark the HULOTTE `Architectural Artifacts` like Java 5 annotations mark the *Abstract Syntax Tree* (AST) of a Java program. HULOTTE annotations isolate and specify the concerns relevant to a targeted application domain, so-called *domain-specific concepts*. Within our approach, it should be noticed that components are used as pure business units, and a component-based architecture then implements the whole business logic of the application. Therefore, annotations are used to specify the domain-specific semantics over the architecture. For instance, in order to address the multi-task applications domain, an annotation can be used on a component to qualify under which *execution model* its business interfaces should be invoked, *e.g.* periodically or sporadically. On a composite component, an annotation can qualify the boundary of a memory scope in which its subcomponents will be allocated. Finally, an annotation can also be used on a binding to specialize the *communication models and protocols* (*e.g.*, asynchronous, shared memory, CORBA, SOAP) between the bound components.

The role of the *platform developer* is therefore to design and implement the runtime platform generation process, and the domain-specific requirements defined by the application developer. Within our approach, each application component is hosted by a container, which is itself implemented as a component-based architecture. Therefore, containers and applications are homogeneously implemented using the same architectural concepts. Throughout this article, we will refer to *platform components* for components implementing the logic of the containers.

2.3 Hulotte Aspect-Oriented Process

The contribution of this article consists in the use of aspect-oriented weaving techniques at two levels: At application level, for weaving HULOTTE annotations on the applicative architecture, and at platform level, for weaving platform components within containers.

The design flow described throughout this article is depicted in Figure 1, as well as the three roles it involves: *i)* the application developer, *ii)* the platform developer and *iii)* the HULOTTE process itself.

For clarity sake, we detail in the following a typical design scenario from the application developer point of view based on this flow, according to the steps

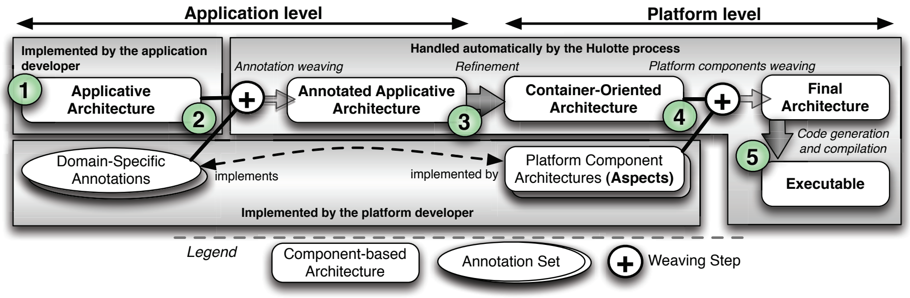


Figure 1: The Aspect-Oriented HULOTTE Process.

numbered in Figure 1: The application developer is in charge of implementing the *applicative architecture* according to its business requirements (`step 1`). Then, she/he injects the domain-specific annotations (`step 2`) in a way which is uncoupled from its architecture by means of a dedicated pointcut language, as it will be presented in Section **3**. This first weaving step outputs an annotated architecture. The latter is refined according to our *container model* (`step 3`), which acts as a host structure for weaving platform components, implemented as aspects. The latter implement domain-specific annotation's logic. This leads us to the second weaving step (`step 4`) which outputs the final architecture. These steps are presented in Section **4**. Finally, a executable is generated (`step 5`).

3 Hulotte Application Level

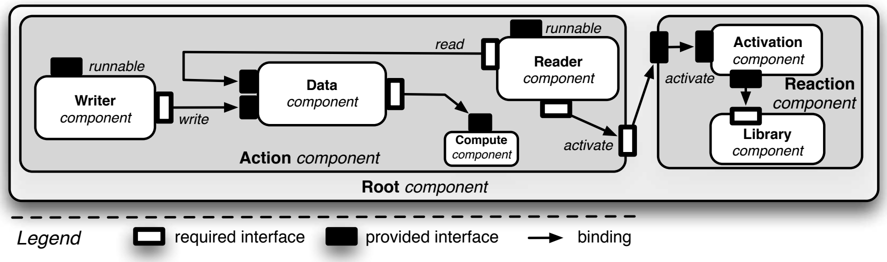
The HULOTTE application level corresponds to the design space provided to the *application developer*. Therefore, the developer composes the application business logic as an assembly of components, mentioned throughout this article as the *applicative architecture*. Then, this applicative architecture is tagged with annotations specific to the target application domain. This section gives some insights on these two design steps.

3.1 Applicative Architecture Design

An illustration of an *applicative architecture* instance (and an excerpt of its textual ADL[1], called HULOTTE-ADL) designed with the HULOTTE component model is sketched out in Figure 2. White boxes represent primitive components and are implemented by the application developer. Primitive components are

[1] At the textual ADL level, there is no distinction between *composite* and *primitive* components.

elementary composition units, connected via bindings and encapsulated within composite components (grey boxes). At this stage of the process, an applicative architecture simply specifies how components are connected together to compose the business logic of the application.




```
component Root {
  component Action {
    component Writer {
      provides java.lang.Runnable as runnable
      requires WriteItf as write
      content WriterImpl
    }
    component Data {
      provides WriteItf as write
      provides ReadItf as read
      requires ComputeItf as compute
      content DataImpl
    }
    component Reader {...}
    component Compute {...}
    binds Writer.write to Data.write
    // ... Other internal Action bindings
  }
  component Reaction {...}
  binds Action.activate to Reaction.activate
}
```

Figure 2: Applicative Architecture Example.

3.2 Weaving of Domain-Specific Annotations

The Hulotte framework exploits annotations to isolate the domain-specific concerns of an applicative architecture. They are explicitly specified by the application developer by means of a weaving process. This enables the uncoupling of the annotation expression from the target architecture. This idea is motivated by the fact that the same applicative architecture can be used in various execution contexts. This section therefore introduces the characteristics of the Hulotte aspect-oriented approach in order to automatically weave domain-specific annotations into the applicative architecture, focusing on the architectural *join point* *model* and its associated *pointcut language*.

Joint Point Model. HULOTTE identifies four types of architectural join points corresponding to any `Architectural Artifacts` of the HULOTTE metamodel that can be marked with a *Domain-Specific Annotation*—*i.e.*, `Component`, either primitive or composite, `Interface`, `Binding` and `Attribute`.

Pointcut Language. Our framework provides a pointcut language to select the join points of the applicative architecture on which domain-specific annotations should be attached. This pointcut language is structured as a query language, called HPATH (*Hulotte Path*), which is inspired by FPATH [DLLC09] and XPATH languages. HPATH is therefore a *Domain-Specific Language* (DSL) that provides a concise and powerful notation to navigate and query HULOTTE architectures. With HPATH, the application developer describes the domain-specific pointcut using dedicated architectural concepts. Finally, it allows to attach `Domain-Specific Annotations` to the `Architectural Artifacts` captured by the pointcut. The syntax of a HPATH expression is depicted in Figure 3.

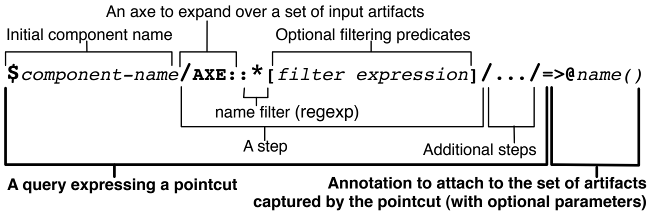


Figure 3: Syntax of a HPATH Expression.

An HPATH query expressing a pointcut is composed of a *root component name* representing the initial artifact on which the query will be executed, and a *sequence of steps* used to navigate through the applicative architecture. The sequence of steps, separated by a slash (`/`) acts as a *pipes and filters* flow, each step consuming and producing a set of HULOTTE's architectural artifacts. Each step is composed of a *navigation axis*, followed by a *name filter* and an optional expression of *filtering predicates*. Several navigation axis are available (most of them are inherited from the FPATH language), for instance:

– `child` (resp. `parent`): from an input set of components, this axe returns their direct subcomponents (resp. their direct super-components).
– `descendant-or-self` (resp. `ancestor-or-self`) corresponds to the transitive and reflective closure of `child` (resp. `parent`). In these cases, the suffix `-or-self` can be omitted, meaning that the axe's input set of components will not be added to the output result set.

– `interface` returns the set of interfaces belonging to a set of input components.
– `binding` returns the set of bindings incoming or outgoing to/from the set of input architectural artifacts (this axe supports either components or interfaces as input).
– `bound-to` returns the set of components bound to the components given as input.
– `method` returns the set of methods specified by interface signatures.

Within a step expression, a *name filter* is then applied over the set of elements returned by the operation. The *name filter* is a regular expression that keeps the matching `Architectural Artifacts`. The last part of a step expression is a sequence of filtering predicates (optional), filtering the resulting set before it is consumed by the next step. Each `Architectural Artifact` accepts a set of predicates, for instance:

– The predicate `size` allows to test the number of artifacts processed by a step.
– `required` and `provided` are predicates that apply to the interface or binding artifact in order to restrict a set of interfaces according to their kind (*i.e.*, required or provided). The predicate `signature` restricts the set of interfaces according to a regular expression matching their full qualified signature names.
– The `annotated-by` predicate restricts the set of architectural artifacts already marked by the annotation given as a parameter of the predicate.

A filter expression can be a conjunction or a disjunction of predicates, or a complete HPath expression. Finally, an HPath expression is suffixed by an annotation name (with optional associated parameters), which will be attached to all artifacts captured by the pointcut.

Annotations Weaving Process. The HPath expressions are specified by the application developer as part of the textual specification of its applicative architecture. The weaving process (illustrated in Figure 1, `step (2)`) consists in interpreting these expressions and producing an annotated architecture where each architectural artifact matched by the queries are attached to the annotation(s) suffixed by the query. Table 1 illustrates some examples of HPath expressions. The right part of the figure describes the architectural artifacts matching each expressions when they are sequentially applied to the applicative architecture specified in Figure 2 from the top-level component `Root`.

Expression `(1)` weaves a *distribution concern* by annotating all the direct sub-components of `Root` with the `DistributedComponent` annotation. Expression `(2)` weaves a *threading concern* by annotating all components providing a

Table 1: HPath Expression Examples.

HPath expressions	Matching element(s) in Figure 2
(1) `$Root/child::* =>` **@DistributedComponent**`(...)`	Components {`Action`, `Reaction`}
(2) `$Root/descendant-or-self::*[ size(./interface::*[provided]) == 1 && ./interface::*[signature(Runnable)]) == 1] =>` **@ActivePeriodic**`( period="10ms", priority="8", maxiter"*")`	Components {`Writer`, `Reader`}
(3) `$Root/child::*/binding::*[ ./required::*[annotated-by(@Distributed)] && ./provided::*[annotated-by(@Distributed)]] =>` **@AsyncBinding**`()`	Binding between components `Action` and `Reaction`
(4) `$Root/child::Action:: descendant-or-self::* =>` **@Reconfigurable**`()`, **@LoggedCompAccess**`(hpcexp= "*;*;write")`	Components {`Writer`, `Data`, `Reader`, `Compute`}
(5) `$Root/descendant-or-self::Data =>` **@ProtectedComponent**`(maxval="1")`	Component {`Data`}

single server interface typed by the Java `Runnable` signature with the `ActivePeriodic` annotation. Expression `(3)` weaves a *concurrency concern* by annotating with the `AsynchronousBinding` annotation bindings specified between components themselves annotated by the distribution concern. Expression `(4)` weaves *reconfiguration and logging concerns* by annotating all the descendants of the `Action` component with the `Reconfigurable` and `LoggedCompAccess` annotations. Note that the latter takes a regular expression as parameter (`hpcexp` stands for *Hulotte pointcut expression*) whose syntax is explained in Section 4.2. Finally, expression `(5)` weaves the annotation `ProtectedComponent` in the component `Data`. It should be noticed that annotations can be also added manually by the application developer to annotate directly the artifacts of its architecture, without using HPath expressions. This capability can avoid the well-documented *pointcut fragility problem* of AOP [SK04], which stipulates that changes to the architecture might cause join points to incorrectly fall in or out of scope. In our case, an annotation can be attached directly to an artifact whose definition may evolve during the application design lifespan.

Finally, as defined by the HULOTTE process, the domain-specific annotations are implemented by the platform developer as described in the next section.

4 Hulotte Platform Level

One of the key motivations of HULOTTE is to provide an implementation of domain-specific concerns in an oblivious manner to the application developer. Separation between the business and domain-specific concerns is an essential software engineering principle to consider, in particular when an execution platform needs to be adapted to support heterogeneous execution contexts for its applicative components. Another motivation within our approach lies in the nature of domain-specific concerns, which can be arbitrarily complex, allowing the design, for instance, of both real-time and reconfigurable distributed component-based systems. Therefore, we propose a framework where the link between the business and domain-specific layers is not hard-coded, but expressed using aspect weaving.

4.1 A Unified Approach

The HULOTTE platform level is handled by the *platform developer*, who is responsible for implementing the execution platform supporting the domain-specific requirements specified by an annotated applicative architecture. As presented in Section 2, the HULOTTE platform is engineered with components. This approach brings two significant features: *i)* The platform developer benefits from an architecture-oriented design space to implement with fine-grained reusable components, the semantics of arbitrary complex domain-specific annotations. *ii)* Our approach is based on an isomorphic component model used at application level, as well as platform level, in a symmetric and unified way. The *container model* on which the platform is built is generalized, defining composition rules and architectural invariants as *architectural patterns* to specify the link between these two architectural levels. As an example, Figure 4 depicts the component-based container of the applicative component `Data` (specified in Figure 2). The internal structure of this container implements the semantics of the domain-specific annotations attached to `Data`—*i.e.*, `@Reconfigurable`, `@LoggedCompAccess`, and `@Protected component`—using HPATH expressions `(4)` and `(5)` given Table 1. Throughout this Section, we rely on this Figure, which will therefore be detailed step by step.

Architectural Patterns. The container architecture introduced in Figure 4 is systematically structured according to two architectural patterns [LMP+09]. First, we define the `ChainComposite` pattern as a composite component, whose

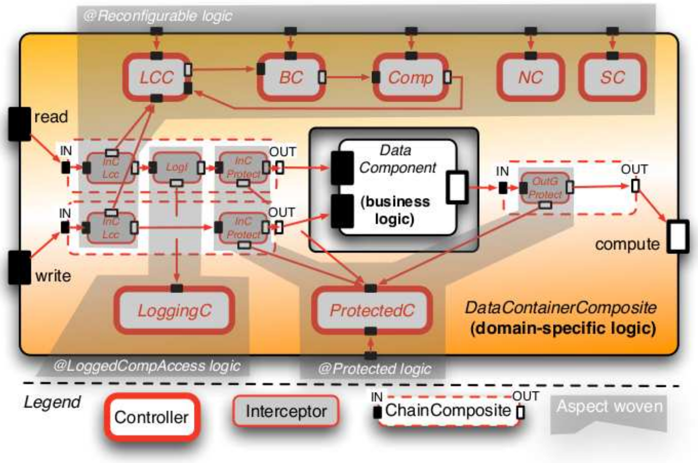


Figure 4: Component-based Architecture of a Container.

subcomponents are special components—*interceptors*. Within the `ChainComposite` pattern, the interceptor components are bound via their incoming and outgoing interfaces (respectively named `IN` and `OUT`, see Figure 4), thus implementing a *pipes and filters* architectural style.

Second, the `ContainerComposite` pattern—*i.e.*, the container itself—is also specified as a composite component. This pattern corresponds to the host infrastructure for the *platform components* implementing the domain-specific logic required by the applicative components. This pattern is systematically applied on each applicative component instances (either primitive, like for the `Data` component in Figure 4, or composite) as follows:

1. A set of *Controller* components[2] implementing various domain-specific services and meta-data influencing the whole component (or component set, *e.g.* life-cycle management, threading management) is composed in the container. In addition to that, controllers can provide interfaces to allow an access to these services from outside of the component.

2. For each interface of the applicative component, a `ChainComposite` pattern is injected, implementing an interception chain, which will be applied over incoming and outgoing invocations of the component. Interceptors and controllers are bound together via bindings (so called a *trap binding* from a client

[2] It should be noticed that we use the term *controller* to qualify services implemented by the platform implemented by aspects, but controller, interceptor and applicative components are isomorphic.

trap interface at interceptor side), thus allowing a centralized management of strategies for the interception mechanisms.

The container composite pattern is used in the same way when applied to *applicative bindings*, but without managing a business logic (since at application level, a binding is an abstract communication channel). In this case, incoming and outgoing `ChainComposites` implement the logic of *stub* and *skeleton* proxies deployed at client and server sides, respectively.

The Hulotte containers bring several advantages for engineering the platform: *i)* They allow the *inversion of control* over the applicative components and provide new interfaces to the environment to introspect/control them. *ii)* They provide a full separation of concerns between applicative and platform components. Moreover, the latter are encapsulated within a composite, acting as a host structure for domain-specific concerns and limiting their scope of action to the nested applicative components. Finally, *iii)* containers provide a structure for which interceptor chains and platform components are fully specified as component assemblies. Based on these Hulotte platform concepts, the following sections detail how the platform developer specifies domain-specific concerns as aspects woven on applicative architectures.

4.2 Aspect Weaving at the Platform Level

Within our approach, implementations of domain-specific concerns are architecture-oriented. As illustrated in Figure 1, each domain-specific annotations are implemented as component architectures by the platform developer. These *platform architectures* are designed as aspects: they are implemented independently for the base applicative architecture on which they will be woven, and independently from each other in a modular way. The `ContainerComposite` pattern, deployed around each applicative components and applicative bindings, is the host infrastructure for weaving these aspects. Each aspect is designed as assemblies of interceptor and controller components. An advice therefore corresponds to interceptors, which will be woven around applicative components (according to the `ChainComposite` pattern composition rules) and bound to controllers via *trap bindings*. The platform developer implements aspects using the Hulotte architectural concepts and should then specify how they will be woven using the join point model.

Join Point Model. At this abstraction level, the join points used by the platform developer are the business interfaces of the applicative components. Indeed, the base language on which our aspect framework relies is an applicative architecture, abstracting the details of the underlying programming language used to implement the components. Therefore, only the incoming or outgoing operation calls crossing the component interfaces are relevant for advice specifications.

Pointcut Expression. A HULOTTE *pointcut expression* (`hpcexp`) is divided into two parts[3]. First, a keyword that specifies if the incoming calls (keyword `SERVER`) or outgoing calls (keyword `CLIENT`) or both of them (no keyword) must be selected. This compares to `before`, `after`, and `around` advices in ASPECTJ terminology, but for a join point expressed as a component. Second, three regular expressions separated by semicolons specifies which interface signature, name, and operations must be matched, respectively (they rely on the *java.util.regexp* package). These expressions are used by the platform developer to specify how interceptor components should be woven around the business interfaces of applicative components, at the granularity of an interface and/or an operation call.

To illustrate these points, the container infrastructure depicted in Figure 4 is obtained by weaving three aspects around the `Data` applicative component, corresponding to the implementations of the domain-specific annotations `@Reconfigurable`, `@LoggedCompAccess` and `@ProtectedComponent` (the resulting aspects woven are represented by shadowed areas in the Figure). We rely on this example to explain precisely how these aspects are specified by the platform developer and then woven by the HULOTTE platform weaving process. Figure 6 gives relevant excerpts of aspects implementing the above mentioned annotations using HULOTTE-ADL. Let's first focus on the `@ProtectedComponent` aspect (lines `5--32`). Basically, its semantics is close to a semaphore used in concurrent programming: only a maximum number of parallel executing threads (`maxval`) are allowed to execute operations provided by the component; if the maximum is reached, new incoming calls are then queued. The two interceptor definitions for this aspect begin lines `9--10` and `20--21`. Note that in this example, annotations are used to mark architectural artifacts with platform-specific concerns, but directly within the ADL declarations and not via HPATH expressions, as it was mentioned earlier. For instance, the `@Interceptor` annotation takes a pointcut expression as a first argument: line `9` specifies that component `InProtect` is an interceptor that should be woven on each client interface whatever their signatures/names/operations are (`hpcexp="CLIENT *;*;*"`). `InProtect` extends `interceptorType` specified in lines `1--4` (the signatures of the interfaces `IN` and `OUT` are variables known only at weaving time according to the advice business interface).

Advice Implementation. HULOTTE relies on generative techniques to create the implementation of interceptor components. The basic way provided to

[3] `(CLIENT|SERVER|) [str reg exp] ; [str reg exp] ; [str reg exp]`

the platform developer is to declaratively express within the aspect specification how the operation calls to the *trap interfaces* are performed according to the advice business calls. These requirements are classical pointcut-advice mechanisms around operation invocations and are specified by an annotation (`WeaveTrapOperation`). It specifies which operation of the trap interface should be invoked and when—*i.e.*, `before` or `after` the advice call. The signature of the trap interface can syntactically reflect the platform services provided by the controller component bound to it. This feature strengthens the symmetry between platform and applicative layers since controllers can be handled as regular components without being implemented knowing they will be used as aspects. Otherwise, our approach supports also the case where the trap interface is based on an operation used to pass the reified original business invocation as a parameter. Finally, more advanced generative tools are also provided to the platform developer for generating arbitrary complex interceptor implementations independently from the intercepted business interface signatures. However, describing these tools exhaustively is out of the scope of this article.

To illustrate these features, let us consider again the *ProtectedComponent aspect* woven around the component `Data`. It defines two interceptors for the following reason: each applicative component's incoming call should be trapped in order to increment the semaphore counter before calling the component `Data` and to decrement it after `Data` returns. However, the reverse policy is required for outgoing calls from `Data`.

According to that, lines `11--18` express that the weaving task should generate the content of the `InCProtect` interceptor via the *trap interface* `sem` with a call to the method `acquire` before the proceed and a call to `release` after (the proceed denoting the business method call executed on the `Data` component). The reverse policy is implemented by the `OutGProtect` interceptor. The result of the weaving task is illustrated in Figure 4. The sequence diagram given in Figure 5 represents the simplified behavior of the resulting container according to an incoming invocation from its `read` interface. Notably, it clearly depicts the message sequences implementing the logic of the *ProtectedComponent aspect*.

Aspects Weaving Process. For each applicative component, the HULOTTE platform weaver first generates a *composite container* used as a host infrastructure to weave the aspects implementing the domain-specific annotations: each applicative instance is nested within a container and empty *chainComposite* components are instantiated around their business interfaces (note that this latter pattern is not explicitly handled by the platform developer within its aspect specifications). This process is illustrated in Figure 1 (`step (3)`). Then, for each aspect specification, the weaving process (Figure 1, `step 4`) consists in generating interceptors, adding them within `ChainComposites` according to their

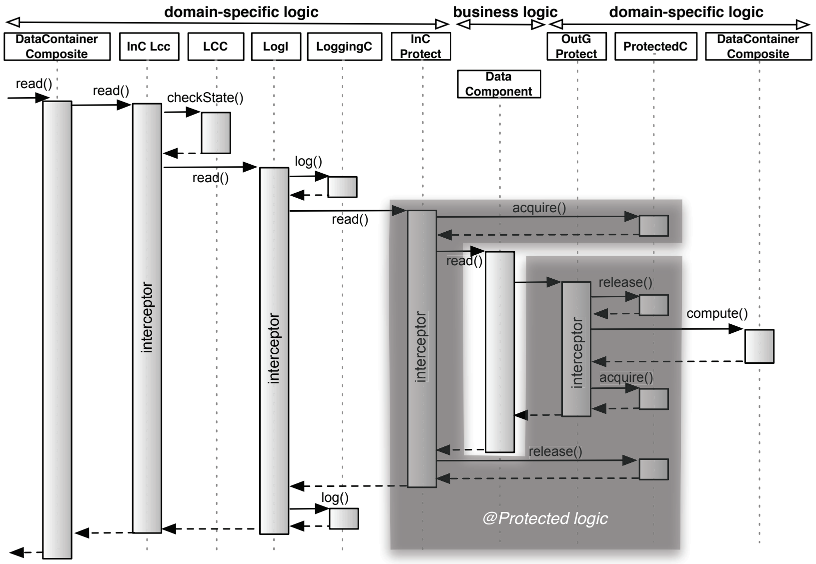


Figure 5: Sequence Diagram of an Incoming Call (from the `read` interface) within the container given Figure 4.

pointcut expressions, adding controller components to the container, and finally setting their bindings. Aspects are simply stored within a component library and identified by the platform developer by the `@CompositeContainerAspect` annotation (*e.g.*, lines `5`, `33`, and `47`). The annotation parameters filled in by the application developer are propagated to aspect specifications by the mean of template parameters at ADL level. For instance, the parameter `maxval` filled in Table 1(5) is propagated to its aspect specification as an attribute value of the `ProtectedC` controller line `26` (while this template parameter is defined line `7`).

There are several benefits when weaving component-based aspects on applicative components to modularize domain-specific concerns. First, aspects can be woven according to the incoming and outgoing interaction points externalized by the base components, through explicitly defined and stable interfaces. Second, the weaving process does not require the source code of the base component, and aspects do not have dependencies on internal elements of the base implementation. Moreover, the internal behavior of the base components is not altered by the weaving process. Finally, dependencies between aspects and base components can be explicitly captured at architectural level, as detailed in the following section.

```
component interceptorType {
  provides ${derived:BOUND-ITF-SIGN} as IN
  requires ${derived:BOUND-ITF-SIGN} as OUT
}
```

```
@ContainerCompositeAspect
  (implements="@ProtectedComponent")
component ProtectedAspect arguments (maxval) {
  provides externPItf as extP
  @Interceptor(hpcexp="CLIENT *;*;*", order="LIN-FOUT")
  component InCProtect extends interceptorType {
    @TrapInterface
    @WeaveTrapOperation(trapweaveexp="acquire",
      advice="before")
    @WeaveTrapOperation(trapweaveexp="release",
      advice="after")
    requires semaphoreItf as sem
    @GenerateBasicInterceptor
    content InComingProtectImpl
  }
  @Interceptor(hpcexp="SERVER *;*;*", order="LIN-FOUT")
  component OutGProtect extends interceptorType {...}
  // protected-controller instance
  component ProtectedC {
    provides semaphoreItf as sem
    provides externPItf as extP
    attributes int initval = ${maxval}
    content ProtectedCImpl
  }
  binds this.extP to ProtectedC.extP
  binds InProtect.sem to ProtectedC.sem
  binds OutProtect.sem to ProtectedC.sem
}
```

```
@ContainerCompositeAspect
  (implements="@LoggedCompAccess")
component LoggedCompAccessAspect arguments(hpcexp) {

  @Singleton
  component LoggingC {...}
  @Interceptor(hpcexp=${hpcexp}, order="ANY")
  component LogI extends interceptorType {
    @TrapInterface
    requires AOPAllianceItf as aopitf
    content LoggingCImpl
  }
  binds LoggingIntercept.aopitf to LoggingC.aopitf
}
```

```
@ContainerCompositeAspect (implements="@Reconfigurable")
component ReconfigurableAspect {
  // life-cycle-controller instance
  component LCC {...}
  // BC, Comp, NC, SC controllers,
  // interceptors, bindings
}
```

```
@ContainerCompositeAspect (implements="@AsyncBinding")
component AsyncBindingAspect {
  @Interceptor (hpcexp="CLIENT *;*;*:void"
    order="LIN-FOUT")
  component stub {...}
  @Interceptor (hpcexp="SERVER *;*;*:void"
    order="LIN-FOUT")
  component skeleton {...}
}
```

Figure 6: Aspects Excerpts Implementing the Domain-Specific Logic of annotations `@ProtectedComponent`, `@LoggedCompAccess`, `@Reconfigurable` and `@AsyncBinding` using HULOTTE-ADL.

4.3 Handling Dependencies between Aspects

AOP is a programming paradigm that increases modularity by allowing the separation of crosscutting concerns. However, a major well-known difficulty appears at weaving time, since a composition of aspects can result in an inconsistent system. Indeed, aspects may be incompatible or may be dependent on each others. In this section, we present how these issues are handled at the application and platform levels of the HULOTTE framework (all these mechanisms have been used within the two case-studies presented in Sections 6.1 and 6.2).

• Within our framework, mutual exclusion between domain-specific concerns is ensured at applicative level by means of OCL constraints over instances of the annotated applicative architectures (*i.e*, just after the weaving `step 2` represented Figure 1), and checked automatically to guarantee that they are verified before the platform weaving process. Moreover, we provide also the capability to express constraints that implementations must fulfill at source code level before weaving. These points are reported in details in [NL09].

• In our case, ordering of a stack of advices on a particular join point corresponds to the execution order of interceptor components within the `ChainComposite` patterns. Far from resolving this problem generally, we however provide to the platform developer a way to specify basic ordering strategies over advice weaving. Within the scope of a single aspect specification, relative orders can be specified over several advice definitions. Moreover, two kinds of absolute orders (meaning that they should be respected whatever other aspects are woven by the weaving process) can be specified: `FIN-LOUT` or `LIN-FOUT`. The former corresponds to an interceptor placed at the first position of a `ChainComposite` for an incoming base interface and/or at the last position for an outgoing base interface. The latter refers to the reverse policy. Examples of absolute orders are given in Figure 6, lines `9`, `20`, and `39`.

• In the case where platform components specified in different aspects need to collaborate, we provide a feature to the platform developer called *late-binding*. Such a binding is processed at the end of the container weaving process. It consists in injecting dependencies between controllers when the source or target controller has been added within the container by an another aspect. For example, a relevant use-case of such a feature concerns a dependency between an applicative component annotated with `@ActivePeriodic` and `@Reconfigurable` annotations. Indeed, the former is implemented by a controller that periodically executes the operation `run` of the interface `java.lang.Runnable` provided by the applicative component. If the *Reconfigurable aspect* is also woven in this component, the life-cycle controller it implements should be synchronized with the periodic controller—*i.e.*, the periodic activation is started/stopped when the component is started/stopped from its life-cycle controller. This requirement is depicted in Figure 7. At the end of the container weaving process, if these two

Reconfigurable and *ActivePeriodic aspects* have been woven, this late-binding specification consists in adding a client interface typed by the `PeriodicItf` signature to the life-cycle controller (named `LCC`), then in weaving an invocation to the operation `startPeriodicAct` at the end of the implementation of `startComp` provided by the life-cycle controller, and finally in binding the two controllers.

```
@ContainerCompositeAspect (implements="@ActivePeriodic")
component ActivePeriodic {
  component PeriodicC { provides PeriodicItf as periodicI}
  @LateBinding (controllerNameDependency="LCC",
    kind="export", trapweaveexp="startPeriodicAct",
    hpcexp="SERVER *;*;startComp:void", advice="after")
  @LateBinding (controllerNameDependency="LCC",
    kind="export", trapweaveexp="stopPeriodicAct",
    hpcexp="SERVER *;*;stopComp:void", advice="after")
  binds ${derived:LATE-BINDING} to PeriodicC.periodicI
}
```

Figure 7: `@ActivePeriodic` Aspect, Excerpt Focused on Late-Binding.

• Finally, a relation between an applicative component and controller components is not always bijective. In some case, controller instances are globally shared by the whole application containers. To handle this requirement, our component model supports *component sharing*, meaning that a single instance of a component can be encapsulated by several composites. At design time, this feature is specified using the `@Singleton` annotation as it is the case for the `LoggingC` controller (cf. Figure 6, line 37).

5 Implementation

This section gives an insight of the basic parts of the HULOTTE framework implementation. It consists of three main units: *i)* The *front-end* processing the IDL files[4] and the description of an applicative architecture and its corresponding HPath expressions stored in ADL files, *ii)* the *middle-end* implementing the two-level weaving engines presented in Sections 3 and 4, and *iii)* the *back-end* generating the code of the final execution infrastructure, which is afterwards compiled by a classical Java compiler, as illustrated in Figure 1(5).

The framework relies on the *Eclipse Modeling Framework* (EMF) technology [BSE+04]. EMF has been used to implement the HULOTTE model introduced in Section 2.2. The *front-end* is then in charge of parsing the IDL and ADL files and of instantiating the corresponding EMF instances. It also instantiates an EMF model of the primitive component implementations. The *implementation*

[4] `.java` files in the context of this article.

model corresponds to the Java AST defined by SpoonEMF[5] and based on the Spoon [PNP06] program transformation framework. The *middle-end* relies on these two models (*i.e.* architecture and implementation models) to implement the Hulotte weaving engines, and more specifically, the following points:

- It implements the constraints checker mentioned in Section 4.3 and based on OCL rules specified over the EMF instances. Implementation details about this feature is given in [NL09].
- It implements the late-bindings presented in Section 4.3. Indeed, a late-binding consists in weaving operation calls in controller implementations. This feature relies on the Spoon API to inject the required calls into implementation models of the controllers. The implementation models are then pretty printed in order to obtain the transformed Java code.
- It generates the Java code of interceptor components. The generation process is based on code templates taking three parameters as arguments: *i)* the signature of the intercepted applicative interface, *ii)* the *pointcut expression* (`hpcexp`) attached to the interceptor definition, and *iii)* the list of `@WeaveTrapOperation` required to generate trap calls. We also mentioned more advanced tools providing by our framework for generating arbitrary complex component implementations. They are defined as Hulotte plugins taking as input introspectable interface definitions and which output Java code according to them. The Java code produced by these plugins are implemented by the platform developer, based on the Spoon API.
- And last but not least, it notifies the *back-end* to apply optional optimizations on the executable, which can be configured by the end-user.

According to the last point, the back-end generates by default an infrastructure for which dependency injections between components are handled by dedicated component factories, themselves generated by the back-end. This mechanism relies on proxy objects interposed on component interfaces, and is the basic feature required to provide reflective and reconfigurable component-based systems. However, it can notoriously impact on the performances—in terms of memory footprint and execution time—of the deployed executable, all the more as the platform is implemented by component architectures. Within the back-end, we have introduced optimization heuristics in order to mitigate these overheads. The heuristics focus on reducing interceptions in inter-component communications and on merging implementations of architectural artifacts. The merge algorithm consists in inlining, for instance, interceptors, controllers, and applicative component implementations in a single Java class. A detailed description of the heuristics provided by our framework is out of the scope of this

[5] Available from `http://tinyurl.com/spoon-emf09`

article, we refer the interested reader to [PLMS08]. These optimization features can be specified as HULOTTE's annotations by the application developer.

Finally, HULOTTE is coupled to basic editing tools provided by the EMF framework. These tools can be used by the application developer to visualize the annotated architecture resulting from the annotations weaving process (*i.e.*, after the weaving step illustrated in Figure `1(2)`). It allows her/him to check if the pointcuts captured by the queries have correctly matched the architectural artifacts, and eventually to debug them. Since the same language (HULOTTE-ADL) is used at platform level, the final architecture can also be edited by these tools (*i.e.*, after the platform weaving process, Figure `1(4)`), to verify if aspects have been correctly woven into each containers.

6 Evaluation

In this section, we evaluate our approach on two case studies. The first is related to real-time and embedded applications, while the second refers to the context-aware middleware domain.

6.1 A Framework for Real-Time Java based Systems

Application-level Overview. HULOTTE has been experimented in the design of a component-based framework for RTSJ-based real-time and embedded systems [PLMS08]. The *Real-Time Specification for Java* (RTSJ) [BGB+00] is a specification for development of predictable real-time Java-based applications. Among many constructs, which mainly pose special requirements on the underlying JVM, two new programming concepts were introduced: *i)* real-time threads (`RealTimeThread`, `NoHeapRealTimeThread`) that have precise scheduling semantics, and *ii)* special types of memory areas (`ScopedMemory`, `ImmortalMemory`), which are outside the scope of action of the garbage collector to ensure predictable memory access among the objects where they are allocated. RTSJ introduces a non-intuitive and difficult-to-take-in-hand programming model and imposes several rules on the software composition process.

Table 2 sums up the main architectural annotations provided to the application developer to mark its applicative architecture. They provide the ability to specify the multi-task and concurrent nature of the application according to RTSJ concepts. An *active component* represents various execution concepts enforced by RTSJ—*non real-time*, *real-time*, and *non-heap real-time*. The semantics of such a component is the one of a monitor controlling the execution of the business operations it provides. It is attached to its own thread of control (or pool of threads), activated periodically or sporadically—*i.e.*, triggered by incoming invocations. `@MemoryArea` expresses the allocation contexts of the components—*heap*, *scoped*, and *immortal memory*.

Table 2: RTSJ-Specific HULOTTE Annotations.

Annotation	Parameters
`@ActivePeriodic`	`threadkind`, `period`, `priority`, `maxiter`
`@ActiveSporadic`	`threadkind`, `priority`, `threadpoolsize`
`@MemoryArea`	`memkind`, `memsize`
`@AsyncBinding`	–
`@ProtectedComponent`	`maxval`

Platform-level Overview. Each annotation is implemented by aspects as described in Section 4.2. The controller implementations rely on the RTSJ API and its library (*Priority Scheduling*, *High-resolution Timers*, *Wait-Free Queues*, and *Memory Contexts*). An aspect has been also implemented to support cross-scope communication, since RTSJ imposes the use of code patterns [PFHV04] to pass values back and forth across the boundaries formed by memory areas. This aspect consists of interceptor components, generated according to the interfaces involved in a binding between components allocated in different areas.

Application Scenario. To apply our domain-specific framework, we have reengineered a large application: the *Real-Time Collision Detector* (RCD) presented in [KHP+09]. The RCD algorithm is about 2.3KLoc and its task is to proceed a periodic stream of aircraft positions and determine if any of these aircrafts are on a collision course. The original object-oriented implementation has been reengineered as a HULOTTE architecture composed of about ten applicative components. The main part of the algorithm is periodically executed by a non-heap real-time `ActivePeriodic` component and processed by components allocated in immortal and scoped areas according to its requirements. The environment is simulated by a non-real time `ActivePeriodic` component and interactions between real-time and non real-time parts of the application is ensured by a `ProtectedComponent`. The results from each algorithm iteration are asynchronously transferred to an `ActiveSporadic` component. The resulting platform produced by the HULOTTE weaving process for this application scenario is about 2.5K generated Loc.

6.2 SPACES: A RESTful Context Dissemination Framework

Application-level Overview. SPACES is a RESTful middleware solution for the flexible dissemination of context information. In particular, SPACES proposes to distribute context information in ubiquitous computing environments by combining the principles of *REpresentational State Transfer* (REST) and the *COntext entitieS coMpositiOn and Sharing* (COSMOS) context framework [CRS07, RCS08]. By combining COSMOS and REST, SPACES therefore in-tends to provide *Context as a Service* and to enable the efficient distribution of context information among heterogeneous devices.

Weaving Context Processing Concerns. In COSMOS, *context policies* are hierarchically decomposed into fine-grained units called *context nodes*. A context node refers to context information controlled by a software component. A Context policy refers to a hierarchical composition of context nodes reflecting the inference of a specific context information. Context nodes leaves (the bottom-most elements, with no descendants) encapsulate raw context data obtained from collectors, such as operating system probes, sensors near the device, and user preferences in profiles. Intermediate context nodes are context operators used to process the context information collected from the lower layers in order to compute an high-level context information. The efficiency of the context policy processing can be improved by tuning the following context node properties. **Active/Passive:** An active node is associated with a thread of control, while a passive node obtains context information upon demand. Typical examples of active nodes include a node in charge of the centralization of several types of context information, a node responsible for the periodic computation of higher-level context information, and a node to provide the latter information to upper nodes. **Observation/Notification:** Communication into a context node's hierarchy can be top-down or bottom-up. The former—implemented by the interface `Pull`—corresponds to observations that a parent node triggers. The latter—realized by the interface `Push`—corresponds to notifications that context nodes send to their parents. **Pass-through/Blocking:** Pass-through nodes propagate observations and notifications while blocking nodes stop the traversal. For observations, COSMOS transmits the most up-to-date context information without polling child nodes. For notifications, COSMOS uses context data to update the node's state, but it does not notify parent nodes.

Table 3: Context Processing HULOTTE Annotations.

Annotation	**Parameter**
`@Active`	`period`: Integer
`@Notification`	`policy`: {`sequential` or `parallel`}
`@Blocking`	`static`: {`true` or `false`}

Table 3 summarizes the HULOTTE annotations associated to the COSMOS parameters. By default, context nodes are configured as passive, observable, and pass-through. The HULOTTE annotations are therefore used to refine the configuration of the marked context nodes.

Weaving Context Distribution Concerns. *REpresentational State Transfer* (REST) is a resource-oriented software architecture style for building Internet-scale distributed applications [Fie00]. Typically, the REST triangle defines the principles for encoding (*content types*), addressing (*nouns*), and accessing (*verbs*) a collection of *resources* using Internet standards. Resources, which are central to REST, are *uniquely addressable* using a universal syntax (*e.g.*, a URL in HTTP) and share a *uniform interface* for the transfer of application states between client and server (*e.g.*, `GET/POST/PUT/DELETE` in HTTP). REST resources may typically exhibit multiple typed representations using—for example—XML, JSON, YAML, or plain text documents. Thus, RESTful systems are loosely-coupled systems following these principles to exchange application states as resource representations. This kind of stateless interactions is particularly interesting in the context of SPACES since it improves the resources consumption and the scalability of the system.

Table 4: Context Distribution Hulotte Annotations.

Annotation	Parameter
`@Host`	`identifier`: Uniform Resource Identifier
`@Provider`	`type`: MIME Type[]

Table 4 summarizes the Hulotte annotations associated to the REST parameters. The Hulotte annotations are used to describe the distribution of context nodes among the physical devices considered in the environment. In SPACES, REST contextual resources are described by the following parameters: **Host** points to a physical device described using the *Uniform Resource Identifier* (URI) format (*e.g.*, `http://device.inria.fr:8080`). Therefore, context identifiers include a communication scheme, a server address, a context path, and a sequence of request parameters: `scheme://context-server/context-path?request-parameters`. **Provider** refers to the different representations, designed by their MIME media types classification [IAN07], under which a REST resource can be retrieved. In particular, SPACES promotes the *Java object serialization* as the default resource representation (`application/octet-stream`) for performance concerns. Nevertheless, SPACES provides also representations of context resources as XML (`application/xml`) and JSON (`application/json`) documents.

Platform-level Overview. The platform level of SPACES is implemented using the Scala programming language [OSV08]. Specifically, the mechanism of *trait* provided by Scala enables SPACES to support a modular implementation of the container controllers and interceptors. Therefore, the context processing and distribution concerns are implemented as specific traits, which are combined in order to implement SPACES controllers and interceptors.

In this context, HULOTTE serves as a weaving framework that selects the SPACES traits according to the annotated applicative architecture describing the context policy and mixes them into the component containers according to a particular weaving strategy (*e.g.*, optimization level). Figure 8 depicts an example of component-based container hosting a context node and generated by the HULOTTE framework from the annotated description of a SPACES context policy. For each context node, 2.3KLoc are generated by the weaving process, and the whole code imported via singleton components (implemented within a library and used by SPACES containers) is about 9KLoc.

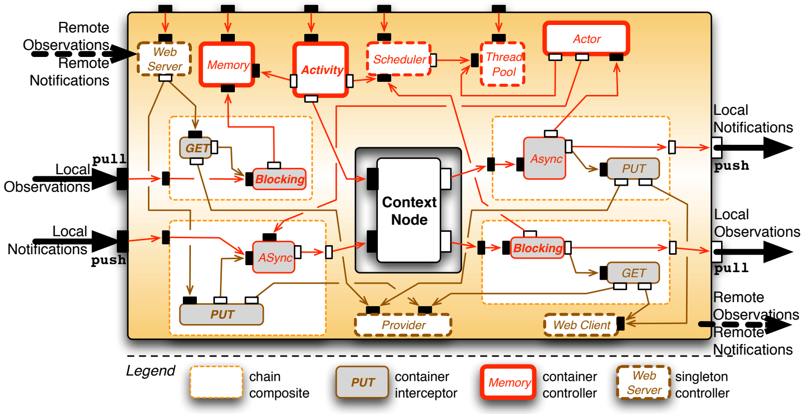


Figure 8: Component-based Architecture of a SPACES Generated Container.

6.3 Evaluation Synthesis, Discussions and Lessons Learned

When developing the two case studies, we witnessed many benefits of our approach that we present in the following, and discuss their limitations before a comparison with existing ones in Section 7.

Relying on Concise Annotations for Specifying Domain-Specific Concerns. HULOTTE allows us to define domain-specific component-based frameworks addressing the challenges of RTSJ-based software development and these of a RESTful context middleware for which dedicated execution and communication models are supported. Therefore, the way annotations are used at application level is not a curb on expression complex domain-specific concerns.

The features provided by HPath make use of convenient queries to easily adapt annotations (and their parameters), which will be woven. HPath provides a rich expression power, the variety of navigation axis it offers allows to capture precise pointcuts over the architecture artifacts. Even if the examples of queries presented in Table 1 might appear complex to the reader, the fact remains that the use of HPath is straightforward since it is based on the component model's artefacts to navigate through the applicative architecture. However, we can mention a drawback arising in using HPath. In minor cases, the use of queries can turn out to be more verbose than annotations manually set into the architecture. This is typically the case when the same annotation should be woven with different parameters, which must be specified by multiple queries.

Finally, if the semantics attached to the annotations is well documented, we believe that the learning curve for application developers is equivalent when using our approach compared to the one required to take domain-specific component models in hand.

Improving Separation of Concerns. The HPath queries allow to specify the annotations in a completely uncoupled way from the architecture's specifications, without polluting the latter. Moreover the queries can be expressed in a centralized way (*e.g.* at the end of the top-level component's ADL file).

In the general case, relying on annotations enforces separation of concerns between the business and domain-specific logics, the latter being externalized at the architectural level with well-defined semantics, emphasizing on the problem domain. Moreover, this feature allows the support of tools reasoning on high-level perspectives of the application, rather than source code implementations, and conducting domain-specific analysis of the system. For instance, we proposed a validation framework [Pls09] to check the compliance with RTSJ based on annotated architectures.

This separation of concerns occurs also at code level since domain-specific code is implemented by the Hulotte platform, rather than buried and tangled within the business classes, and the composition process of platform components is established without any dependency on the internal elements of the applicative code. As a consequence, the business code implementing the components becomes more readable and maintainable—reflecting the functional needs of the application without any constraints imposed for instance, by RTSJ or RESTful context middleware. Moreover, the domain-specific code handled by the platform alleviates drastically the burden on the application developer, since from one application to another, implementing these concerns is a redundant, time-consuming, and error-prone task.

Exploiting CBSE at Platform Level. When considering the experiments we have conducted using Hulotte, the large majority of the domain-specific annotations are implemented as component architectures, which validate our proposal of using the component paradigm at platform level. However, this design choice has a limitation, in particular considering these two following issues: *i)* At a stage within the HULOTTE compilation flow, architecture transformations are required to implement the distribution concern, since a standalone executable is generated on each physical distributed node. *ii)* As presented in Section 6.1, a memory allocation concern has been considered for applicative components. A minor part of its implementation impacts on a process of HULOTTE: the generation of the component factories in order to control the instantiation of these components. These two points can not be implemented as component architectures woven into containers but are instead implemented using specific extension points defined within the HULOTTE compilation flow. However, their descriptions are out of the scope of this article.

We can also highlight the benefits raised by our approach from the aspect paradigm point of view. First, the platform components are exclusively woven according to the well defined interfaces of the applicative components. It simplifies greatly the implementation of advices, compared to classical code-centric AOP approaches, the latter relying on many low-level constructs from the base programming language on which aspects are woven (*e.g.* method declarations and calls, field access, etc.). Moreover, composition between platform and applicative components does not distort the applicative code, since the latter remains totally unchanged after the weaving process.

Handling Dependencies between Aspects. Our framework provides the *Object Constraint Language* (OCL) support used by the platform developer *i.e.*, the domain expert, to express the constraints between annotations [NL09]. These constraints are automatically checked at application level once the annotations weaving process is completed. This is of great help for the application developer, giving him a feedback on using annotations in a consistent way. We have successfully applied the OCL support on several constraints imposed by annotations of both case-studies.

The notion of *component sharing* introduced in Section 4.3—widely used for designing the RESTful context middleware—allows to preserve a strong encapsulation of the components nested within composite containers, while sharing between them built-in middleware services. The notion of *late binding* was also used within the two case-studies according to the use-case presented in Section 4.3 (for synchronizing life-cycle controllers of *active components*). Even if late bindings are quite complex to define, nevertheless, we believe that they provide an interesting feature for defining component interactions without any code modification for the developer.

Towards more Flexibility and Reuse. The use of concise annotations at architectural level introduces flexibility and configurability in the application design process with regard to domain-specific concerns. First, annotations parameters ease the configuration of platform-level components. For instance, the developer can simply specify and change a priority, a period of a thread managed by the platform, or the size of an RTSJ memory area thanks to parameters attached to their annotations. The propagation of these information from application-level annotations to platform-level aspect implementations is automatically handled by the platform weaving process by mean of ADL template parameters (see Section 4.2, page 16). Second, the strong decoupling between annotations and architectural artefacts provides a straightforward way for adapting the execution contexts of applicative components. For instance, according to the two presented case-studies, we are able to easily change the internal concurrency of the *Real-Time Collision Detector* (by changing the way components are executed and synchronized), to change the component allocations within different memory areas, or to adapt the deployment policy of a context node. When using HPath, all these platform adaptations are conducted by the developer by simple modifications of the queries centralized in Hulotte-Adl files.

Our approach improves reuse of platform components since they are specified as regular components without being implemented knowing they will be used as aspects. It was typically the case of the singleton components used to implement SPACES containers. Indeed, more than 9KLines of componentized code were incorporated by the Hulotte process imported from component libraries provided outside the scope of our case-study. Interceptor components are also fully reusable units from one aspect to another since they are implemented (and generated) independently of applicative component specifications on which they will be woven.

We have also conducted an experiment related to the *reconfigurable aspect* presented in Section 4. It was initially implemented to provide full architecture introspection and reconfiguration capabilities at runtime and was successfully applied in the context of our two case-studies. Moreover, from its initial implementation, we have derived several aspects, each of them providing a subset of these capabilities, *e.g.*, for providing only minimal introspection features when reconfiguration is not required at runtime. This experiment has highlighted the benefits brought by the use of architectures within aspect implementations where fine-grained platform components are easily reusable as advices. Finally, we are currently implementing a connection between the RTSJ concerns and the RESTful context concerns. We will therefore provide an RTSJ-based implementation of SPACES to ensure predictability over context node executions, reusing an already implemented Hulotte distributed aspect [MPL+08] based on Real-Time CORBA.

Performance Issues. Finally, in [PLMS08], we have performed a quantitative evaluation demonstrating the efficiency of the optimizations presented in Section 5 over component-based containers. Indeed, the impact over the performances are null when platform component implementations are merged within the business code, compared to a reference application where domain-specific concerns are implemented by the application developer. These results show that the weaving process promoted in this article, based on high-level abstractions, does not impact on the performances when component reification at runtime is not required.

7 Related work

Specializing Architectural Artifacts with Annotations. In programming languages, the use of annotations is widely applied to specialize their basic constructs. However, to the best of our knowledge, only the Think ADL [LNB09] and UML2 [OMG07] exploits this feature to specialize architectural constructs. Within Think ADL, a set of *flexibility-oriented* annotations are provided to the application developer allowing to generate, for the same architecture, a set of systems with different flexibility capabilities. Furthermore, Think shares with our approach the capability to express these annotations in a uncoupled way from the base architecture using AOP techniques [LNB09]. However, these annotations are limited to expression of flexibility degrees required by components at runtime and do not address other extra-functional concerns. The pointcut language provided for annotations weaving is based on pattern-matching rules over architectural artifacts names, but it does not allow to capture more precisely the base artifacts as it is the case with HPath queries. In turn, UML2 defines the *composite structure diagram* for specifying software architectures, and introduces the notion of profiles [FV04]. The latter is the built-in lightweight mechanism that serves to customize UML artifacts for a specific domain or purpose via *stereotypes*. Thus, the latter could be used to extend the semantics of the composite structure diagram artifacts. Our approach share with UML the notion of annotation, close to the one of a stereotype.

Aspect Weaving for EJB Component. JBoss AOP[6] and Spring AOP[7] (and JAsCo [SVJ03] based on Java Beans) are Java frameworks for AOP. When applied to application servers, aspects are woven to EJB components. We can put forward two main differences with Hulotte. First, EJB's are basically deployment units and not composition units which reify their dependencies allowing to specify the application as an architecture. Therefore, advices can not be specified according to high-level architectural constructs, and the base language remains code-centered. Second, aspects are implemented as plain Java classes whereas Hulotte implements them as an architecture.

Symmetric Approaches: Aspects as Components. FuseJ [SFV06], CAM-DAOP [PFT05], and FAC [PSCD06], are approaches sharing the goal of

[6] JBossAOP: `http://jboss.org/jbossaop`
[7] Spring AOP: `http://www.springsource.org`

reifying aspects as first class entities in the component based programming model. The first defines its own ADL by introducing the concepts of *gate* and *connector*, while the second and the third identify the concepts of *component* and *aspect*. In particular, FAC is based on FRACTAL [BCL+06], which has itself inspired HULOTTE. A distinction is made in these three approaches between classical component and aspect-oriented interactions. The former is similar to the notion of *applicative binding* and the latter allows defining around advices (so called *aspect bindings* in FAC). Aspect-oriented interactions, intercepted from operations externalized at component level, allow—by means of *Plain Old Java Objects* (POJO) interceptors—the delegation of a behavior implemented itself by a component, realizing this symmetric aspect/component architecture. However, in both approaches, the weaving process results in a flatten architecture where applicative and aspect components are composed at the same level, whereas HULOTTE exploits the notion of container. The same distinction with our proposal can be made with the AO component and composition model defined by the AOSD-Europe project [Pro08]. Moreover, compared to these three approaches, HULOTTE adds the capability to advice applicative bindings, a required feature to support implementations of specific communication schemes between components. Furthermore, HULOTTE provides a deeper insight of the aspect weaving impact by reflecting the woven advices within the component containers as domain-specific components.

Extensible Container-Based Approaches. Even if component containers are a key part of mainstream component frameworks such as EJB, they support a predefined and non extensible set of extra-functional services. On the contrary, the PIN component model [Mor06] is based on generative techniques to produce custom containers encompassing component interactions (stubs and skeletons generation) and implementing extra-functional concerns. This approach shares also with HULOTTE a strong separation of concerns between the containers logic and those of applicative components. Despite of these similarities, the work presented in this article differs from theirs in that they propose the use of ASPECTC++ and template meta-programming for generating containers, the weaving process is then code-oriented whereas our solution abstracts it at architectural level. In this aspect, ASBACO [MB05] and AOKELL [SPDC06] are aligned with the approach in this article because both rely on containers engineered as a component-based architectures. However, with ASBACO, integrating platform and applicative components is performed with load-time mixin technique based on a bytecode engineering library, resulting in an infrastructure where both levels are tangled, contrary to our approach. Moreover, we believe that, with this technique, it becomes quite difficult to ensure traceability—required for evolution purposes—between the final infrastructure built by the weaving process and the starting applicative and platform architectures. Indeed, in our approach, the final infrastructure is fully specified as an architecture thanks to the use of architectural patterns. On the contrary, AoKell preserves the platform architecture but suffers from the same drawback since the integration between the two levels is based on AspectJ. Finally, these two last approaches do not rely on aspect weaving within their containers for integrating extra-functional concerns implemented in a modular way and independently from each other.

Aspect-Oriented Architecture Description Languages. AO-ADL [PF07], AspectualACME [GCB^{+}06], or AspectLEDA [MTM09] generally propose to integrate aspects as first class entities in legacy ADLs. The objective of these extensions is to design and reflect crosscutting concerns as *aspect connectors* (or *aspect coordinators*) within the software architectures. These connectors wire business artefacts and intercept the communications in order to inject the crosscutting artefacts. Unlike these approaches, the integration of Hulotte differs with respect to several issues. First of all, Hulotte adopts a two-steps approach in order to weave crosscutting concerns, where domain-specific annotations are first attached to the architecture artefacts. The use of domain-specific annotations supports both the interception and the injection of crosscutting concerns within business components. Thanks to these architectural annotations, conflicts can be detected by observing the annotations attached to the architecture. Next, the injection technique used by Hulotte consumes the architectural annotations and exploits code generation and optimization algorithms in order to reduce the run-time overhead. In particular, Hulotte exploits the concept of *component container* to isolate the crosscutting concerns and thus provide a clear separation between business and technical concerns. These crosscutting concerns are reflected at run-time as components, which can be dynamically introspected and reconfigured. Finally, Hulotte defines also a *pointcut language*, named HPath, which can be used to weave aspects *a posteriori* by selecting a set of architecture artefacts to be modified.

8 Conclusion

The research activities presented in this article aim at exploring trails on a unified approach, for which the architectural concepts, methodologies, and principles used to implement the variety of existing *Domain-Specific Component Frameworks* can be factorized. Our goals are to improve reuse, both at application and platform levels, and to propose more flexible solutions for deploying components in heterogeneous execution contexts depending on the targeted application domains.

In this context, this article reports on the Hulotte framework, which relies on using AOP principles at two CBSE levels. First, at the application level, relying on a join point language whose expression capabilities allow the application developer to specialize, by means of domain-specific annotations, its business architectures. Software architectures provide appropriate high-level abstractions of the system—components, their interactions, their compositions—to specify domain-specific execution, communication, or allocation models. Moreover, the uncoupling between the annotations expression and the base architectures on which they will be woven allows the developer to qualify and adapt the domain-specific concerns in a highly flexible way.

Second, at the platform level, where domain-specific annotations are implemented by aspects as fine-grained component assemblies in a symmetric and unified way. Advices are expressed according to the external incoming and outgoing interaction points specified at architectural level by the base applicative components. The weaving process relies on containers deployed on the base applicative architectures, and defining the composition rules between the applicative and the aspect components. This process corresponds to an incremental refinement of the platform where aspects can be injected and composed to generate dedicated containers fitting the domain-specific requirements of the application, while preserving the traceability and a full separation of concerns between these two levels.

Our approach has been validated and has shown its benefits on two case studies addressing real-time embedded applications and the domain of distributed context middleware. It has however a limitation such as the impossibility to completely implement some parts of domain-specific concerns as component architectures. Even if this point is currently handled programmatically by specific extension points of the HULOTTE framework, an open issue is to specify these requirements as code-centric aspects. Another issue is related to the support of dynamic weaving of the platform components at runtime. This is a typical domain-specific concern, which can be implemented by a HULOTTE aspect.

Acknowledgment

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the "*Contrat de Projets Etat Region*" (CPER) 2007-2013 and by the french Minalogic MIND project[8].

References

[aos] Aspect-oriented Software Development web site. http://aosd.net;.

[BCL+06] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B.: The FRACTAL Component Model and its Support in Java: Experiences with Auto-adaptive and Reconfigurable Systems; *Software Practice & Experience*, 36(11–12):1257–1284, 2006.

[8] `http://mind.ow2.org`

[BGB+00] Bollela, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., and Turnbull, M.: *The Real-Time Specification for Java* Addison-Wesley, 2000.

[BHM09] Bureš, T., Hnětynka, P., and Malohlava, M.: Using a Product Line for Creating Component Systems; In *Int. Symp. on Applied Computing (SAC'09)*, pages 501–508. ACM, 2009.

[BSE+04] Budinsky, F., Steinberg, D., Ellersick, R., Merks, E., Brodsky, S., and Grose, T.: *Eclipse Modeling Framework* Addison-Wesley, 2004.

[CHP06] Carlson, J., Haakansson, J., and Pettersson, P.: SaveCCM: An Analysable Component Model for Real-Time Systems; In *2nd Workshop on Formal Aspects of Components Software (FACS'05)*, volume 160 of *ENTCS*, pages 127–140, 2006.

[CL02] Crnkovic, I. and Larsson, S.: *Building Reliable Component-based Systems* Addison-Wesley Professional, 2002.

[Cle02] Clemens Szyperski: *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Addison-Wesley, 2002.

[CRS07] Conan, D., Rouvoy, R., and Seinturier, L.: Scalable Processing of Context Information with COSMOS; In *7th IFIP Int. Conf. on Distributed Applications and Interoperable Systems (DAIS'07)*, volume 4531 of *LNCS*, pages 210–224. Springer, June 2007.

[DHT01] Dashofy, E. M., Hoek, A. V. d., and Taylor, R. N.: A Highly-Extensible, XML-Based Architecture Description Language; In *Working Conf. on Software Architecture (WICSA'01)*, page 103. IEEE, 2001.

[DLLC09] David, P.-C., Ledoux, T., Léger, M., and Coupaye, T.: FPath and FScript: Language Support for Navigation and Reliable Reconfiguration of Fractal Architectures; *Annales des Télécommunications*, 64(1–2):45–63, 2009.

[Fie00] Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures* PhD thesis, University of California, Irvine, 2000.

[FV04] Fuentes, L. and Vallecillo, A.: An Introduction to UML Profiles; In *UPGRADE, The European Journal for the Informatics Professional*, pages 5–13, April 2004.

[GCB+06] Garcia, A., Chavez, C., Batista, T. V., Sant'Anna, C., Kulesza, U., Rashid, A., and de Lucena, C. J. P.: On the Modular Representation of Architectural Aspects; In Gruhn, V. and Oquendo, F., editors, *Proceedings of the 3rd European Workshop on Software Architecture (EWSA)*, volume 4344 of *Lecture Notes in Computer Science*, pages 82–97. Springer, September 2006.

[IAN07] IANA: MIME Media Types; `http://www.iana.org/assignments/media-types`, March 2007.

[KHP+09] Kalibera, T., Hagelberg, J., Pizlo, F., Plšek, A., Titzer, B., and Vitek, J.: CDx: A Family of Real-time Java Benchmarks; In *7th Int. Workshop on Java Technologies for Real-time and Embedded Systems (JTRES'09)*, 2009.

[LMP+09] Loiret, F., Malohlava, M., Plšek, A., Merle, P., and Seinturier, L.: Constructing Domain-Specific Component Frameworks through Architecture Refinement; In *35th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA'09)*, pages 375–382, August 2009.

[LNB09] Lobry, O., Navas, J., and Babau, J.-P.: Optimizing Component-Based Embedded Software; *Int. Conf. on Computer Software and Applications*, 2:491–496, 2009.

[LNBL09] Loiret, F., Navas, J., Babau, J.-P., and Lobry, O.: Component-Based Real-Time Operating System for Embedded Applications; In *12th Int. Symp. on Component-Based Software Engineering (CBSE'09)*, LNCS, pages 209–226. Springer, June 2009.

[LT09] Lau, K.-K. and Taweel, F.: Domain-Specific Software Component Models; In Lewis, G., Poernomo, I., and Hofmeister, C., editors, *Proc. 12th Int.*

Symp. on Component-based Software Engineering, LNCS 5582, pages 19–35. Springer-Verlag, 2009.

[LW07] Lau, K.-K. and Wang, Z.: Software Component Models; *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[MB05] Mencl, V. and Bures, T.: Microcomponent-Based Component Controllers: A Foundation for Component Aspects; *Asia-Pacific Software Engineering Conf.*, 0:729–737, 2005.

[Mor06] Moreno, G. A.: Creating Custom Containers with Generative Techniques; In *5th Int. Conf. on Generative Programming and Component Engineering (GPCE'06)*, pages 29–38. ACM, 2006.

[MPL+08] Malohlava, M., Plšek, A., Loiret, F., Merle, P., and Seinturier, L.: Introducing Distribution into a RTSJ-based Component Framework; In *2nd Junior Researcher Workshop on Real-Time Computing*, October 2008.

[MTM09] Martínez, A. N., Toledano, M. Á. P., and Murillo, J. M.: An ADL dealing with aspects at software architecture stage; *Information & Software Technology*, 51(2):306–324, 2009.

[NL09] Noguera, C. and Loiret, F.: Checking Architectural and Implementation Constraints for Domain-Specific Component Frameworks using Models; In *35th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA'09)*, pages 125–132, August 2009.

[OMG07] OMG: Object Management Group: Unified Modeling Language – Superstructure Version 2.1.1;, 2007.

[OSV08] Odersky, M., Spoon, L., and Venners, B.: *Programming in Scala* Artima, 2008.

[PF07] Pinto, M. and Fuentes, L.: AO-ADL: an ADL for describing aspect-oriented architectures; In *Proceedings of the 10th international conference on Early aspects*, pages 94–114, Berlin, Heidelberg, 2007. Springer-Verlag.

[PFHV04] Pizlo, F., Fox, J. M., Holmes, D., and Vitek, J.: Real-Time Java Scoped Memory: Design Patterns and Semantics; In *7th Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 101–110. IEEE, 2004.

[PFT05] Pinto, M., Fuentes, L., and Troya, J. M.: A dynamic component and aspect-oriented platform; *Comput. J.*, 48(4):401–420, 2005.

[PLMS08] Plšek, A., Loiret, F., Merle, P., and Seinturier, L.: A Component Framework for Java-based Real-time Embedded Systems; In *9th Int. Conf. on Middleware (Middleware'08)*, December 2008.

[Pls09] Plsek, A.: *SOLEIL: An Integrated Approach for Designing and Developing Component-based Real-time Java Systems* Phd thesis, USTL, September 2009.

[PNP06] Pawlak, R., Noguera, C., and Petitprez, N.: Spoon: Program Analysis and Transformation in Java; Technical Report 5901, INRIA, 2006.

[Pro08] Project, A.-E.: Reference Architecture v3.0;, 2008 http://www.aosd-europe.net/deliverables/d103.pdf.

[PSCD06] Pessemier, N., Seinturier, L., Coupaye, T., and Duchien, L.: A Model for Developing Component-based and Aspect-oriented Systems; In *5th Int. Symp. on Software Composition (SC'06)*, volume 4089 of *LNCS*, pages 259–273. Springer, March 2006.

[RCS08] Rouvoy, R., Conan, D., and Seinturier, L.: Software Architecture Patterns for a Context-Processing Middleware Framework; *IEEE Distributed Systems Online (DSO)*, 9(6), June 2008.

[SFV06] Suvée, D., Fraine, B. D., and Vanderperren, W.: A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development; In *9th Int. Symp. on Component-Based Software Engineering (CBSE'06)*, LNCS, pages 114–122. Springer, 2006.

[SK04] Störzer, M. and Koppen, C.: PCDiff: Attacking the Fragile Pointcut Problem, Abstract; In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.

[SMF+09] Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., and Stefani, J.-B.: Reconfigurable SCA Applications with the FraSCAti Platform; In *6th IEEE International Conference on Service Computing (SCC'09)*, pages 268–275. IEEE, September 2009.

[SPDC06] Seinturier, L., Pessemier, N., Duchien, L., and Coupaye, T.: A Component Model Engineered with Components and Aspects; In *9th Int. Symp. on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *LNCS*, pages 139–153. Springer, 2006.

[SVB+08] Sentilles, S., Vulgarakis, A., Bures, T., Carlson, J., and Crnkovic, I.: A Component Model for Control-Intensive Distributed Embedded Systems; In *11th Int. Symp. on Component-Based Software Engineering (CBSE'08)*, 2008.

[SVJ03] Suvée, D., Vanderperren, W., and Jonckers, V.: 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*, pages 21–29, New York, NY, USA, 2003. ACM.