Software Engineering of Component-Based Systems-of-Systems: A Reference Framework

Frédéric Loiret, Romain Rouvoy, Lionel Seinturier, Philippe Merle

Software Engineering of Component-Based Systems-of-Systems: A Reference Framework

Frederic Loiret
KTH (Royal Institute of Technology)
Stockholm, Sweden
floiret@kth.se

Romain Rouvoy, Lionel Seinturier, Philippe Merle
INRIA Lille – Nord Europe, Project-team ADAM
Univ. Lille 1 - LIFL CNRS UMR 8022, France
firstname.lastname@inria.fr

ABSTRACT

Systems-of-Systems (SoS) are complex infrastructures, which are characterized by a wide diversity of technologies and requirements imposed by the domain(s) they target. In this context, the software engineering community has been focusing on assisting the developers by providing them domain-specific languages, component-based software engineering frameworks and tools to leverage on the design and the development of such systems. However, the adoption of such approaches often prevents developers from combining several domains, which is a strong requirement in the context of SoS. Furthermore, only little attention has been paid to the definition of a modular toolset and an extensible runtime infrastructure for deploying and executing SoS. In this paper, we therefore propose a reference framework to leverage on the software engineering of SoS. Our reference framework has been validated on the development of two platforms, namely HULOTTE and FRASCATI, to demonstrate that the resulting complexity is isolated in the core toolset, while the development of domain-specific extensions is leveraged and simplified by clearly identified abstractions.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: [Domain-specific architectures, Service-oriented architecture (SOA)]

General Terms

Design

1. INTRODUCTION

The software engineering community has invested a lot of effort in the development of complex systems deployed from embedded devices to Internet-scale environments. These complex systems, also known as *Systems-of-Systems* (SoS), are characterized by an assembly of a wide diversity of building blocks [12], where processes, tools and design methods should be adapted to various application domains with heterogeneous requirements. At the same time, the *Component-Based Software Engineering* (CBSE) principles have been thoroughly investigated to provide advanced solutions leveraging the characteristics of an application domain. In this context, the high-level abstractions offered by CBSE are tailored for capturing domain knowledge and are used as a cornerstone of the design process, in particular *i)* for generating domain-specific middleware platforms [21], or implementation artifacts linked to third-party middleware (*e.g.*, RT CORBA [23]), supplying end-user applications with reusable patterns, redundant and error-prone tasks; and *ii)* for conducting domain-specific analysis in order to automatically infer analysis models that are of particular importance for a given domain (*e.g.*, schedulability analysis in safety-critical systems). Nevertheless, most of the existing CBSE approaches [9] focus on a single application domain and do not try to identify the key abstractions and principles to introduce enough flexibility throughout the design process for supporting the challenges raised by SoS.

In this paper, we propose a reference framework to isolate the common principles generally applied to develop *Component-Based Systems-of-Systems* (CBSoS). The originality of our solution is based on *i)* the extensive use of *domain-specific annotations* and *ii)* the adoption of a *reflective approach*, where both the CBSoS infrastructure and the toolset apply our reference framework. We assess this reference framework by reporting on two component-based platforms we developed: HULOTTE and FRASCATI. The experiences we realized suggest that our reference framework exhibits a number of benefits, including support of principled development of domain-specific concerns in CBSoS, elimination of redundant development effort, separation of concerns, and homogeneous design and implementation methods simplifying both software maintenance and evolution.

The remainder of this paper is organized as follows. Section 2 presents the reference framework whose implementations and evaluations are detailed in Section 3. Section 4 discusses related works, and Section 5 concludes the paper.

2. REFERENCE FRAMEWORK

Our reference framework is organized as a triangle of three complementary contributions to achieve the desired properties: *i)* a minimal set of high-level model artifacts required for specifying component-based software and their domain-specific extensions, *ii)* architectural styles and guidelines for implementing domain-specific middleware platforms, and *iii)* the key processing steps and extension points required for analyzing and producing the supporting platforms.

The keystone of the reference framework is a component model, whose concept semantics can be easily extended by

CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.

attaching annotations (cf. Section 2.1). These annotations are supported by an open container infrastructure which facilitates both weaving and management of domain-specific concerns (cf. Section 2.2). The integration of these concerns as well as the validation of the generated architecture is ensured by a modular toolset (cf. Section 2.3).

2.1 Versatile Component Model

The simplified abstract syntax of the component model we use as a cornerstone of the reference framework is specified by the meta-model depicted in Figure 1.

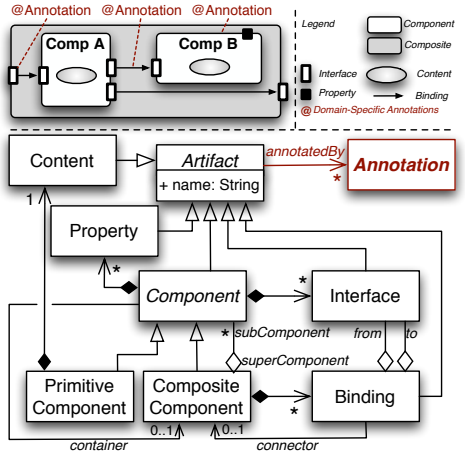

Figure 1: The Component Meta-model.

The key principle of this model allows the design of generic architecture instances whose semantics, roles, domain-specific properties, and meta-data attached to core artifacts can be specialized via the use of *annotations*. Annotations can be simple string-based attributes as well as arbitrary complex data structures or views of the system, which can themselves reference core artifacts. For instance, an annotation can specialize an interface with a particular role (required/provided or data producer/consumer) and a dedicated IDL (a set of operation-based signatures or simple data types). An annotation can also specialize a component and a binding, for setting a domain-specific execution or communication semantics, which will be handled by the underlying middleware platform; or it can provide a domain viewpoint of a composite for analysis purposes. Annotations therefore play a central role throughout the architecture lifecycle, since they are exploited either at design-time by the application developer, at compile-time and/or at runtime by the toolset presented in Section 2.3 to drive interpretation steps.

2.2 Extensible Container

Our reference framework promotes an homogeneous design methodology where both the application, the middleware platform, and the domain-specific services are designed and implemented using the above presented component model. This choice is motivated by the following points:

- To *leverage on the use of architecture-based specifications* in order to improve software production metrics (such as separation of concerns, modularity, reuse, high-level design) at the middleware platform level, which is a key point when designing middleware for SoS, where well identified redundant patterns and algorithms can be abstracted and reused in various application domains;
- To *rely on a single design model* to investigate multiple functional and non-functional properties of SoS encompassing services provided by the underlying middleware. This is of critical importance for designing dependable systems whose reliability constraints must be conjointly ensured at both application and middleware levels;
- To *reify built-in middleware services* as fine-grained components at runtime for providing reflective and reconfiguration capabilities, which is a key requirement for supporting the evolution of family of systems deployed on large-scale environments;
- To *leverage on the use of high-level abstractions at design-time* in order to apply transformation and optimization techniques (in terms of memory footprint, real-time responsiveness and execution time) to generate monolithic and static binaries of the SoS, which is a key requirement for real-time and embedded systems.

Our reference framework makes the distinction between *i)* middleware services, which are local to an application-level *component* instance or a *binding* instance implemented by component-based *containers* and *connectors* respectively, and *ii)* those which are globally shared by the whole application, implemented by *third-party components*.

2.3 Modular Toolset

This section presents the core features and the workflow of a modular and extensible toolset for analyzing and generating CBSoS. This toolset is itself implemented with the component model presented in Section 2.1. A set of core components implements the domain-agnostic logic of the toolset, thus easing the interpretation steps for analyzing and generating implementation artifacts, which are reusable within different family of systems, while the domain-specific logic is implemented by clearly-defined *extension points*. Plug-ins are designed as components implementing an interface dedicated to the extension point on which they should be deployed allowing the domain expert to inject the required logic for handling domain-specific concepts. The reference architecture of the toolset focuses on pre-defined extension points, which are highlighted in Figure 2.

1/ The `Front-End` component is in charge of loading and checking heterogeneous descriptors used to specify CBSoS and instantiating the associated reference component model as well as the domain-specific annotation models. Five extension points are defined for this purpose: *i)* `ADL Loader` supporting various assembly descriptors. *ii)* `IDL Loader` supporting various *IDLs* (*e.g.*, Java, CIDL). *iii)* `Property Loader` supporting various *Property Description Languages* (PDLs, *e.g.*, Java, XSD). *iv)* `Content Loader` supporting various *Component Implementation Languages* (*e.g.*, Java, C). *v)* The last extension point identified at this stage of the workflow (`Description Checker`) performs semantic verifications on the instantiated models.

2/ `Container Generator` is in charge of generating the domain-specific middleware components based on the platform model presented in Section 2.2. The core components of the toolset (not represented in Figure 2) implement a *Visitor pattern* traversing the architecture instantiated by the

`Front-End`. At this step of the process, each application-level component and binding is given as input of the extension points handling the generation of the platform (`Personality Factory`, `Connector Factory`, and `Third-party Integration` components), before an initialization phase of the container and the connector structures. If required, `Content Generation` is in charge of generating the content implementation of the container components, typically in the case of interceptor components, whose content depends on the signature of intercepted interfaces. The generation phase is then finalized by the `Container Checker` extension point.

The generation step thus produces a complete architecture of the system including application-level components composed with the container infrastructure.

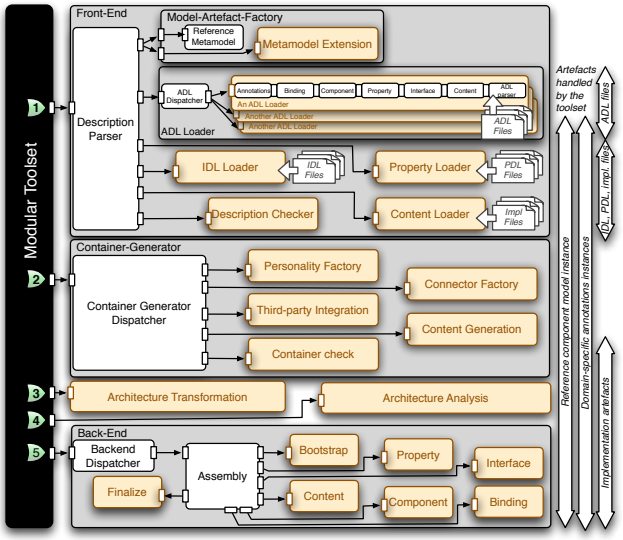

Figure 2: The Toolset and its Extension Points.

3/ `Architecture Transformation` component is an extension point encapsulating plugins that apply transformations on the generated architecture. Within this interpretation step, the architecture can be restructured according to domain-specific requirements. For instance, optimizations can be implemented to fulfill performance requirements, like merging composite components in a single primitive using algorithms dedicated to inlining multiple contents into a single and monolithic one, or flattening the architecture by using only primitive components. The distribution support, when subparts of the generated architecture must be deployed on different distributed nodes according to deployment configurations expressed by annotations is another example of transformations performed at this step. The toolset provides reusable components to perform these transformations (merge, split up, unwrap algorithms among others).

Overall, the processing performed by plugins within this extension point determines how the generated architecture will be reified at runtime.

4/ `Architecture Analysis` is an extension point where plugins related to domain-specific analysis can be bundled within the toolset prior to the last code generation phase of the `Back-End` (*e.g.*, static analysis, event-tree analysis or schedulability analysis in safety-critical systems, performance analysis, reachability analysis to establish liveliness and safety properties of concurrent systems). From the architecture model built in the preceding steps, model transformations can be performed to automatically infer domain-specific analysis models, which can be in turn exploited by external analysis engines (*e.g.*, model checkers, SAT solvers).

5/ `Back-End` reflects the last step of the toolset workflow. It is in charge of producing code-level implementation artifacts for a target executable language, including the generation of the bootstrap and substrate code. The back-end is organized according to the key concepts of the reference component model, as illustrated in the bottom part of Figure 2. The `Assembly` component implements a *two-steps-depth-first* traversal of the input architecture. The first step triggers the extension points if generation of implementation artifacts for each key concept of the model to instantiate is required. The second step consists of generating *bootstrap files* of the assembly. Finally, the `Finalize` extension point is invoked, encapsulating plugins required to generate the final binary of the system (*e.g.*, makefiles generation, compilation, linking).

3. EVALUATION

In order to assess its applicability, the reference framework has been fully applied to the design of two platforms: Hulotte [17][1] which addresses *Distributed and Real-Time Environments* and FraSCAti [27][2] which addresses *Large-Scale Environments*.

In the remainder of this section, we report on the core features of the two platforms that distinguish them. Table 1 summarizes the set of coarse-grained domain-specific features presented in this section and specifies the plugins they implement, according to the extension points detailed in Section 2.3.

3.1 Hulotte: Distributed & Real-Time Embedded Environments

The default ADL language supported by Hulotte is an annotation-aware extension of Fractal-ADL [4], based on XML. Two other languages may also be optionally bundled in the toolset, for parsing architecture descriptors and IDLs specified with Think-ADL [1] (a grammar-defined textual ADL) or FCM[3] (an XMI-based ADL inherited from UML architectural concepts).

The `Back-End` supports two target programming languages: C/assembler and Java. Both are extensible according to the extension points described in Section 2.3 and both are configurable—via the use of annotations—for producing substrate code allowing introspection and reconfiguration of the architecture at runtime. However, since we are concerned by performance issues raised in embedded systems, these features can be applied to a subset of the architecture, or not at all if not required, thus optimizing the memory footprint of the binaries produced by the toolset. Interested readers can refer to [16] for further details. Four domain-specific extensions, described below, have been designed.

Real-Time Audio Applications. Domain-specific extensions have been implemented in order to provide to the application developer a design space for component-based audio and music applications implemented in C, which is fully reported in [18]. These multitask, concurrent and real-

[1] `http://adam.lille.inria.fr/soleil/hulotte/`
[2] `http://frascati.ow2.org`
[3] Flex-eWare Component Model (`http://www.flex-eware.org`).

Table 1: Domain-Specific Features and their Toolset Extension Points.

		Front-End						Container					Arch Trans.	Arch Analysis	Back-End						
Domain-Specific Features		Metamodel	ADL	IDL	Property	Content	Checkers	Personality	Connector	Third-Party	Cont. Gen.	Checkers			Bootstrap	Property	Interface	Content	Component	Binding	Finalize
Hulotte: DRTE Environment	Real-Time Audio	✓	✓	✓			✓	✓	✓		✓	✓	✓								
	RTSJ Framework	✓	✓				✓	✓	✓	✓	✓	✓	✓		✓			✓	✓	✓	✓
	Behav. Models Inf.	✓		✓		✓								✓							
	Impl. Const. Check	✓		✓		✓								✓							
FraSCAti: VLS Environment	Interfaces	✓	✓	✓			✓										✓		✓		
	Properties	✓	✓		✓											✓			✓		
	Comp. Personalities	✓	✓				✓	✓		✓	✓	✓	✓		✓				✓		
	Implementations	✓	✓			✓	✓	✓			✓	✓	✓		✓			✓	✓		✓
	Bindings	✓	✓				✓		✓	✓	✓	✓	✓		✓					✓	

time applications implement audio flows processing algorithms controlled by the end-user via HMIs (*Human-Machine Interfaces*), and requiring basic services of a Real-Time Operating System, implemented by a *third-party component*.

Component Framework for RTSJ. Hulotte has been experimented in the implementation of a component-based framework for RTSJ-based systems, presented in [25]. The *Real-Time Specification for Java* (RTSJ [3]) is a specification for the development of predictable real-time Java-based applications with the concepts of: *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 the garbage collector to ensure predictable memory access among the objects where they are allocated.

Behavioral Models Inference from a Static Analysis. Inferring global behavior properties from an architecture is a requirement to ensure dependability constraints of real-time systems. We propose a behavior specification based on automata, abstracting the internal control flows of the components, where transitions are labeled by incoming and outgoing interactions externalized by the interfaces of the component, and by internal access to global variables defined within the component's scope.

Constraints Checker at Architectural and Implementation Levels. Since we rely on a generic mechanism where architectural artifacts can be annotated by arbitrary annotations, their use imposes constraints for the application developer [22]. Therefore, the implementations of the components must also fulfill the constraints imposed by the use of domain-specific annotations (*e.g.*, for a component annotated with a "thread-dispatch annotation", its implementation must not spawn new threads).

3.2 FraSCAti: Large-Scale Environments

The cornerstone ADL on which FraSCAti is built is the *Service Component Architecture* (SCA) Assembly Language [2] for building large-scale distributed SOA systems. The `Front-End` is thus in charge of loading the SCA assembly descriptors and instantiating the associated SCA and FraSCAti EMF metamodels. While the SCA metamodel groups all the concepts defined by the SCA specifications, the FraSCAti metamodel captures arbitrary domain-specific extensions.

Application-level bindings between components are fully dynamic in FraSCAti since they are implemented at the middleware level as SCA components, *e.g.*, the URI of a Web service (available as a property of these components) can be changed at runtime, thus reconfiguring the architecture of the distributed application. Dynamic component instantiation is another original feature of FraSCAti since the toolset provides an API, which can be invoked at runtime to create new instances of components with different personalities (detailed below), according to the application management requirements. Finally, since the toolset is itself implemented by an SCA composite, this enable the dynamic deployment of new plugins to tailor the toolset to new usage conditions upon demand, unforeseen at startup. Five domain-specific extensions, described below, have been designed.

Heterogeneous IDLs and PDLs. Various *Interface* and *Property Description Languages* can easily extend the kernel of the toolset within the `IDL/Property Loader` extension points, whose plugins provide their parsers and instantiate consequently their EMF models. In addition to Java, the set of plugins currently supports WSDL, UPnP, and C headers in terms of IDLs as well as XSD as a PDL.

Component Personalities. FraSCAti currently supports three personalities designed by component-based *containers* following the architectural styles described in Section 2.2: SCA, OSGi, and Spring.

Component Implementation Languages. In addition to Java 5, which is the default component implementation language of FraSCAti, the toolset can be packaged with plugins supporting other languages: Java Beans, Scala, Spring, OSGi, Fractal, BPEL, and scripting languages.

Bindings. Currently, nine plugins are implemented as FraSCAti extensions for supporting heterogeneous communication protocol technologies: Java RMI, SOAP, HTTP, JSON-RPC, SCA, OSGi, JNA, SLP, and UPnP. These protocols are encapsulated as *connectors*, encapsulating *stubs* and *skeletons* SCA components, which implement protocol specificities, such as message marshalling.

Third-Party Integration. In the case of FraSCAti, the notion of *third-party components* presented in Section 2.2 is used for implementing classical built-in middleware services (*e.g.*, logging, transaction, security, etc.) [27]. These middleware services are reified as services required by the application architecture via annotations, and are instantiated by the `Third-party Integration` extension point of the toolset. The binding mechanism between these two layers is handled dynamically by containers based on aspect-oriented techniques for injecting and managing application-level requests.

4. RELATED WORK

In terms of comparison with *versatile component models*, our contribution is related to Fractal [4], OpenCom [8],

THINK [1], and SOFA [5]. These approaches provide a serious foundation for developing CBSoS, from IT to embedded systems, since they propose extension mechanisms via flexible execution infrastructures also based on the container and connector idioms. However, they fail to manage variability of their conceptual models, such as the capability to handle multiple ADLs, IDLs and implementation languages, they lack generic principles for specializing and parameterizing application-level components with arbitrary domain-specific concerns, and they do not provide a modular toolset handling variability with clearly identified extension points.

In the *Architecture Description Languages* area, most of the initial propositions [20] report an important diversity in terms of semantics implicitly attached to architectural artifacts and interpreted consequently for analysis or synthesis. They rely on monolithic toolsets and are not extensible, except ACME [14] somehow, an architecture description interchange language between different ADLs. A more recent proposal, xADL [10], is an extensible ADL based on modular and composable XML schemas. xADL addresses the design of SoS but does not provide a modular toolset nor a container infrastructure to support the deployment and the execution of these systems. We can also make a connection between the principles presented in this paper for specializing domain-specific middleware platforms and *Aspect-Oriented ADLs*, such as AO-ADL [24], AspectualACME [13], or AspectLEDA [19].

Our work has some relationships with *Model-Driven Engineering* (MDE) approaches, such as [7, 26] or Fujaba [6] based on MDE toolsets for real-time component-based applications.These approaches focus on generating and deploying executable systems and provide extensive model analysis and model transformation capabilities. However, they do not emphasis on the integration of heterogeneous requirements of CBSoS, which is one of the challenge addressed by our contribution.

Finally, [11] and [15] clearly share with our work the idea of an extensible framework for heterogeneous architecture description processing, including architectural analysis, code generation, etc. However, both approaches fail *i)* to propose a systematic handling of the interpretation logic based on core architectural constructs to *finely drive* the domain-specific processing tasks, and *ii)* to *expose clearly identified extension points* throughout their workflows.

5. CONCLUSION

Component-Based Software Engineering (CBSE) has been thoroughly investigated for the development of *Systems-of-Systems* (SoS). Although a plethora of component models and middleware frameworks have been proposed, no consensus has been achieved on a reference framework for building SoS scaling from embedded to very-large-scale environments. Based on our experience in this domain, we present in this paper a reference framework for building component-based SoS, which can be applied in various environments and accommodate a wide variety of domain-specific requirements. This reference framework builds on a *versatile component model*, which is exploited by a *modular toolset* to build *extensible containers*. The originality of our contribution lies in the adoption of the same versatile component model for designing and executing the modular toolset as well as the container infrastructure. This reference framework has been applied on the development of two open-source platforms, HULOTTE and FRASCATI, to validate the acquaintance of the introduced concepts. In particular, we demonstrated that the adoption of this reference framework leverages on the integration of domain-specific requirements from the perspectives of the domain expert and the application developer.

Acknowledgements. The work presented in this paper has been partially funded by the ANR ARPEGE 2008-SEGI-10 ITEmIS project.

6. REFERENCES

[1] M. Anne, et al. Think: View-Based Support of Non-functional Properties in Embedded Systems. In *Proc. of ICESS*, 2009.
[2] Beisiegel, M. et al. Service Component Architecture, 2007. `http://www.osoa.org`.
[3] G. Bollela, et al. *The Real-Time Specification for Java*. 2000.
[4] E. Bruneton, et al. The FRACTAL Component Model and its Support in Java. *SPE*, 36(11-12), 2006.
[5] T. Bureš, P. Hnětynka, and F. Plášil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proc. of SERA*, 2006.
[6] S. Burmester, et al. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of ICSE*, 2005.
[7] J. Carlson, et al. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *Proc. of SEAA*, 2010.
[8] G. Coulson, et al. A Generic Component Model for Building Systems Software. *ACM TOCS*, 26(1), 2008.
[9] I. Crnkovic, et al. A Classification Framework for Software Component Models. *IEEE TSE*, 2010.
[10] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proc. of ICSE*, 2002.
[11] G. Edwards and N. Medvidovic. A Methodology and Framework for Creating Domain-Specific Development Infrastructures. In *Proc. of ASE*, 2008.
[12] P. Feiler, et al. *Ultra-Large-Scale Systems*. 2006.
[13] A. Garcia, et al. On the Modular Representation of Architectural Aspects. In *Proc. of EWSA*, 2006.
[14] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*. 2000.
[15] M. Leclercq, et al. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *Proc. of ICSE*, 2007.
[16] F. Loiret, et al. Component-Based Real-Time Operating System for Embedded Applications. In *Proc. of CBSE*, 2009.
[17] F. Loiret, et al. Constructing Domain-Specific Component Frameworks through Architecture Refinement. In *Proc. of SEAA*, 2009.
[18] F. Loiret, et al. A Three-Tier Approach for Composition of Real-Time Embedded Software Stacks. In *Proc. of CBSE*, 2010.
[19] A. N. Martínez, et al. An ADL Dealing with Aspects at Software Architecture Stage. *Information & Software Technology*, 51(2), 2009.
[20] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Language. *IEEE TSE*, 26(1), 2000.
[21] G. A. Moreno. Creating Custom Containers with Generative Techniques. In *Proc. of GPCE*, 2006.
[22] C. Noguera and F. Loiret. Checking Architectural and Implementation Constraints for Domain-Specific Component Frameworks using Models. In *Proc. of SEAA*, 2009.
[23] OMG. CORBA Component Model. `http://www.omg.org`.
[24] M. Pinto and L. Fuentes. AO-ADL: an ADL for describing aspect-oriented architectures. In *Proc. of Early Aspects*, 2007.
[25] A. Plšek, et al. A Component Framework for Java-based Real-time Embedded Systems. In *Proc. of Middleware*, 2008.
[26] A. Radermacher. Generating Execution Infrastructures for Component-Oriented Specifications with a Model Driven Toolchain. In *Proc. of GPCE*, 2009.
[27] L. Seinturier, et al. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *SPE*, 2011.