Checking Architectural and Implementation Constraints for Domain-Specific Component Frameworks using Models

Carlos Noguera, Frédéric Loiret

Checking Architectural and Implementation Constraints for Domain-Specific Component Frameworks using Models

Carlos Noguera
System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussels, Belgium
Email: cnoguera@ssel.vub.ac.be

Frédéric Loiret
INRIA ADAM Team
University of Lille & INRIA Lille – Nord Europe
LIFL UMR CNRS 8022, ADAM Project-Team
Villeneuve d'Ascq, France
Email:frederic.loiret@inria.fr

***Abstract*—Software components are used in various application domains, and many component models and frameworks have been proposed to fulfill domain-specific requirements. The ad-hoc development of these component frameworks hampers the reuse of tools and abstractions across different frameworks. We believe that in order to promote the reuse of components within various domain contexts an homogeneous design approach is needed. A key requirement of such an approach is the definition and validation of reusable domain-specific constraints. In this paper we propose an extension to the Hulotte component framework that allows the definition and checking of domain-specific concerns. From the components' architecture to their implementations, concerns are defined and checked in an homogeneous manner. Our approach is illustrated and evaluated through the design of an example component-based application for the multitasking and distributed domains.**

***Index Terms*—CBSE, Model-Driven Engineering, Domain-Specific Models, Conformance Checking**

I. Introduction

Component-Based Software Engineering (CBSE) emphasizes software architecture by decomposing engineered systems into logical modules. Examples of this emphasis are found in *Architecture Description Languages* [1] (ADLs) or component frameworks[1] design [2] . Within these approaches, the first-class design entities are the architectural artifacts, which comprise components, their attributes, and the bindings between their interfaces. Semantics attached to architectural artifacts depend on the targetted application domain, exposing relevant *Domain-Specific Concerns* (DSCs) and patterns at architectural level. Examples of DSCs are, grid computing [3], dynamic adaptability [4], distribution support [5] or embedded [6] and real-time constrained domains [7].

The integration of domain-specific concerns in component-based architectures is usually performed in an ad-hoc manner, thereby negating the benefits of reuse that component-based architectures are supposed to offer. In order to address this, we have previously developed the Hulotte framework [8]. Hulotte allows component framework developers to integrate and implement domain-specific concerns using a dedicated design process.

Although the use of Hulotte eases the implementation of domain-specific component frameworks, in order to be used by architects, we believe that domain-specific component frameworks must cater for software engineering concerns that ease their use. In programming languages one of the roles of the compiler is to check that the input program respects the constraints of the language, giving helpful error messages when it is not the case. Similarly, an architect using the domain-specific component framework is expected to adhere to the rules of the framework. It is therefore necessary, when developing the DSCs, to also specify the domain-specific constraints that govern the use of those concerns. In this paper we address the problem of the definition of domain-specific constraints for Hulotte-based applications.

We have identified two kinds of domain-specific constraints: those that DSCs impose on other DSCs, and those that DSCs impose on the source code elements that implement them. We call the former *architectural* constraints and the latter *implementation* constraints. Constraints at the architectural level occur due to the extension/narrowing of the component semantics that the DSCs impose. For example, suppose that components are extended to manage the notion of distribution; then it might be possible that bindings between components in different nodes are restricted to a certain kind of binding. Constraints at implementation level occur whenever the domain-specific concern relies on low-level implementation libraries that have their own restrictions. In the case of the distribution concern, it might be possible that values passed between nodes are restricted to primitive values like integers or strings.

Contribution In order to define and check constraints at different abstraction levels, information not only on the architecture, but also on its implementation are needed. We achieve this by merging a model of the architecture with a model of the implementation using a pivot representation to bind them. The contributions of this paper are twofold:

- A model that incorporates structural information on the architecture of the application and its implementation,

[1]We understand a component framework as a set of kinds of components particular to a given domain

providing an uniform way of defining domain-specific constraints as invariants,
- A toolchain implementing the complete design process presented in this paper, easily extensible towards arbitrary domain-specific concerns.

The rest of this paper is organized as follows. In Section II, we introduce HULOTTE and motivate its usefulness on the development of domain-specific component frameworks through an example. In Sections II-B and II-C, we identify a number of constraints for our example component framework. In Section III we sketch a solution to the problem of the definition and checking of those constraints. The implementation for this solution is presented in Section IV, while its application to our example is discussed in Section V. We position our work against current state of the art in Section VI, and conclude in Section VII.

II. DEVELOPMENT OF DOMAIN-SPECIFIC CONCERNS IN COMPONENT-BASED APPLICATIONS

One of the main benefits expected in using the component paradigm is reuse [9]. However, it has been argued [10] that the vast, and increasing number of proposals to address these domain-specific requirements does not encourage reuse, while sharing common concepts and tooling support. From this observation, we proposed HULOTTE – a prototype framework for the specification and implementation of arbitrary domain-specific concerns in a unified way, which is easily extendable towards different application domains [8]. HULOTTE relies on the generic metamodel presented in Figure 1.

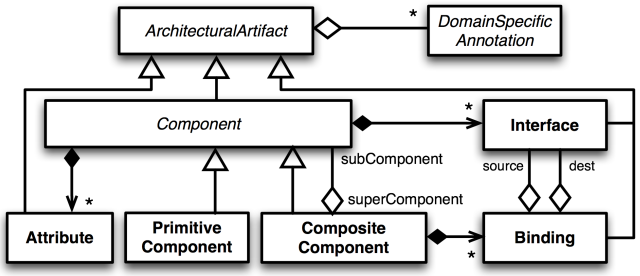

Fig. 1. The Hulotte Metamodel

The metamodel is based on general CBSE principles [9], containing the basic architectural artifacts `Component` (either `Primitive` or `Composite`), `Attribute`, `Interface` and `Binding`. It allows the design of generic component-based applications which are afterwards annotated by `DomainSpecific Annotations` capturing and defining the parameters of the concerns dedicated to the targetted domain. Note that, since the DomainSpecificAnnotation metaclass is associated to ArchitecturalArtifacts, it is possible to annotate any entity in an HULOTTE component architecture.

A. Motivating example

Throughout this paper, we consider the use of HULOTTE to design component-based applications specific to the multitasking and distributed domains. From these two domains, coming from our experiments presented in [7] and [11], we identify six DSCs: in multitasking, periodic, sporadic and protected components; and in distribution, asynchronous bindings, and distributed components. Thus, the following domain-specific annotations are provided to the application developer:

- `@periodic` and `@sporadic` annotations mark components as *active*. An *active component* will be attached to its own thread of control managed by a generated HULOTTE runtime. The component activation is either periodic —*i.e.*, dependent on a periodicity given by the developer—, or sporadic —*i.e.*, triggered by incoming events.
- The `@protected` annotation for a component specifies that the HULOTTE runtime must guarantee mutual exclusion over the execution of its services.
- `@asynchronous` allows the expression of asynchronous bindings between components.
- Finally, the `@distributed` annotation defines the component's allocation within distributed nodes.

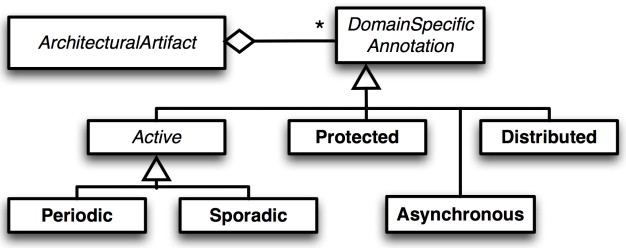

Fig. 2. Domain-Specific Annotations for Multitasking and Distributed Domains

These domain-specific annotations are integrated into the HULOTTE metamodel by extending the `DomainSpecific Annotation` metaclass as depicted in Figure 2.

An application example based on these annotations is illustrated in Figure 3. In this application, components are distributed in two nodes (in gray) that communicate via an asynchronous binding. In the first component (`ActionComponent`), a periodic `Writer` sends data to a shared protected component. This data is then read by a second `Reader` periodic component that, through the asynchronous binding, sends the data to the second `Reaction Component`.

Domain-specific annotations serve two purposes; first, they clarify the developer's intentions, making the architecture

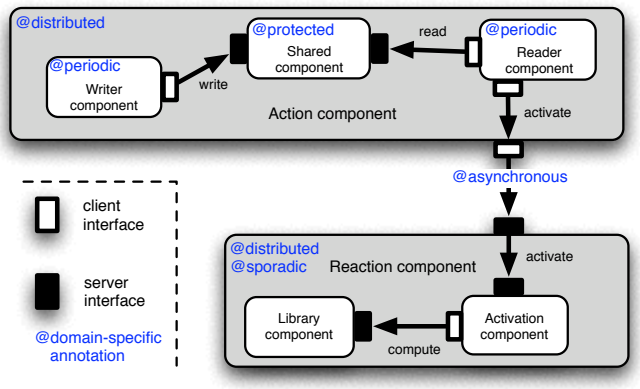

Fig. 3. Motivating Example

easier to understand by refining the semantics of the generic component-based artifacts. In our example this is evidenced by the use of the `distributed` and `asynchronous` annotations. Second, annotations are used by the HULOTTE framework to modify the behaviour of the components. In our example, `periodic` and `sporadic` annotated components will spawn their own threads. Because of this, annotations are more than mere documentation. When defining the annotations, the component-framework developer must define the rules of engagement for the annotations, i.e., what constitutes a correct use of the annotations on a given architecture.

We have identified two kinds of constraints that domain-specific annotations must respect: constraints over the use of other annotations, and constraints that domain-specific annotations impose on the implementation of the components.

B. Constraints Over DSCs at architectural level

Domain-specific annotations extend the semantics of the architectural artifacts on which they are defined. Therefore, they offer a way to specialize a component-based application focused on the functional aspects with domain concerns. However, these DSCs impose constraints on the use of annotations at architectural level. For instance, considering our motivating example, we can highlight the following constraints, further referred as *architectural level constraints*:

- A *periodic* component must not export server interface.
- Each component must be either *distributed* or nested within a unique *distributed* parent composite.
- A component must not be *active* and *protected*.
- An *active* or *protected* composite must not define *active* or *protected* subcomponents whatever their hierarchichal encapsulation level.
- An *asynchronous* binding must not be defined if its destination interface is not exported by *sporadic* component.
- A cyclic composition chain between the same *active* or *protected* component instances must be forbidden to avoid deadlocks.

Using our approach, these constraints should be checked in order to guide the developer in specifying consistent annotated architectures.

C. Constraints Imposed by DSCs at implementation level

HULOTTE provides also a framework composed of a set of high-level tools, methods and patterns allowing to generate runtime platforms in a generic way according to the annotated architectural artifacts. The goal is to spare the application developer from dealing with DSCs at implementation level. For instance, the code implementing the distributed deployment of the application or the execution models of the active components will be generated automatically.

Therefore, since the components may be used in various execution contexts, their implementations must also fulfil constraints imposed by the use of DSC's. In our example, we highlight the following *implementation level constraints*:

- The methods specified within interfaces involved in a distributed binding (i.e. between *distributed* components) must have parameters and return values of primitive types to ease marshaling, and must not declare checked exceptions.
- For *asynchronous* bindings, the methods must define `void` as return type.
- Since the *protected* component services are executed as a critical section within the HULOTTE runtime, the implementations of these services must not be based on internal synchronization mechanisms (such as the `synchronize` Java keyword) to avoid potential deadlocks.
- The *periodic* components must implement a unique task entry point which will be periodically called by the runtime (such as the Java `Runnable` interface).
- The *active* component implementations must not create new threads.

III. SOLUTION STRATEGY

In the previous section, we highlighted the constraints which should be fulfilled to allow the reuse of generic component definitions within various domain-specific contexts.

The scope of the *architectural level constraints* stated in Section II-B is determined by the HULOTTE's architectural abstractions sketched out in Figure 1. Since these constraints operate at the same abstraction level, ensuring their correctness is easy to realize, using for instance classical declarative constraint languages from the architectural models defined by the developer. However, the *implementation level constraints* given in Section II-C implies reasoning at two abstraction levels in a top-down manner: one for the architecture and one for the underlying programming language used to implement the primitive components.

In order to fulfill that requirement, the traceability between these abstraction levels should be insured. It becomes necessary to express the mapping of the high-level architectural artifacts into the implementation, for instance, to express the mapping of an attribute definition or a required interface at architectural level into fields at the implementation one.

To manipulate the different abstraction levels we opt for a model representation. We therefore identify three kinds of required models in order to define and check architectural and implementation constraints:

1) An architectural model that represents a high-level view of the application. This model must contain both generic component elements, as well as domain-specific ones.
2) A model of the implementation language. This model must reflect the structure of the source code elements that implement the application.
3) A mapping between the architectural model, and the implementation model. This mapping takes the form of a pivot model whose elements are related to elements on both the architectural and implementation model, thereby providing a continuum between the two models.

The modeling paradigm offers an appropriate conceptual framework and tooling support to alleviate the complexity of the verification process required by our approach. Therefore, the three abstraction levels defined within HULOTTE – architectural level, implementation level and the pivot representation between them – have been implemented using EMF metamodels [12], as it is described in the next section. Implementing these three levels in the same technology space allows the expression of the HULOTTE's constraints in a unified way. They are implemented with OCL [13] and are detailed in the Section V.

IV. IMPLEMENTATION

Our prototype is based on an extension of FRACTAL [14], a generic-purpose and hierarchical component model, and on its support in Java. Although we only consider the Fractal and Java in this paper; this does not preclude our approach from being applied to other component frameworks and implementation languages, provided that suitable meta-models for them are defined.

In order to implement the constraints required for the definition of DSCs using components, we bring together three different metamodels for each of the abstraction levels described in Section III: HULOTTE, for the architectural description, SPOONEMF for the source code description, and FRACLET model as a pivot between HULOTTE and SPOONEMF. Each of the EMF metamodels are briefly described below.

A. HULOTTE

For the implementation of the DSCs presented in Section II-A, namely multitasking and distribution, we project the HULOTTE concepts into the Fractal component model. In order to do this, we have implemented a parser that reads Fractal description files extended with our domain-specific annotations[2], and constructs an instance of the HULOTTE metamodel described in Figure 1.

[2]FractalADL is an XML-based language, so we use a SAX parser to read Fractal description files as well as our extensions

B. Fraclet Model

Fraclet [15] is an annotation framework that reduces the complexity of the implementation of component-based applications. Java annotations in Fraclet are used to mark classes, fields or methods that implement Fractal concepts. For example, if a Java class represents the implementation of a component, this class will be annotated with the `@FletComponent`[3] annotation. The goal of Fraclet is to allow for the application developer to concentrate on the implementation of the business behaviour of the application, leaving the Fractal-specific boilerplate code to be generated automatically by the Fraclet annotation framework.

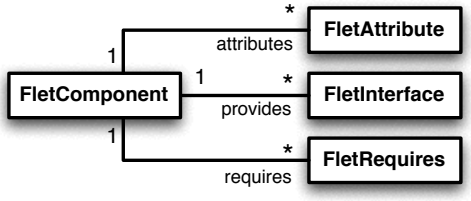

Fig. 4. Fraclet Metamodel

Fraclet, in effect, serves as a conceptual bridge between the high-level entities of the component world and the low-level ones of the implementation language. Because of this, we have chosen Fraclet as the basis for the component-implementation pivot described in Section III. To implement this pivot, we rely on the ModelAn [16] tool-set. ModelAn serves as a reverse-engineering tool that extracts a metamodel out of a set of annotations. By applying ModelAn to the Fraclet annotation framework, we obtain the metamodel depicted in Figure 4.

Additionally to the Fraclet metamodel, ModelAn produces a model-instantiation tool that, from a Fraclet-annotated program, creates an instance of the Fraclet meta-model.

C. SpoonEMF

SpoonEMF[4] is a metamodel for Java, based on the Spoon [17] program transformation framework. Using Eclipse's Modeling Framework, SpoonEMF models each node in the Java AST, as well as the type and name bindings between them. A subset of SpoonEMF's metamodel is presented in Figure 5. In this figure, the models for abstract types (*CtType*), concrete classes (*CtClass*) and the methods inside it (*CtMethod*) are presented.

In addition to Java's metamodel, SpoonEMF provides a parser that, given an application's source code will produce an instance of the model that represents it. Conversely, SpoonEMF is able to take source-code models and pretty print it in order to obtain Java code. These two features make SpoonEMF a fit complement to HULOTTE, and fulfill the requirements stated in Section III.

[3]For FracLETComponent

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

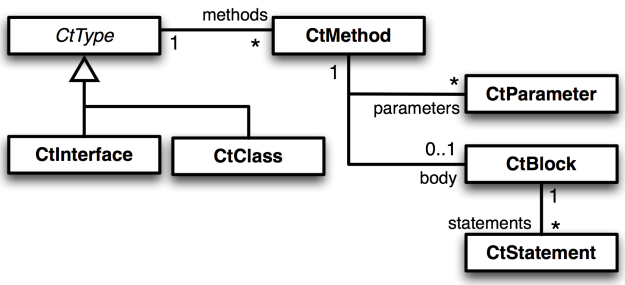

Fig. 5. Subset of SpoonEMF's Java Metamodel

D. Merging HULOTTE, Fraclet and SpoonEMF models

Having defined models for each of the abstraction levels, i.e. architecture, code and mapping between them, we must now define a way to merge them so that constraints that transcend abstraction levels can be expressed. This is done by the inclusion of two new metamodel elements: the concept of *content* of a component, and the concept of *target* of a fraclet annotation as shown in figure 6. The content of a component models the relationship between a *Primitive Component* in HULOTTE and its implementation, defined in the Fraclet model. This relation is reified through a `ContentDesc` metaclass attached to the HULOTTE's Component metaclass. Through the content of a component it is possible to reach the Fraclet annotations that represent it.

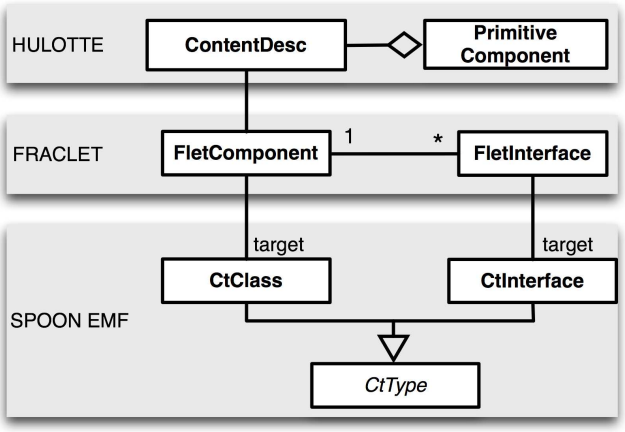

Fig. 6. Merging of Architectural, Pivot and Source Models

To navigate from a Fraclet annotation to the source code element that implements it, we define a direct reference (`target`) from each of the metaclasses defined in Figure 4 to the corresponding metaclass defined in SpoonEMF. Thus, the target of a FletComponent in Fraclet is a CtClass, the target of an FletInterface, a CtInterface, etc. Through the composition of `targets` and `ContentDescs`, it is possible to navigate from a high-level DSC to a low-level source code element.

Instances of the merged models are obtained from applications by using the parsers provided by each of the models. HULOTTE, Fraclet and SpoonEMF parsers are used to go from HULOTTE textual files and annotated Java code to an instance of the model that represents the application under study.

In the following section we present how the merged architecture model and the implementation model are used to define and check the constraints identified in Section II-A.

V. VALIDATION

Sections II-B and II-C enumerate constraints that arise when extending a component framework with the domain-specific concepts explained in section II-A. In this section we present how these constraints are implemented by translating them into OCL invariants over instances of the merging of the metamodels of HULOTTE, Fraclet and Spoon, described in section IV. We use EMF's OCL interpreter to check the constraints on instances of the merged metamodels.

We start with the constraints on the use of the DSCs; and in the following section, treat the constraints that the use of DSCs imposes on their implementation.

A. Constraints over DSCs

In Table I, five OCL invariants are shown. Each expression corresponds to a constraint defined in section II-B. To simplify the construction of the OCL invariants, we have defined a helper `UTIL` class that contains a number of convenience methods to test the presence of annotations on architectural artifacts. The first constraint expresses that a *periodic* component must not export a server interface, which is a straightforward translation to OCL. The second constraint uses a recursive function over the containment relationship in components to check the constraint that states that every component must either be *distributed* or be contained in a component that is. A similar recursive function is used for the third constraint, where *protected* components are forbidden from containing other protected components. The fourth constraint is again a simple translation that checks that a *synchronous* binding is only destined to *sporadic* components. Finally, the fifth constraint is translated using a recursive function that accumulates visited nodes to check for cycles among *active* components.

B. Constraints over Implementation

For the definition of the constraints that DSCs impose on implementation, we employ the link between the HULOTTE and Java models that is provided by the Fraclet-metamodel. All of the constraints defined in Table II use this link through a helper class called `SpoonUTIL`. The first constraint checks that methods in distributed bindings define parameters with primitive types. The `SpoonUTIL` helper class is used to get the Java interfaces (and methods) that implement given distributed components. The second constraint is implemented in a similar manner, and represents the constraint that states that *asynchronous* bindings must define methods with no

1	**Context**:Component **inv**:UTIL.isPeriodic(self) **implies** self.interfaceSet->**forAll**(itf|UTIL.isClient(itf))
2	**Context**:Component **def**:isDistributedN(c: Component) : Boolean = UTIL.isDistributed(c) **or** c.SuperComponent->**exists**(cs|isDistributedN(cs)) **inv**:isDistributedNested(self)
3	**Context**:Protected **def**:superIsProtected(c: Component) : Boolean = UTIL.isProtected(c) **or** superIsProtected(c.superComponent) **inv**:self.AnnotatesSet->**forAll**(c | **not** superIsProtected(c))
4	**Context**:Binding **inv**:UTIL.isAsynchronous(self) **implies** UTIL.isSporadic(self.dest)
5	**Context**:Active **def**:cycle(c: Component, actives: Set(Component), visited:Set(Component)) : Boolean = c.bindings->**forAll**(b | **if** (actives->includes(b.dest)) **then true** **else if** (visited->**includes**(b.dest)) **then false** **else if**(UTIL.isActive(b.dest)) **then** cycle(b.dest, actives->**union**(Set{b.dest}), visited) **else** cycle(b.dest, actives, visited->**union**(Set{b.dest}))) **inv**:self.annotatesSet->**forAll**(c | **not** cycle(c, Set{}, Set{}))

TABLE I
OCL EXPRESSIONS FOR DSC CONSTRAINTS

1	**Context**:DistributedNode **inv**:self.annotatesSet->**forAll**(c | c.bindings->**forAll**(eb | SpoonUTIL.getFracletInterface(eb.source).target.Methods.Parameters **union** SpoonUTIL.getFracletInterface(eb.dest).target.Methods.Parameters->flatten()->**forAll**(p | p.Type.isPrimitive())))
2	**Context**:Asynchronous **inv**:self.annotatesSet->**forAll**(b | SpoonUTIL.getFractalInterface(b.source).target.Methods.Type.SimpleName = 'void' **and** SpoonUTIL.getFractalInterface(b.dest.target.Methods.Type.SimpleName = 'void')
3	**Context**:Protected **inv**:self.annotatesSet->**forAll**(c | SpoonUTIL.getFracletComponent(c).target.Methods->**forAll**(m | m.body.statements->flatten()->**select**(s | s.oclIsKindOf(CtSynchronized))->isEmpty()))
4	**Context**:Periodic **inv**:self.annotatesSet->**forAll**(c| SpoonUTIL.getFracletComponent(c).target.SuperInterfaces->**select**(s| s.QualifiedName = 'java.lang.Runnable')->notEmpty())
5	**Context**:Active **inv**:self.annotatesSet->**forAll**(c | SpoonUTIL.getFracletComponent(c).target.Methods->**forAll**(m | m.body.statements->flatten()->**select**(s | s.oclIsKindOf(CtNewClass) **implies** s.asOclType(CtNewClass).Type.QualifiedName = 'java.lang.Thread')->isEmpty()))

TABLE II
OCL EXPRESSIONS FOR IMPLEMENTATION CONSTRAINTS

return type. Constraints three and five check that certain statements are not present in the body of methods implementing the DSCs: synchronized statements in *protected* components, and instantiation of threads in *active* components. Finally, constraint number four checks that classes that implement *periodic* components extend Java's Runnable interface.

VI. RELATED WORK

We believe that our approach lies in the intersection of three domains: architecture compliance checking, source-code bug finders, and domain-specific modeling.

There are several ways to statically check the structural architectural compliance of an application [18]: reflexion models and relation conformance rules. A reflexion model [19] maps a high-level (architectural) model with a model of the source code that implements it in order to establish whether the source code complies with the architecture. In our case, this mapping is performed by the developer when he annotates the source code with Fraclet attributes. In our approach, the high-level model is extended to cater for domain-specific features that redefine the semantics of the conformance. In this sense, our approach extends reflexion models with domain-specific conformance rules. Relation conformance rules ensure that the relations defined in the architecture are respected in its implementation. In our approach, the introduction of DSC induces new relation conformance rules *at architectural level* (*cf.* §II-B). Our tool checks that the new conformance rules are satisfied in the architecture, their validation at source code level remains a task for future work. Tools such as xADL [10] also allow the linking of architectural entities with their implementation artifacts for the purpose of runtime manipulation of the architecture. Implementation constraints, however are not taken into account in xADL.

Checking domain-specific architectural rules on the source code is possible through code checkers like PMD [20] and the one defined in [21]. However, without a concrete link between the source code rule and the domain from which it stems, the semantic of the rules becomes obscure, hampering their maintenance. Our approach provides such a link.

Domain-specific meta-modeling tools, such as MetaEdit+[5], the Generic Eclipse Modeling System [22] or the Generic Modeling Environment [23], provide some sort of rule definition language in order to define the structural properties of the models they define. However, they are geared towards a full-MDE implementation in which the models are used to generate code, and therefore the validity of the generated code is supposed. In our approach, both the model and the source code are developed by programmers. This requires checking constraints on both the model and the source code.

VII. Conclusion and Future Work

CBSE has been widely tailored to various and heterogeneous application domains, from Architecture Description Languages and Component Frameworks Design communities. These approaches provide higher-level and architecture-oriented design spaces to the application developer. However, we believe that the challenge to address is that of promoting the reuse of components within these various application contexts towards an homogeneous design approach.

With HULOTTE, we proposed a generic component model for which the domain-specific concerns are expressed declaratively on the architectural artifacts. It supports the separation of concerns between the functional architecture –reifing the business logic of the application– and its annotations capturing the domain aspects. This model is easily extensible towards arbitrary domain-specific concerns. However, we have observed that domain-specific concerns imply domain-specific constraints. It is thereby necessary to provide a framework which allows the definition of the (re)use conditions of the functional components and to check them according to these domain-specific constraints.

Therefore, this paper brings the following contributions: first, in order to support the components' reuse according to these concerns in an homogeneous, our approach is based on the modeling paradigm. All the necessary abstractions to check these specific reuse conditions are reified at the modeling level, thus providing a continuum from the high-level architectural artifacts to the code. Our metamodel provides an uniform way of defining domain-specific constraints as invariants at different abstraction levels. Second, we propose a set of tools based on EMF models and a OCL interpreter implementing the complete toolchain of the design process presented in this paper. This toolchain is easily extendable towards arbitrary annotations specified at architectural level. Moreover, it plays an integral part as a front-end of the HULOTTE's generation process, allowing component-framework designers to generate dedicated runtime platforms according to domain-specific annotations.

To evaluate our approach, we have shown the definition of annotations specific to the multitasking and distributed domain based on our generic component model. We have motivated the interest of our integrated metamodel in order to fulfill the constraints imposed by the use of these annotations at architecture level and at implementation level. Finally, we have presented a concrete use-case of our framework, where these constraints have been translated into OCL invariants checked automatically by the EMF's OCL interpreter over the annotated model instances[6].

Future Work We envision three possible avenues for future work. Just as DSCs induce structural constraints in the source code that implements them, they can also induce behavioural constraints. In the case of structural constraints, they are checked on a structural representation of the source code, i.e., the AST. If we are to take into account behavioral constraints, then a new representation for the source code is needed. In the case a behaviour protocol, for example that the methods of a certain interface must be invoked in a specific order, a call graph representation of the program would suffice. Second, in HULOTTE, DSFs are translated into core concepts (components, interfaces architectural patterns). This translation could also require the checking of well-formness constrains in order to verify its correctness. Finally, we can use the high-level model of the architecture provided by HULLOTTE to drive source code-level optimizations. In Fractal, the reconfigurability of its components is obtained at the expense of its efficiency. Fractal run-times such as Julia [14] offer optimization strategies at the expense of runtime flexibility. These optimizations are performed in a system-wide manner. We believe that by annotating the components that must remain reconfigurable, the optimization can be performed in a selective manner.

[5] http://www.metacase.com/mep/

[6] For the implementation issues, the interested reader could download the HULOTTE constraints checker according to the example presented in this paper from http://tinyurl.com/cchecker

REFERENCES

[1] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Language," in *IEEE Transactions on Software Engineering*, January 2000, pp. 26(1):70–93.

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

[3] F. Baude, D. Caromel, and M. Morel, "From Distributed Objects to Hierarchical Grid Components," in *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*. Springer Verlag: Lecture Notes in Computer Science, LNCS, 2003.

[4] E. Gjørven, F. Eliassen, and R. Rouvoy, "Experiences from Developing a Component Technology Agnostic Adaptation Framework," in *11^{th} International Symposium on Component-Based Software Engineering (CBSE'08)*, 2008, pp. 230–245.

[5] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic, "A Component Model for Control-Intensive Distributed Embedded Systems," in *11^{th} International Symposium on Component-Based Software Engineering (CBSE'08)*, 2008, pp. 310–317.

[6] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[7] A. Plšek, F. Loiret, P. Merle, and L. Seinturier, "A Component Framework for Java-based Real-time Embedded Systems," in *9^{th} International Middleware Conference (Middleware'08)*, Leuven, Belgium, December 2008.

[8] F. Loiret, M. Malohlava, A. Plšek, P. Merle, and L. Seinturier, "Constructing Domain-Specific Component Frameworks through Architecture Refinement," in *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '09)*, August 2009.

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

[10] E. M. Dashofy, A. V. d. Hoek, and R. N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language," in *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2001, p. 103.

[11] M. Malohlava, A. Plšek, F. Loiret, P. Merle, and L. Seinturier, "Introducing Distribution into a RTSJ-based Component Framework," in *2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC'08)*, Rennes, France, 2008.

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

[13] *UML 2.0 OCL Specification*, OMG, ptc/03-10-14.

[14] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani, "The Fractal Component Model and its Support in Java," *Software: Practice and Experience*, vol. 36, pp. 1257 – 1284, 2006.

[15] R. Rouvoy and P. Merle, "Leveraging Component-Based Software Engineering with Fraclet," *Annals of Telecommunications, Special Issue on Software Components – The Fractal Initiative*, vol. 64, no. 1-2, p. 65, 2009.

[16] C. Noguera and L. Duchien, "Annotation Framework Validation using Domain Models," in *Fourth European Conference on Model Driven Architecture Foundations and Applications*, Berlin, Germany, June 2008, pp. 48–62.

[17] R. Pawlak, C. Noguera, and N. Petitprez, "Spoon: Program Analysis and Transformation in Java," INRIA, Tech. Rep. 5901, 2006.

[18] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *WICSA '07: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, p. 12.

[19] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," in *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA: ACM, 1995, pp. 18–28.

[20] T. Copeland, *PMD Applied*. Centenial Books, 2005, iSBN 0-9762214-1-1.

[21] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and Continuous Checking of Structural Program Dependencies," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 391–400.

[22] J. White, D. C. Schimdt, A. Nechypurenko, and E. Wuchner, "Introduction to the Generic Eclipse Modeling System," *Eclipse Magazine*, vol. 6, pp. 11–18, Jan. 2007.

[23] C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *International Workshop on Intelligent Signal Processing*. IEEE, 2001.