Software Architecture Constraints as Customizable, Reusable and Composable Entities

Chouki Tibermacine[1], Christophe Dony[1], Salah Sadou[2], and Luc Fabresse[3,4]

[1] LIRMM, CNRS and Montpellier University, France
[2] VALORIA, Université Bretagne-Sud, Vannes, France
[3] Université Lille Nord de France, France
[4] École des Mines de Douai, France
{tibermacin,dony}@lirmm.fr,sadou@univ-ubs.fr,luc.fabresse@mines-douai.fr

Abstract. One of the major advantages of component-based software engineering is the ability for developers to reuse and assemble software entities to build complex software. Whereas decomposition of software into components has been and is largely addressed for what concerns the business (functional) part of applications, this is not yet the case for what concerns their documentation (non-functional) part. In this paper, we propose a new and original solution to express component-based software non-functional documentation, and we will focus more especially on architecture constraints, which formalize parts of architecture decisions, as executable, customizable, reusable and composable building blocks represented by components. Component-based applications using business and constraint components can be modeled with CLACS, a dedicated ADL which is also introduced in the paper. Architecture constraints can be executed at design-time within CLACS. CLACS is implemented as a plugin in the Eclipse IDE.

1 Introduction: Context and Motivation

Architecture constraints play an important role in design decision documentation and architecture validation. These constraints are often specified either textually or formally, but no means are proposed to customize them for their reuse in different contexts or to compose them in order to define complex constraints. The goal of the work presented in this paper is to propose a way to build architecture constraints as checkable entities embedded in a special kind of software components that can be reused, assembled, composed into higher-level ones and customized using standard component-based techniques. The purpose is as well to put reusable constraint-component on shelves (design for reuse) and to produce new constraints by composition of existing ones (design by reuse) and then to simplify the expression and definition of constraints (ascending design). An additional fundamental goal is to define a uniform paradigm to develop business and non-functional (constraint-) components. We aim thus at proposing an operational component-based design environment providing new capabilities to

express architecture constraints that can be executed at design-time to check the conformity of architecture designs and in which business components can be compiled into instructions of a component-based programming language.

The remaining of the paper is organized as follows. In the following section, we first introduce CLACS, the ADL we built for the SCL [4] component programming language which has been developed in our team. We then explain how using this ADL we can describe constraints as components and how these components can be connected to other constraint components or business ones. Before concluding and presenting the future work at the end of this paper, we make an overview of the related works.

2 Architecture Constraint-Components

Our solution is embedded into an operational software suite (CLACS-SCL) made of an architecture description language (ADL) called CLACS (*Constraint Language for Architectures of Component-based SCL-like software*), and of a component-oriented programming language named SCL (*Simple component language* [4]). CLACS is a modeling alternative for SCL.

2.1 Constraint-Components vs. Business Components

In order to not add (yet-)other constructs for constraint-component modeling, we chose to use the same constructs as for business component modeling. SCL business components and CLACS constraint components share most of their characteristics. The difference between them is expressed in the implementation of services. In business components, services represent traditional operations with a body containing the SCL code implementing the business logic. In constraint-components, the body contains the code of the constraint to be checked (specified in ACL [7] which is an adaptation of OCL).

2.2 Constraint-Component Specification in CLACS

Suppose we define a constraint-component which checks the Façade pattern. The descriptor of this component can be specified in CLACS and instantiated in a given architecture description. Each *Façade checker*, instance of this descriptor, owns one provided port named `Checking` that exports a constraint checking service having this signature : `boolean isFacade(aPort:Port,aSubComp:Component)`. Each *Façade checker* can then be connected, through that checking port, to any business component requiring this service i.e. having a corresponding required interface.

2.3 Connecting Constraints to Architectures

When designing a software architecture, the developer can connect constraint-components to business ones. The binding used to connect these two model elements makes it possible to validate the architecture design according to the constraints embedded in the constraint-component. When invoked within our modeling environment, a constraint-component provided service returns true if the architecture of the business component to which it is connected fulfills the

constraint. When such a connection is established and a constraint evaluated, the constraint expressions interpretor automatically binds the `context` identifier, used in constraints expressions, to the business-component to which the constraint will be applied. When composite constraint-components are built in which a constraint-component is connected to another one, a transitive closure is computed on that link until a business-component is found.

2.4 Constraint-Component Composition

A constraint-component can be assembled together with other constraint-components to build more complex ones. We have defined one kind of binding for each logical operator (and, or, xor and implies). Delegation bindings linking constraint-components can be of kind "`affirmative`" or "`negative`". In the first case, if the constraint-component is bound to a business-component, this means that the architecture of the latter component should respect the constraint embedded in the former component. However, in the second case (`negative` delegation binding), the architecture of the business component should not respect the architecture choice formalized within the constraint-component.

2.5 Constraint Checking

Architectural constraint checking is performed at design time. Thus, constraint-components are interpreted at this stage contrarily to business-components which are executed after their deployment. The constraint checking is implemented by a simple function. Depending on the kind of constraint-components (composite or primitive), the local evaluation corresponds to a delegation to another sub-component or to an ACL interpreter. The propagation of the context within the different constraint-components is done during constraint checking. This allows the evaluation of the constraints on the appropriate business component.

3 Related Work

Different existing ADLs embed constraint languages. Acme [5] and Wright [2] are two representative examples of them. Constraints in Acme and Wright do not represent first-class entities for composition. In addition, constraints in these languages are fixed expressions, which cannot be parameterized to reference a part of the architecture description (with identified components). As presented in the previous sections, CLACS implements a customizability feature at the architecture constraint description level, which allows designers to define reusable constraints. Being embedded in components, these constraints can be easily assembled to extend existing architecture constraint specifications.

Design pattern schemas [6] and component specification patterns [1] are descriptions which allow the definition of templates of OCL constraints with some parameters which are fixed during the instantiation of the templates. As in our work, constraints are parameterized with model elements and are used as library modules. However, model elements (parameters) in our case are architectural elements and constraints target structural descriptions, whereas, in [6], model elements are UML class entities and in [1] constraints target the functional (behavioral) aspect of components.

4 Conclusion and Future Work

Sometimes, defined manually (from scratch) architectural decisions' documentation is complex, error-prone and time-consuming. Having a means to define such documentations by hierarchical composition of constraints is beneficial for two accounts: First, by decomposing the models of architecture constraints in several small interfaced documentation parts, a common repository of reusable (parametrized) assets is provided for software architects; and second, this is a logical way of doing in the continuum of artifact development in component-based software engineering[5].

We implemented CLACS as a prototype in the Eclipse IDE by using some existing plugins [3]: the EMF (Eclipse Modeling Framework) module which allowed us to define an Ecore metamodel of CLACS to generate an editor, and the GMF (Graphical Modeling Framework) plugin to give a graphical dimension to the editor. SCL code generation feature in this editor allows to generate SCL code starting from EMF models. This has been done using the JET (Java Emitter Templates) Eclipse plugin [3]. At the conceptual level, we plan to enrich constraint-components with the other parts of architecture decision documentation. This will help to incrementally build complex non-functional documentations by composition and thus get the advantages of component-based software engineering. In addition, we are investigating the proposition of a model of reflective components. At the tool level, we plan in the near future to work on constraint-component code generation. This will help to check architecture constraints at the evolution stage on implementation artifacts (SCL code). Our aim in the future is also to build a repository of classified architecture constraints.

References

1. J. Ackermann and K. Turowski. A library of ocl specification patterns for behavioral specification of software components. In *Proc. of CAiSE'06*, pages 255–269. Springer-Verlag, 2006.
2. R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997.
3. Eclipse. Eclipse Modeling Project. Eclipse Board Web Site : http://www.eclipse.org/modeling/
4. L. Fabresse, C. Dony, and M. Huchard. Foundations of a Simple and Unified Component-Oriented Language. *Journal of Computer Languages, Systems & Structures*, 34/2-3:130–149, 2008.
5. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge Univ. Press, 2000.
6. M. Giese and D. Larsson. Simplifying transformations of ocl constraints. In *Proc. of MODELS'05)*, Montego Bay, Jamaica, October 2005.
7. C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *In Journal of Systems and Software (JSS), Elsevier*, 2010.

[5] In the same spirit, the Eiffel language has been proposed for, at the same time, programming applications' business-logic and formalizing functional constraints (contract programming with assertions).