Customizable, Reusable and Composable Architecture Contracts

Chouki Tibermacine, Roseline Bénière, Christophe Dony, and Luc Fabresse
LIRMM, CNRS and Montpellier-II University, France
161, rue Ada, 34392 Montpellier Cedex 5, France

Abstract. One of the major advantages of adopting component-based software engineering is the ability for developers to reuse and assemble software entities to build complex software. Business (functional) component reuse and assembly has been dealt with by many researchers in the literature. However, the issue of reusing and assembling of (non-functional) documentation of software components has not been addressed deeply. In this paper, we propose an original solution to define architecture contracts which represent non-functional documentation of business component software. These contracts are considered as reusable, composable and customizable building blocks. They are defined by constraints formalizing architecture decisions associated to the guaranteed quality attributes. These different parts of architecture contracts are thus provided through ports by a specific kind of components. These ports can be either required by business components to document architecture design decisions taken while designing these components, or required by other contract components to build more complex contracts. These contracts have been implemented in CLACS, an ADL which is introduced in this paper. This architecture design language is associated (at implementation stage) with SCL, a pure component-oriented programming language. Through all of these languages and their tool-support, we provide a homogeneous environment for component-based software development.

1 Introduction: Context and Motivation

It is well known that some of the main “ilities” of component-based software engineering are reusability, composability and customizability. Reusability represents the ability for a given piece of software to be reused by developers. While shifting from design to implementation, developers are thus able to concretize a given design element by using a pre-developed software entity (development by reuse). Within the development process, developers are responsible for putting on shelves the produced software artifacts (components) during implementation for future system development (development for reuse). The second non-functional characteristic is inherent to component-based software development. Indeed, in this development paradigm, software building blocks that explicit their dependencies with their environment offer a connection capability of these different
pieces of software to build a complex system. Customizability is the ability for a software to be changed by developers in order to adapt it to given context. There are different methods to reach customizability. One of the most known techniques used for this purpose is parameterization. Indeed defining parameters in the signature of a given software entity allows the developer to customize the software entity behavior according to the passed parameters.

As basically introduced in [19], “architecture contracts” represent a non-functional documentation for component-based software architectures. During architecture design, for each motivated architecture design decision, developers have the ability to formalize, in these contracts, this decision and its rationale. The architecture decision is defined as an OCL-like architecture constraint and the rationale is represented by the quality attribute concretized by the design decision. These contracts are used during the evolution stage of the software lifecycle. They have two goals: i) software architecture comprehension: before applying changes on a component-based software, this documentation helps to better understand the overall structure. It explicit the architecture decisions made by developers and what are the reasons behind these decisions; ii) software Architecture Change Assistance: while applying changes on a given software system, an assistance system notifies the developers if an architecture decision has been altered and precises the affected quality attributes and at which degree. In [19], we presented in more detail the component-based software evolution assistance mechanism. In this paper, we do not focus on this aspect of using architecture contracts. We deal however with their description and reuse.

In an architecture contract, the most important part (by its size) is the architecture constraint. As illustrated in the example given below (which is introduced and explained in [18]), architecture constraints are sometimes long, complex expressions with some repeated parts.

```oclm2xml
context ACS:CompositeComponent
inv: ACS.subComponent.port ->forAll(p:Port|(p.kind = 'Input') or (p.kind = 'Output'))
and
-- Each subcomponent should have input and output ports
ACS.configuration.role.connector ->AsSet()->forAll(com:Component|com.port ->exists(p:Port|p.kind = 'Input') and (p.kind = 'Output'))
and
-- Each connector should define two roles (one sink and one source)
ACS.configuration.role.connector ->AsSet()->forAll(com:Connector|com.role->size() = 2 and ((com.role.kind = 'Source') or (com.role.kind = 'Sink'))) and
-- Each connector should bind two components
ACS.configuration.role.connector ->AsSet()->forAll(com:Connector|com.role->forAll(r:Role|ACS.subComponent ->exists(p:Port|(r in ACS.configuration.binding) and ((p.kind = 'Input') and (r.kind = 'Sink')) or ((p.kind = 'Output') and (r.kind = 'Source')))))
and
-- The graph should be connected
ACS.configuration.isConnected
and
-- The graph should contain the number of vertices - arcs
ACS.configuration.role.connector ->AsSet()->size() = ACS.subComponent->size()-1
and
-- The graph represents a list
ACS.subComponent->forAll(com:Component|com.port ->exists(p:Port|p.kind = 'Input') and (com.port->size() = 2) and (com.port->exists(p:Port|p.kind = 'Input')) and (com.port->exists(p:Port|p.kind = 'Output'))) )
```

This constraint represents the choice of the pipeline style to organize the subcomponents of a given imaginary composite component called ACS. Two remarks can be done on this example. First, the constraint is composed of many parts
that are assembled together. For example, a part of the constraint states that the set of all the subcomponents of ACS represent a connected graph and another part precises that this graph should be represented by a list. This illustrates the **composability issue** that we deal with in our work. Second, the different parts of the constraint could be easily reused in other contexts. Indeed, the part of the constraint stating that the subcomponents represent a list can be reused in many other contexts. This represents the **reusability issue** that we deal with.

The previous constraint is related to the maintainability quality attribute and supposes that “all” the sub-components of this component are organized as a pipeline. Here is another example that deals with the portability attribute. The constraint in the listing below formalized the façade architecture pattern -with analogy to façade objects [9]- and states that the **DataManagement** provided interface of ACS component must be bound internally to one and only one interface. The latter interface corresponds to the provided interface of **DataAdminRetrieval** component. This component represents the façade component. All communications from clients to ACS data management services transit by this component.

```context ACS:CompositeComponent inv:
let boundToDataManagement:Bag*ACS.port.interface
 => select(i:Interface|i.kind = 'Provided'
and i.name = 'DataManagement').port.binding
in
((boundToDataManagement->size() = 1)
and (boundToDataManagement.interface
 => select(i:Interface|i.kind = 'Provided').port.component.name
 => includes('DataAdminRetrieval')))```

We can observe that, in the context of the super-component ACS, this constraint contains identifiers that represent the sub-components or other architecture elements concerned by the constraint. Defining these architecture elements as parameters would allow us to make this constraint customizable for other uses. This represents the **customizability issue** that we deal with in our work.

The current expression of architecture constraints and non-functional documentation of component-based software (architecture contracts) is not optimally designed for answering the issues identified in the previous examples. In this paper, we present our work on the definition of such artifacts as reusable, customizable and composable architectural entities (components). The paper is organized as follows. In the following section, we introduce a new model for the concept of an architecture contract. In section 3, we first present CLACS an ADL we built for the SCL component programming language which has been developed by our team. We explain then how this ADL allows the description of contracts as components and how these components can be connected to other contract components or business ones. We expose in section 4 the prototype tool developed for implementing our proposals. Before concluding and presenting the ongoing and future work at the end of this paper, we make an overview of the related works.
2 A New Model for Architecture Contracts

As introduced in [19], an architecture contract is a documentation of architecture design decisions. It defines in a formal way the links between architecture decisions and quality attributes implemented by these decisions. We describe in this section how such a documentation has been enriched in the work presented in this paper and which information it embeds.

![Diagram of links between Architecture Decisions and Quality Attributes](image)

Fig. 1. Links between Architecture Decisions and Quality Attributes

An architecture contract abstracts the links between a given quality attribute and an architecture decision associated to this attribute. Figure 1 shows how these links are organized. We associate to a link a degree of satisfaction. An architecture decision in collaboration with other decisions contribute to the satisfaction of a given quality attribute. Each degree of satisfaction represents a percentage. In the ideal situation (where the developers are confident in the pertinence of their design decisions), the sum of all degrees associated to the same quality attribute (within the same architectural element) would be equal to 100. For example, a portability quality attribute can be concretized by three different architecture decisions: the choice of the facade pattern [9], the choice of the MVC pattern [3] and the use of an API. If the developers consider that the two first decisions contribute more, in the concretization of the portability quality attribute, than the third one, because they are critical, they can associate to them high scores (for example 40 % to each decision) and the last architecture decision a low score (15 % for example).

An architecture decision in a contract is formalized by an architecture constraint. Here again, a degree is a percentage associated to the link between an architecture decision and an architecture constraint. This score represents the
extent to which the constraint formalizes the design decision. If we consider that several constraints formalize the same architecture decision, it is possible for the developer to precise how the different constraints share the formalization of the design decision. In some cases, a given constraint may have a degree of formalization more important than others. In the ideal situation (where the developers are sure of the completeness of their formalization), the sum of all degrees associated to the same architecture decision would be equal to 100. The constraints written in a given contract are defined with a predefined constraint language.

A quality attribute in a contract is a non-functional property representing an ISO 9126\(^1\) characteristic or sub-characteristic (Reliability, Maintainability, Portability, ...). It has a degree of criticality (inspired from Kazman’s quality attribute scores and Clements’ quality attribute priorities [6]) which is specified by developers and represents the importance of this quality attribute in the architecture. Its possible values are: very high, high, medium, low and very low.

Associated to a given architecture decision, a quality attribute can enhance other quality attributes. For example, the choice of the pipeline architecture style targets the maintainability quality attribute, which enhances in this case the efficiency attribute of the system. Contrarily, a given quality attribute can collide with other quality attributes. For example, the security quality attribute collides generally with the efficiency attribute. This depends of course on the documented architecture decision and the application context. It is on the responsibility of developers, fully aware of the application’s context and the architecture decisions they made, to document these optional parts (the other quality attributes that collide-with or enhance the documented quality attribute) of an architecture contract.

In the current implementation of architecture contracts, architecture constraints are specified using a modified version of OMG’s OCL [16]. As illustrated in the example of the previous section, an architecture constraint in this language navigates in the metamodel of the language used to define the architecture of the components but apply to only one instance of that metamodel (a model which represents the component-based architecture description). The evaluation of a given constraint tells us whether the architecture description conforms to the constraint guards or not.

### 3 Architecture Contracts as Components

In this section, we present how to specify architecture contracts as components. We present then how this entities are assembled together to model and implement complex contracts. At the end of this section, we show how we evaluate these component contracts to check the validity of architecture descriptions and how we reuse them.

To define architecture contract components, we developed an ADL (Architecture Description Language) called CLACS (Contract Language for Architecting Components in SCL). This language is based on the SCL (Simple Component Language [8]) component-oriented programming language. We chose SCL because it is a pure component-oriented language where components are first-class entities: connectors and primitive types are components, and argument passing is done using an original component connection mechanism. Before detailing how architecture contract components can be defined, we introduce SCL and CLACS.

3.1 SCL Component Model

SCL [8] that stands for Simple Component Language is the result of our study and research about component-oriented programming. SCL has been designed to be:

- minimal because all its abstractions and mechanisms respond to an identified need;
- simple because these abstractions and mechanisms are of a high-level;
- detailed because during the conception of SCL we study a lot of crucial points usually forgotten by other propositions such as self-references, arguments passing based on connections or considering base components (collections, integer, etc) in a unified world;
- dedicated to component-oriented programming because it integrates the two key points that we identified: decoupling (which means that a component doesn’t contain an explicit reference to another component) and unanticipation (which means that the concrete components that will be connected later on to one component are unknown by its programmer).

SCL synthesizes the existing well-known features of component-oriented programming. In SCL, a component is a runtime entity created by the instanciation of a descriptor. A component has ports, which have three properties: i) a direction (required or provided), ii) a visibility (internal: the port is private to the component which defines it, or external: the port can be bound to other external components), and iii) an interface which defines the port. Services are defined within interface descriptions. Standard (simple) bindings link ports of components of the same level of hierarchy. As in UML, delegation bindings link components to their subcomponents. Port binding is achieved using the bindTo primitive. More sophisticated bindings can be defined using connectors and glue code. A connector is a component that receives service invocations through its source ports and transmits them through its target ports by executing the glue code. The glue code allows the connection of ports which are not compatible (ports that exchange data which need to be adapted before submission from the source to the target).

SCL also integrates other original features not detailed in this article (because out of the scope of this work) such as an extensible and uniform component
connection mechanism or the possibility to use a component as a “regular” component or a “crosscutting” component as in aspect-oriented development. SCL is currently prototyped in Smalltalk.

3.2 CLACS: an ADL for SCL Component Modeling

In order to help SCL developers to model their application, we developed a graphical environment for component modeling. This environment is based on an ADL called CLACS. We proposed to define a new ADL because existing ones do not answer completely SCL requirements. In addition, in the modeling environment, SCL code generation was an important functionality that we had to deal with. The ADL should thus provide the same architecture constructs as in SCL and with the same semantics. Of course, in this ADL, the focus was on the design and not on the implementation syntactic constructs.

In CLACS, we can model component-based applications that are serialized in an XML format. Section 4 give more details about the underlying tool-level implementation technologies. In CLACS, we find the same concepts as in SCL. A CLACS document represents a component descriptor. Figure 2 shows a metamodel of CLACS.

![Fig. 2. CLACS Metamodel](image-url)
3.3 Contract-Component Specification

In a contract component description, we define constraint components. In this section, we start first by presenting (constraint-)components, then we explain how these components are enhanced with non-functional information to model contracts.

Constraint-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. The only differences are in the specification of services that define constraints, and the use of the kind attribute in the ComponentDescriptor and Service meta-classes (see Figure 2). The body of a constraint is defined in a service which represents the checking operation of the constraint it embeds. The service’s signature is composed of its name and a set of arguments representing architectural elements defined in the metamodel (component instances, ports, ...). The checking of constraints always returns a boolean value. Through these signatures, we have introduced constraint parameterization. Formal parameters defined in the (constraint-)service signature are used in the body of the constraint. Effective parameters are passed by components bound to the (constraint-)component implementing this (constraint-)service. Provided ports that export constraint services can be used by other constraint components to build aggregate constraint components, or by business components to attach the constraint to the component description. Required (constraint) ports are defined either in constraint components or business ones. Defining these ports in a given component explicits the fact that the component needs to use the constraint-checking service. Another kind of bindings has been added to the ADL. It is called constraintConnection and can link a business component to a constraint one. In the listing below, we illustrate by a simple example the XML serialization of a constraint component description in CLACS.

```xml
<cl:Component_Descriptor name="ConstrainedPortComponent" kind="constraint">
  <cl:Interface name="Intf1">
    <cl:Service name="isConnectedPort" kind="constraintCheck">
      <cl:Arg>
        <cl:Name>aPort</cl:Name>
        <cl:Type>Port</cl:Type>
      </cl:Arg>
      <cl:Body language="OCL">
        context.port->exists(p | (p.name = aPort.name)
        and (p.binding->size() <> 0))
      </cl:Body>
    </cl:Service>
  </cl:Interface>
  <cl:Port name="CheckPort">
    <cl:Direction>provided</cl:Direction>
    <cl:Visibility>external</cl:Visibility>
    <cl:DefinedByInterface>Intf1</cl:DefinedByInterface>
    <cl:Description>
      The port received as a parameter should be connected
    </cl:Description>
  </cl:Port>
</cl:Component_Descriptor>
```
The constraint defined in this component states that a given port (aPort, received as a parameter) should be connected. We voluntarily simplified the context of a constraint (context identifier) which always references a component. If the constraint component is connected to a business one, context represents thus this (business) component. If the constraint component is connected to another constraint component, context represents this business component to which is connected the latter constraint component.

(Constraint-)Component Composition Once a (constraint-)component defined, it is possible to connect it with other constraint (or business) components. In the following listing, we illustrate the composition of the previous component with another (constraint-)component. Then, we show how to connect a constraint to a business component.

```
<cl:Component_Descriptor name="AllPortsConstrainedComponent" kind="constraint">
  <cl:Interface name="Intf2">
    <cl:Service name="hasAllPortsConnected" kind="constraintCheck">
      <cl:Body language="OCL">
        context.port->forAll(p | CheckPort.isConnectedPort(p))
      </cl:Body>
    </cl:Service>
  </cl:Interface>
  <cl:Port name="CheckPort">
    <cl:Direction>required</cl:Direction>
    <cl:Visibility>external</cl:Visibility>
    <cl:DefinedByInterface>Intf1</cl:DefinedByInterface>
  </cl:Port>
  <cl:Port name="CheckAllPorts">
    <cl:Direction>provided</cl:Direction>
    <cl:Visibility>external</cl:Visibility>
    <cl:DefinedByInterface>Intf2</cl:DefinedByInterface>
    <cl:Description>All the ports should be connected</cl:Description>
  </cl:Port>
</cl:Component_Descriptor>
```

In this description, we define a new (constraint-)component (AllPortsConstrainedComponent) which provides a constraint port named CheckAllPorts. This port defines a constraint service which states that all the (business) ports should be connected. In the constraint we make use of the signature of the constraint-service isConnectedPort(Port aPort) through the required constraint port CheckPort. In this call we pass, each time (in the forAll(...) collection operation) as a parameter, p which represents one of the ports of the context component.

In the following example, we illustrate the description of a business component named PasswordManager. This component defines a business required port (Randomizer) and a constraint required port defined by the same interface as the one described previously for the constraint provided port of the first (constraint-)component (ConstrainedPortComponent).
In the example, illustrated in Figure 3, we define a component descriptor (of a “composite component” called PasswordManager) which assembles a component instance of PasswordGenerator with a constraint component one.

![Diagram](image)

**Fig. 3.** Example of an Architecture Constraint Component Assembled with a Business Component

In this example, the PasswordManager component defines an instance of the PasswordGenerator component and an instance of the ConstrainedPortComponent component. In addition to the two delegation bindings illustrated in Figure 3, a binding of type constraintConnection is defined between the business component instance and the constraint component one. This binding is specified as illustrated in the following CLACS code. In this binding, we precise the parameter passed for the constraint component (pg.Randomizer).

```
<cl:Binding glue="false" kind="constraintConnection">
  <cl:Source>cpc.CheckPort</cl:Source>
  <cl:Target>pg.RandomizerConnectionConstraint</cl:Target>
  <cl:UsedService name="isConnectedPort">
    <cl:Arg>pg.Randomizer</cl:Arg>
  </cl:UsedService>
</cl:Binding>
```

**Contract-Component Description** As for architecture constraints, in CLACS metamodel (cf. Figure 2), architecture contracts, architecture decisions
and quality attributes (as specified in section 2) are represented as standard components. Figure 4 presents an example of a contract component which documents the portability quality attribute concretized by the façade architecture pattern (introduced in section 1). This component provides a `check()` service to verify the compliance of the architecture description of business components with the contract, and a `customize` service adapt the contract component to the application context. The different kinds of arguments that can be passed to the customization service are the degree of criticality of the quality attribute, its description, which represent (as stated in the previous section) excerpts of the non-functional requirements specification, and the different parameters of the inner constraint components. Some of the sub-components bound to the internal required ports (`description`, `degreeOfSatisfaction`, `ad`, `qa`, ...) are not shown in the figure for reasons of space limitation.

```
+ customizePortabilityFacade(ComponentInstance comp, Port port, int degreeOfCriticality, String qaDescription)
+ check()
```

```
+ customizeFacadeAD(Component comp, Port port)
+ check()
```

```
+ customizePortability(String qaDescription, int degreeOfCriticality)
+ getDescription()
+ getPortability()
```

```
+ isFacade(Component comp, Port port)
```

```
+ description
+ degreeOfSatisfaction
+ collidesWith
+ enhances

```

Fig. 4. Example of an Architecture Contract Component

### 3.4 Contract-Component Evaluation

Contract evaluation is based on the architecture description of the documented component-based application. Each contract should be validated when architecture design is finished. Then, after each evolution of the system, the contract is evaluated for the new architecture descriptions. It detects if architecture constraints are not satisfied. This case means that the architecture decision associated to this constraint has been affected. It notifies then the developer with the information embedded in the contract. The role of this notification mechanism
and its benefits are outside of this paper’s scope. Tool-support for architecture contracts is detailed in the following section.

3.5 Contract-Component Reuse

Like business components, constraint or contract components are reusable entities. Each time a contract is defined for a given component in an application, this contract can be added in a component repository. Contract reuse requires a specific repository content organization. The repository could be organized according to the ISO 9126 quality model which introduces different levels of quality attributes such as: Reliability, Usability, Efficiency, Maintainability, Functionality and Portability. For example, the previous contract component (PortabilityByFacade) could be put in the Portability domain in the repository.

4 Tool-support for Architecture Contracts

We have developed the SCL language together with an Eclipse plugin which provides the following functionalities:

1. modeling architectures of business components in CLACS;
2. checking the architectural validity of these descriptions;
3. generating SCL code starting from these descriptions and loading it in its running environment;
4. modeling contract-components in CLACS;
5. checking constraint-components for the non-functional validation of CLACS component-based architectures.

In order to implement these functionalities, we used some existing Eclipse plugins, such as: the EMF\(^2\) (Eclipse Modeling Framework) module which allowed us to define an Ecore metamodel of CLACS, and the GMF\(^3\) (Graphical Modeling Framework) plugin to give a graphical dimension to the editor. By parsing files generated for a given architecture description, we enhanced this editor with some functions which check: i) whether the referenced interfaces in port definitions exist (in the same file or in an external one imported in the same directory), ii) whether the referenced component descriptors in component instances exist (in the same file or in an external one imported in the same directory); and iii) whether bindings link existing ports of existing component instances. We added to this editor a code generation feature that allows to generate SCL code starting from EMF models. Using the JET (Java Emitter Templates) Eclipse plugin\(^4\), we defined a set of templates to produce SCL portions of code for each abstraction of the CLACS metamodel. Currently, this code generator produces SCL code for business component descriptors. Architecture contracts are used at the architecture design stage and not at the implementation one. We are

\(^2\) Eclipse Foundation Website: http://www.eclipse.org/emf/

\(^3\) Eclipse Foundation Website: http://www.eclipse.org/gmf/

\(^4\) Eclipse Foundation Website: http://www.eclipse.org/modeling/m2t/?project=jet#jet
working now on the generation of SCL code representing contract-components. In the listing below, we give an excerpt of the SCL code generated for the example introduced in the previous section (PasswordManager component in figure 3).

```
(SclBuilder new: #PasswordManager
category: 'PasswordManager_category')
requiredPorts:
  #IRandomizer->#(#getRandomNumber));
providedPorts:
  #IGenerator->#(#generatePwd:).
```

As in UML modeling tools, only skeletons of SCL applications are generated; services’ body should be implemented by programmers (the "To be completed" parts in the example). CLACS modeling tool provide a functionality for loading the completed code in Squeak, which is a Smalltalk development environment where an SCL interpreter has been integrated.

At architecture design stage, the evaluation of constraint-components uses the Eclipse OCL plugin\(^5\). In CLACS modeling tool, when a developer launches the constraint checking, model files are parsed to identify constraintConnection bindings. The constraint-component descriptors are then parsed to extract the OCL constraint to be evaluated. Then, the OCL plugin is used for the evaluation of the extracted constraints by providing the necessary context components.

For contract checking, we developed, in the near past, with an industrial partner AURES [19] (ArchitectURe Evolution aSsistant). This tool supports Acme [10], Fractal [2] and Component-based Web applications [12] architecture descriptions. Since we proposed a new model of architecture contracts in section 2, this tool cannot be directly integrated in the new plugin. However, this integration only requires the reviewing of the evolution assistance algorithm.

5 Related Work

In Krueger’s taxonomy of software reuse processes [13], the author identifies five activities: classification, abstraction, selection, specialization and integration. In our work, “classification” relates to the organization of architecture contracts into domains corresponding to the quality attributes implemented by these contracts. “Abstraction” is addressed by providing interfaces to constraints and detailed information about quality attributes. “Selection” relates to the activity of choosing an architecture contract after querying the repository. “Specialization” is the customization task of contracts, which are white-box artifacts, during their reuse. “Integration” is addressed by composition.

Non-functional requirements (NFRs) modeling (elicitation in general) has been addressed in several works by Mylopoulos et al. [14, 4]. In [17], their work focused on NFRs reuse. Quality requirements are modeled as “softgoals” which are linked to functionalities (“goals”) modeled in graphs. The authors introduce the Q7 language for expressing the basic knowledge for software reuse. Based on this language, they proposed a process which allows the selection of existing quality softgoals from existing graphs, and the specialization and the integration of

\(^5\) Eclipse Foundation Website: http://www.eclipse.org/modeling/mdt/
the selected quality softgoals in a targeted goal graph (graph of functionalities). Contrarily to this approach where quality attributes are attached to functions and relationships between them are identified, in our work, quality attributes are defined at component-level and not at function-level. Functions (services in CLACS) are parts of architecture descriptions and architecture decisions are parts of quality attribute documentation. These decisions bridge thus the gap between software functional elements and quality attributes and represent the core of the reuse and assembly mechanism.

In [7], the authors present ConFract, a contracting system for Fractal [2] component model. These contracts are defined for hierarchical components and address the functional and behavioral aspect of component description and assembly. They are described using a language called CCL-J which is inspired from OCL and enhanced with Fractal and object-oriented abstractions. These contracts are checked during dynamic reconfiguration (component replacement, for example) of Fractal applications. They ensure the assembly consistency at different hierarchy levels (interface, internal composition, external composition and library elements). As precised above, the ConFract system deals with functional aspects of components and not with non-functional ones as in our work. Moreover, in their work, the authors do not address reuse, parameterization and composition of contracts. They focus on contract description, negotiation and checking of contracts dynamically on Fractal component instances.

Some existing ADLs provide constraint languages such as Acme [10] which is a representative example. It provides Armani, a first-order predicate language which allows the description of architecture constraints: invariants and heuristics. Invariants should not be violated, while heuristics should be observed but can be selectively violated. Constraints in Acme and in all other ADLs in the literature do not represent first-class entities for reuse or composition, whereas CLACS defines these features not only at the architecture constraint level but also at a higher level of abstraction: the contract-level.

Design pattern schemas [11] and component specification patterns [1] are descriptions which allow the generation of OCL constraints in a given context (for class models in the first paper and for software component specifications in the second). These descriptions define templates of OCL constraints with some parameters which are fixed during the instanciation 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 [11], model elements are UML class entities and in [1], constraints target the functional (behavioral) aspect of components.

At the best of our knowledge, there is no work in the literature which, at the same time, addresses component software quality documentation reuse through parameterized (non-functional) model elements, and deals with formal documentation composition using connectors as in business (functional) model elements.
6 Conclusion: Synthesis and Future Work

According to the OMG’s Reusable Assets Specification, “reusable assets” are artifacts that provide a solution to a recurrent problem for a given context [15]. In this paper, we presented architecture contracts as recurrent non-functional solutions to recurrent documentation problems to address the customizability and reusability issues presented in the introduction. Architecture contracts are “white-box” assets that can be customized for a given application context. “Variability points” represent the architectural elements to be constrained and the variable quality attributes’ properties (descriptions and degrees of criticality). “Rules for usage” represent component assembly principles, which are based in this work on binding construction and thus on traditional interface-matching.

Sometimes, defined manually (from scratch) this kind of architecture decisions’ documentation is complex, error-prone and time-consuming. Having a means to define such documentations by hierarchical composition of constraints (answering to the composability issue) is beneficial for two accounts. First, by decomposing the models of architecture decisions in several small interfaced documentation parts, a common repository of reusable (parametrized) assets is provided for the community; and second, this is a logical way of doing in the continuum of artifact development in component-based software engineering. The obtained development process starts with component architecture design and documentation with CLACS and architecture contracts, and ends with component implementation and execution with SCL and its runtime environment.

At the conceptual level, we plan in our perspectives to enrich the component-based software evolution assistance with a support to this new model of contracts. Introducing the refined linking mechanisms through degrees and relationships between quality attributes will inevitably enhance the evolution assistance algorithm and more particularly the notification reports. In the past, we conducted an industrial experiment on the use of architectural contracts in component-based software documentation and evolution [19]. Our aim in the future is to build a wide repository of architecture contracts and make it available for component-based (or other development paradigms) stakeholders. We plan then to provide a means to enrich this repository with architecture non-functional documentation defined in other (non-formal) formats [5, 20].

At the tool level, we plan in the near future to work on contract component code generation. This will help to check architecture contracts at the evolution stage on implementation artifacts (SCL code). We believe that SCL language provides the necessary expressiveness. The evaluation of these contracts on SCL code have thus to be implemented within SCL interpretation environment. This may even allow developers to beneficiate from the use of contracts during dynamic (runtime) evolution of SCL components. Another tool level issue that we will dealing with in the future, is to develop the component repository support-tool for constraint and contract components. This can be achieved by extracting information provided in CLACS descriptions to build some high-level documents which could be processed by some existing indexation techniques to build access points for contract components.
References