Component-based Specification of Software Architecture Constraints

Chouki Tibermacine
LIRMM, CNRS and
Montpellier II University
France
tibermacin@lirmm.fr

Salah Sadou
VALORIA, Université
Bretagne-Sud, Vannes
France
sadou@univ-ubs.fr

Christophe Dony
LIRMM, CNRS and
Montpellier II University
France
dony@lirmm.fr

Luc Fabresse
École des mines de Douai
France
fabresse@mines-douai.fr

ABSTRACT

Component-based software engineering provides for developers the ability to easily reuse and assemble software entities to build complex software. Component-based specification of software functionality has been and is largely addressed, however this is not yet the case for what concerns software non-functionality. In this paper, we propose a new way to express component-based software non-functional documentation, and we will focus more specifically on architecture constraints which formalize parts of architecture decisions, as executable, customizable, reusable and composable building blocks represented by components. Checking of architecture constraints is provided via service invocation through ports of a special kind of components, called constraint-components. The signatures of these checking services can be defined in required interfaces of business components, to document decisions taken while designing their architecture. They can also be part of other required interfaces of constraint components, making it possible to build higher-level or more complex constraints while reusing existing ones. We present an example of implementation of constraint components using, an ADL which is introduced in this paper. Architecture constraints can then be checked on the architecture of business components at design-time using the CLACS tool support, which has been implemented as an Eclipse plugin.

1. INTRODUCTION: CONTEXT AND MOTIVATION

In the last two decades different techniques for architecture decision documentation have been proposed [28, 14, 25, 13, 16, 26] in the software engineering literature. Several languages, anthologies and templates have been defined, ranging from text-based solutions to more formal ones. Text-based techniques [28, 14, 15, 25, 13, 12] are solutions that better organize architecture decision documentation at the design phase of the software lifecycle. They propose to developers to explicit, in a structured way, design decisions made during architecture description. In contrast, formal techniques [16, 3, 27] impose to developers to write decision descriptions as expressions in languages that can be fully processed by a support tool. These solutions, which apply particularly to some critical aspects of architecture decisions, provide a documentation less ambiguous and a support for (among others) automatic checking during architecture changes [26]. A part of such documentation is composed of architecture constraints. Examples of constraints include the choice of a particular architectural style or pattern, like the layered style. In a previous work [27] we presented a family of languages called ACL based on the Object Constraint Language (OCL [19]). These languages allow the expression of constraints at each phase of the component-based development process. In contrast to other constraint languages which work only with the ADL that has been proposed with [10, 3], ACL is parameterized with the language to which it will be associated. Indeed Armani [16] constraints, for example, are checkable only on architecture descriptions defined in Acme ADL [10]. ACL constraints can be associated to different ADLs and checking is supported through a transformation mechanism.

When defining component-based software architecture descriptions, architecture constraints are generally intended for the validation of some specific architectural elements (components, in most cases). This limits their potential reuse with architectural elements of other architecture descriptions. In addition, this kind of documentation often includes some parts which can be used individually for documenting parts of design decisions. Unfortunately, there is no means to extract these parts, to make them parametrized entities that can be factorized and used in different reuse contexts. We defend thus in this paper the idea of defining blocks of constraints as customizable and reusable entities. We observed that in the literature, these issues have not been addressed deeply. To doing so, we turned towards component-based software development.

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 a given context. There are different methods to reach customizability. One of the most known techniques used for this purpose is parametrization. Indeed defining parameters in the signature of a given software entity allows the developer to customize the software entity behavior according to the passed arguments.

The goal of the work presented in this paper is to propose a way to build basic 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. In synthesis, we aim 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 identify and illustrate *via* an example, the problems that are tackled in this paper. Section 3 makes an overview of the contribution of our work. In section 4, we first present CLACS, the ADL we built for the SCL [8] 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[1]. In Section 5, we illustrate through an example the application of the approach proposed in this paper. Section 6 introduces the prototype tool developed for implementing our proposals. Before concluding and presenting the future work at the end of this paper, we make an overview of the related works.

---

[1]The goal of our work in this paper is not to present ACL as an architecture constraint language parametrized by different ADLs (as indicated previously). The focus is on its use with CLACS.

2. PROBLEM STATEMENT BY EXAMPLE

The following example uses an existing proposition [27] and illustrates the problems we want to handle. The constraints below are defined in ACL (Architecture Constraint Language [27]), which is a slightly modified version of the OMG's Object Constraint Language. The first constraint role is to check whether an architecture conforms to the pipeline architectural style [24]. The constraint is here applied in the *context* of (*i. e.* applies to) a component named ACS (*Access Control System*).

```
(01) -- Each subcomponent's port is either input
(02) -- or output
(03) context ACS:CompositeComponent inv:
(04) ACS.subComponent.port->forAll(p:Port |
(05) (p.kind='Input') or (p.kind = 'Output'))
(06) and
(07) -- Each connector should define two roles,
(08) a sink and a source role
(09) ACS.configuration.binding.role.connector->asSet()
(10) ->forAll(con:Connector | (con.role->size() = 2)
(11) and ((con.role->exists(r:Role | (r.kind = 'Source')
(12) or (r.kind = 'Sink'))))
(13) and
(14) -- Each connector should bind two components
(15) -- (input bound to sink and output to source)
(16) ACS.configuration.binding.role.connector->asSet()
(17) ->forAll(con:Connector|con.role->forAll(r:Role |
(18) ACS.subComponent ->exists(com:Component|com
(19) .port->exists(p:Port|(r in ACS.configuration.binding)
(20) and ((p.kind = 'Input') and (r.kind = 'Sink'))
(21) or ((p.kind = 'Output') and (r.kind = 'Source'))))))
(22) and
(23) -- The graph representing the configuration
(24) -- should be connected
(25) ACS.configuration.isConnected
(26) and
(27) -- The graph should contain a number of arcs
(28) -- equal to the number of vertices - 1
(29) ACS.configuration.binding.role.connector->asSet()
(30) ->size() = ACS.subComponent->size()-1
(31) and
(32) -- The graph should represent a list
(33) ACS.subComponent->forAll(com:Component|
(34) (com.port->size() = 2)
(35) and (com.port->exists(p:Port|p.kind = 'Input'))
(36) and (com.port->exists(p:Port|p.kind = 'Output')))
```

We can firstly observe in this example that the constraint is composed of many "independent" sub-parts that are assembled together via the and logical operator (Lines 06, 13, 22, 26 and 31). All the subparts assembled together represent the pipeline architecture style; but it is easy to observe that some of these subparts have their own consistent semantics. For example, the sub-part in lines 23 to 25 checks that the set of all ACS subcomponents bindings is a connected graph and the sub-part in Lines 32 to 36 checks that this graph is represented by a list. These two sub-parts could meaningfully be reused independently from the others either alone or within another more global constraint to check whether an architecture configuration is organized as a connected graph or as a list. They represent reusable entities that can be named and placed on a repository in order to be checked out by developers of new architecture descriptions to formalize their design choices.

On this first example, we can secondly observe that the constraint is expressed in a context-independent way : it simply checks that "**all**" ACS sub-components, whatever their number or their names and without referencing them explicitly, are "pipelined". This is not always the case. In many situations, constraints have to make explicit references to some attributes, sub-parts or sub-components of the architecture they control. The example of the listing below, also written in ACL, illustrates that case; it presents a constraint that formalizes the *facade* architecture pattern -with analogy to façade objects [9]-. The context is the composite component named `ACS`. This ACL expression states that the `DataManagement` provided port of ACS must be bound internally to one and only one other port (Line 06). The latter port corresponds to the provided port of `DataAdminRetrieval` component (Line 09). This subcomponent represents the *facade* element in the architecture. All communications from clients to ACS data management services transit by this component.

```
(01) context ACS:CompositeComponent inv:
(02) let boundToDataManagement:Bag=ACS.port
(03) ->select(p:Port|p.interface.kind = 'Provided'
(04) and p.name = 'DataManagement').binding
(05) in
(06) ((boundToDataManagement->size() = 1)
(07) and (boundToDataManagement.interface
(08) ->select(i:Interface|i.kind = 'Provided').port
(09) .component.name->includes('DataAdminRetrieval')))
```

We can observe that, in contrast to the example of the first listing, this constraint contains identifiers that reference specific elements in the architecture description (the subcomponent `DataAdminRetrieval` in Line 09 and the port `DataManagement` in Line 04). This constraint is therefore not reusable in other contexts without editing. To give it a syntactic signature and to reference the architecture elements as parameters in this signature would clearly make this constraint more generic and reusable in other contexts.

ACL and existing languages or tools (see related work section) for expressing architectural constraints do not give yet optimal answers to the issues we have identified above. The proposal described in the following sections aims at proposing a better solution to reach these goals.

3. GENERAL APPROACH
The examples and the discussion of the previous section bring to the fore the following challenges:

- **Reusability**: constraints should be specified as reusable named entities easily referenced from repositories.

- **Customizability**: to be applicable in different contexts, architecture constraints should be parameterized by any elements that constitute architectures. The constraint in the second example should thus have two parameters: one of type `Port` and one of type `Component`.

- **Composability**: the first example showed that complex constraints are beneficially built as combination

of other ones. We argue, more generally, that all architecture constraints could be embedded in components that export services for constraint checking and for composability.

In order to answer these challenges, we propose an approach where constraints are embedded in a special kind of software components (see the following section for details about their specification). These components provide services via ports for checking these constraints at design time. These components do not exist at implementation or execution time. They are assembled with business (functional[2]) components which require the checking of constraints on their internal architecture description. In the proposed approach, each time the architect wants to check a given constraint on a business component, she/he should specify a new kind of required (non-functional or constraint) port that is removed when the business component is implemented. This required port is then connected to a provided port of a constraint component. The architect can add multiple required constraint ports if she/he needs to check several constraints on the internal architecture of her/his business components.

A provided administration port is integrated automatically to business components which have required constraint ports. This port allows to check all the constraints that are connected to these components. Each business component that has an administration port is instantiated at design time for using its administration port and thus for checking the constraints that are associated to the component. We chose to use a port for the checking of non-functional properties (structural constraints) of a component in order to ensure consistency of using components in the design stage.

For the first example we have presented in the previous section, we would obtain, using our approach, a set of constraint-components representing the different parts of this constraint (see Figure 1). Each component provides a single port for checking a part of the constraint through a service. On the other hand, the business components contain one required port by a non-functional property needed. To meet the non-functional property (pipeline) of the business component from our example, all obtained constraint components should be organized in a composite constraint component, which will provide an interface to check that the component conforms to the pipeline architecture style. Thus, binding a required port of a business component to a provided port of constraint component means that the latter is responsible for checking the validity of the concerned non-functional property. As for the provided administration port, in business components, the ports requiring constraint components exist only at design-time.

The services provided by constraint components can be parametrized by some architectural elements used in the constraint. This is illustrated in the bottom of Figure 1. In this way constraints become more generic and can be reused in different contexts (with different business components). Besides this, constraints become a modeling element that can be connected together to build more complex constraint

---

[2]Constraint components can be considered as non-functional components here.

Unstructured Textual Constraints - - - - - - - - - - - - → **Constraint Components**

```
(01) -- Each subcomponent's port is either input
(02) -- or output
(03) context ACS:CompositeComponent inv:
(04) ACS.subComponent.port->forAll(p:Port |
(05) (p.kind='Input') or (p.kind = 'Output'))
```
boolean arePortsOfKind(PortKind[] kinds)

service

constraint
a:A

component

```
(06) and
(07) -- Each connector should define two roles,
(08) a sink and a source role
(09) ACS.configuration.binding.role.connector->asSet()
(10) ->forAll(con:Connector | (con.role->size() = 2)
(11) and ((con.role->exists(r:Role | (r.kind = 'Source')
(12) or (r.kind = 'Sink'))))
```
boolean areRolesOfKind(RoleKind[] kinds)

constraint
b:B

binding

```
(13) and
(14) -- Each connector should bind two components
(15) -- (input bound to sink and output to source)
(16) ACS.configuration.binding.role.connector->asSet()
(17) ->forAll(con:Connector|con.role->forAll(r:Role |
(18) ACS.subComponent ->exists(com:Component|com
(19) .port->exists(p:Port|(r in ACS.configuration.binding)
(20) and ((p.kind = 'Input') and (r.kind = 'Sink'))
(21) or ((p.kind = 'Output') and (r.kind = 'Source'))))))
(22) and
...
```
boolean inToSinkOutToSource()

constraint
c:C

parameterization

```
(01) context ACS:CompositeComponent inv:
(02) let boundToDataManagement:Bag=ACS.port
(03) ->select(p:Port|p.interface.kind = 'Provided'
(04) and p.name = 'DataManagement').binding
(05) in
(06) ((boundToDataManagement->size() = 1)
(07) and (boundToDataManagement.interface
(08) ->select(i:Interface|i.kind = 'Provided').port
(09) .component.name->includes('DataAdminRetrieval')))
```
boolean isFacade(aPort:Port,aSubComp:Component)
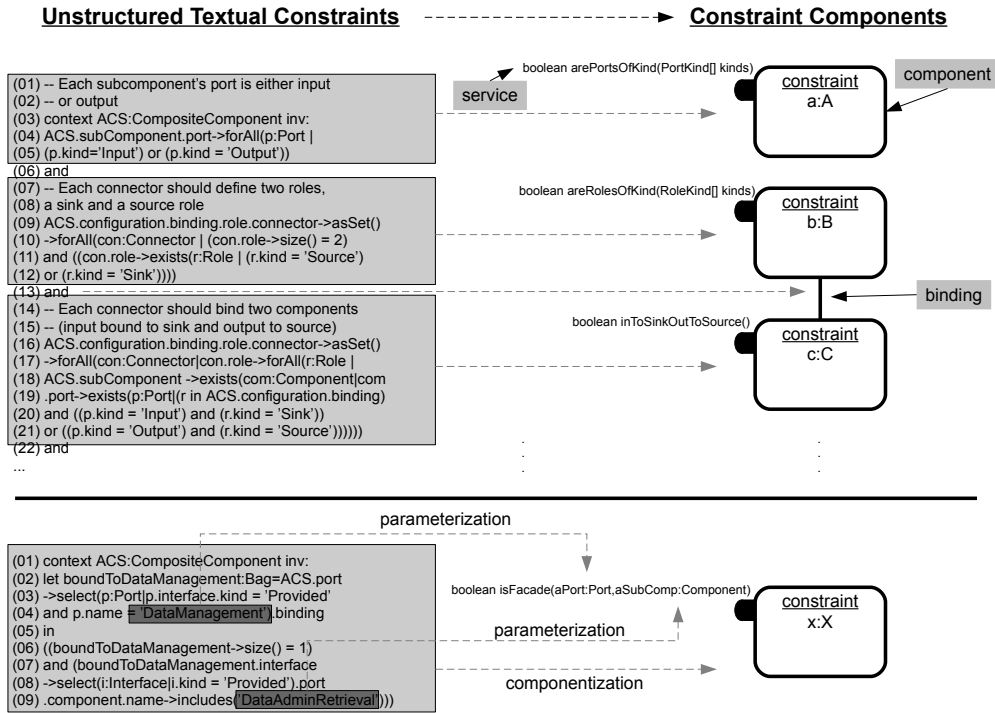
parameterization

componentization

constraint
x:X

Figure 1: Component-based Specification of Constraints

components (as illustrated through the binding at the right of Figure 1).

4. ARCHITECTURE CONSTRAINTS AS COMPONENTS

In this section, we present a new constraint component model as a means to describe customizable, reusable and composable architectural constraints. Our solution is embedded into an operational software suite (CLACS-SCL) made of an Architecture Description Language (ADL) called CLACS, and of a component-oriented programming language named SCL [8]. CLACS is as a modeling alternative for SCL. SCL is a pure component-oriented language in which components are first-class entities, connectors and primitive types are components too, and argument passing is done using an original component connection mechanism [8]. Using that suite, component-based application architectures can be graphically composed in CLACS and deployed in SCL for execution; architecture constraints can be defined, composed and executed in CLACS. Building that suite has been motivated by the following considerations:

1. The SCL language is based on a generic component metamodel. This allows us to implement our solution in any specific component model.

2. CLACS allows SCL code generation, which is very helpful to build executable components, in order to experiment our approach.

4.1 CLACS Component Metamodel

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. Figure 2 shows a metamodel of CLACS constraint components and SCL business components.

In CLACS and SCL, a component is an instance of a component descriptor. A component has ports, which are defined by three properties: i) a direction (required or provided), ii) a visibility (internal: the port is private to the component that owns it and can only be bound internally for example to delegate to an inner component, or external: the port can be bound to other external components), and iii) an interface which specifies the port type. An Interface is either of "business" kind or of "constraint checking" kind. Interfaces are a collection of signatures and a signature describes a service, its formal parameters and its return type. Standard bindings (or connections) link ports of components of the same hierarchical level. As in UML, delegation bindings link components to their subcomponents. More sophisticated bindings can be defined using connector components that can be seen as adaptors. A connector receives service invocations through its *source* port and transmits them through its *target* ports by executing the *glue* or adaptation code.

The differences between business and constraint components are expressed via the following elements (see again Figure 2): the `kind` meta-attribute in `ComponentDescriptor`, `Interface` and `Binding` meta-classes. Thus, the `kind` meta-attribute takes the value `constraint` for constraint components, and
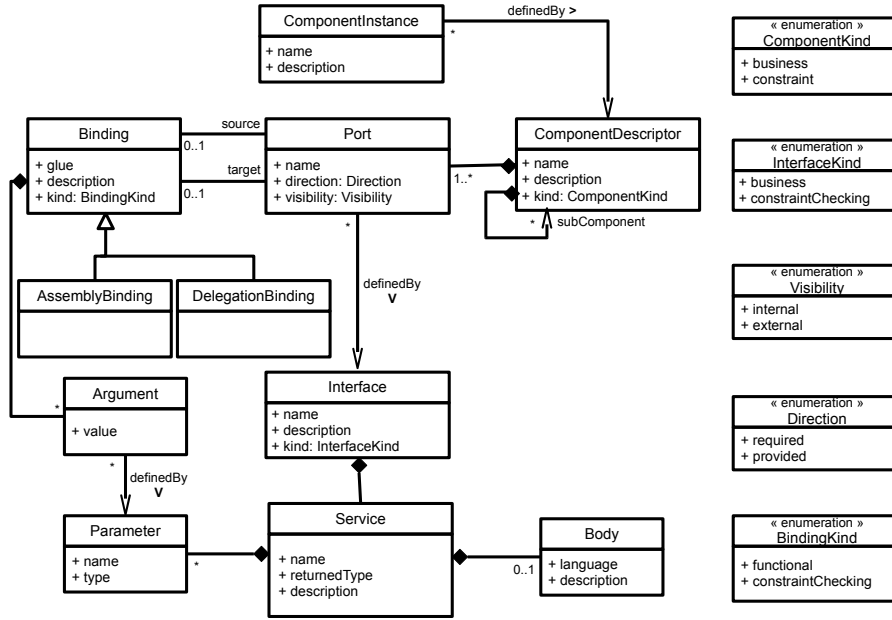
Figure 2: CLACS Metamodel

ComponentInstance
+ name
+ description

definedBy >

« enumeration »
ComponentKind
+ business
+ constraint

Binding
+ glue
+ description
+ kind: BindingKind

source

Port
+ name
+ direction: Direction
+ visibility: Visibility

target

ComponentDescriptor
+ name
+ description
+ kind: ComponentKind

subComponent

« enumeration »
InterfaceKind
+ business
+ constraintChecking

AssemblyBinding

DelegationBinding

definedBy

« enumeration »
Visibility
+ internal
+ external

Argument
+ value

Interface
+ name
+ description
+ kind: InterfaceKind

definedBy

« enumeration »
Direction
+ required
+ provided

Parameter
+ name
+ type

Service
+ name
+ returnedType
+ description

Body
+ language
+ description

« enumeration »
BindingKind
+ functional
+ constraintChecking

business for the other components. In addition, the implementation of services is different from a constraint component to a business one. In the latter case, services represent traditional operations with a body containing the SCL code implementing the business logic. In constraint-components, the body contains the ACL code of the constraint to be checked. Bindings between components can be of kind "functional" if they connect business components, or of kind "constraint checking" if they connect a business component to a constraint component or if they connect two constraint components together.

4.2 Specifying Constraint-Components

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. This sub-section proposes an example of a constraint-component definition and the following sub-section an example of such a connection.

Figure 3 depicts the definition of a simple constraint-component descriptor. This component allows to check the *Facade* pattern presented in the previous section. This descriptor can be instantiated in a given architecture description. Each *extitFacade checker*, instance of this descriptor, owns one provided port named Checking that exports a constraint checking service having the following signature : boolean isFacade(aPort:Port,aSubComp:Component). Each *extitFacade checker* can then be connected, through that checking port, to any business component requiring this constraint service.

When invoked within our modeling environment, a constraint-component provided service returns true if the architecture
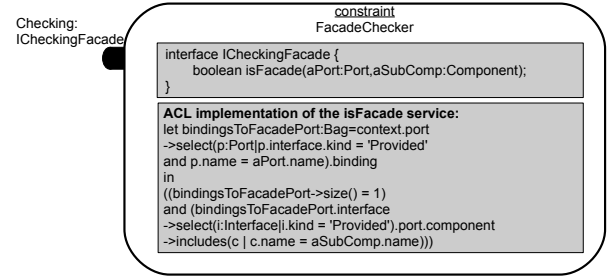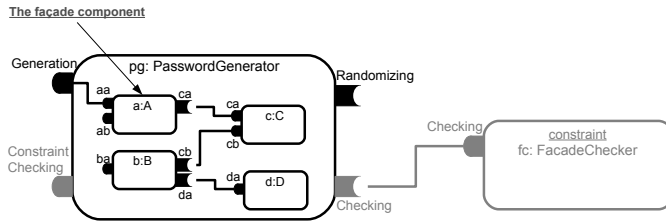
Checking:
ICheckingFacade

constraint
FacadeChecker

```
interface ICheckingFacade {
    boolean isFacade(aPort:Port,aSubComp:Component);
}
```

ACL implementation of the isFacade service:
```
let bindingsToFacadePort:Bag=context.port
->select(p:Port|p.interface.kind = 'Provided'
and p.name = aPort.name).binding
in
((bindingsToFacadePort->size() = 1)
and (bindingsToFacadePort.interface
->select(i:Interface|i.kind = 'Provided').port.component
->includes(c | c.name = aSubComp.name)))
```

Figure 3: Example of a primitive Constraint-Component

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 (see again Figure 3), 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.

4.3 Connecting Constraints to Business Components

Figure 4 presents two connected sub-components: a PasswordGenerator named pg and a FacadeChecker (as defined in the previous section) named fc. A PasswordGenerator uses random numbers to automatically generate passwords. It has a (business) provided port Generation and a (business) required port (Randomizing). At design stage, it also holds another provided port (ConstraintChecking) and an-

Figure 4: Example of a Constraint-Component Assembled with a Business One

other required (constraint) port (`Checking`). Through the latter it is connected to the `Checking` provided port of the `FacadeChecker` constraint component. Both ports are drawn in gray in the figure, to indicate their temporary nature (exist only at design-time) comparatively to the other business ports.

The semantics of the constraint checking binding is that the `PasswordGenerator` designer wants to check that the internal organization of this component conforms to the facade architecture pattern.

The binding in the graphical representation in Figure 4 is serialized as shown in the code of the listing below.

```
(01) <cl:AssemblyBinding glue="false" kind="constraintChecking">
(02)    <cl:Source>fc.Checking</cl:Source>
(03)    <cl:Target>pg.Checking</cl:Target>
(04)    <cl:UsedService name="isFacade">
(05)       <cl:Arg>aa</cl:Arg>
(06)       <cl:Arg>a</cl:Arg>
(07)    </cl:UsedService>
(08) </cl:AssemblyBinding>
```

Note how are passed the arguments for the `isFacade` service of the `FacadeChecker`, the port `aa` and the component `a` (Lines 05 and 06 in the following code). We can observe in Figure 4 that component `a:A` in the modeled architecture plays the role of a extitFacade (the only sub-component of `PasswordGenerator` that is connected by its provided port to the external ports of its encompassing component).

4.4 Specifying Constraint-Component Requirements

Before detailing in the following subsection how constraint-components can be assembled, we expose in Figure 5 a constraint-component (`InputSinkOutputSourceRestrictor`) with a required port. This component represents the checking of a part of the constraint that formalizes the Pipeline architecture style. In this component descriptor there are two interface specifications. The first is named `IInSinkOutSrc-Restriction` and represents the type of the provided port, and the second (put in the dashed box in the figure) is named `IPortsRestriction` specifies a type for the required port `IO_PortRestriction`.

The required port is used in the body of the constraint (see again Figure 5) to invoke the service `arePortsOfKind`. We can observe in Figure 5 (underlined expression) how the ser-
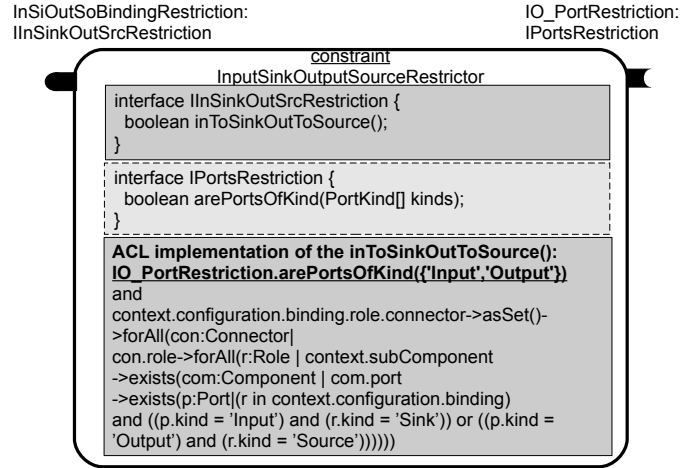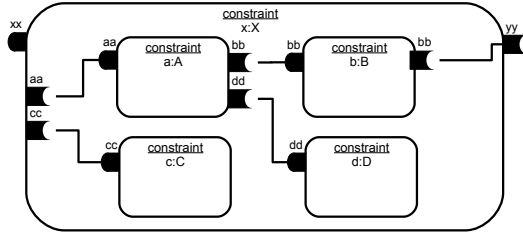

Figure 5: Example of a Constraint-Component with a Required Port

vice invocation is associated with the remaining of the constraint provided by the constraint-component `InputSinkOutputSourceRestrictor` using an `and` operator.

4.5 Composing Constraint-Components

A constraint-component can be assembled with (or bound to) other constraint-components to build more complex ones. As in UML and many other component models, bindings can be either of type `Delegation` or `Assembly`. Delegation bindings of kind "`functional`" are used exclusively between business components (this is not discussed in this section). A delegation binding of kind "`constraintChecking`" is used for building a composite constraint-component starting from other constraint-components. This binding allows the composite to delegate the checking of part of the architecture constraint to its subcomponents. Figure 6 illustrates such a composition. The composite component (x:X in the Figure) asks its subcomponents a:A and c:C, which are connected to it by using internal required ports (see the left part of Figure 6), to make a constraint checking. The checking results are used in the service provided by the port `xx` of the composite component. These can be combined by an `and`, an `or` or any other logical operator in the provided service specification. In this service specification we can invoke the service (`serviceA()`) provided by the port `aa` of component a:A and the service (`serviceC()`) provided by the port `cc` of component c:C in the same way as stated in the previous subsection: `aa.serviceA()` and `cc.serviceC()`. Arguments are passed in the binding specification as indicated previously. A delegation binding can link a required external port of a subcomponent to a required external port of its composite component (see the right side of Figure 6). A constraint checking in this case is expected from another constraint-component connected with the composite component.

Assembly bindings link a required port of a given component to the provided port of another component of the same hierarchical level (for example, the components a:A and b:B). In Figure 6, we defined two assembly bindings, one between a:A and b:B, and the other between a:A and d:D. In the same

Figure 6: Example of a Constraint-Component Composition

way as for delegation bindings, it is in the specification of the service provided by the component a:A that we define how the results returned by the other components (b:B and d:D) are combined. In this example, we consider that a:A specifies its own constraint (represented by A) which is composed with the constraints provided by B and D. Here, x:X specifies only a composition of other constraint-components. It states that the constraint provided by a:A must be respected and the constraint implemented by c:C must not. This is equivalent to: `A and (not C)`.

5. CONSTRAINT-COMPONENTS BY EXAMPLE

We show in this section how our approach can be applied. We demonstrate how constraint-components can be used to model the example of an architecture constraint introduced in Section 2. Figure 7 illustrates an assembly of constraint-components to build a "Pipeline Constraint Checker": a component able to check whether an architecture conforms to the pipeline architectural style.

This assembly contains six components which represent the different parts of the constraint. These components are connected together using different kinds of bindings. They are encapsulated in a single composite component `Pipeline-ConstraintChecker`. This composite component delegates first the checking to the component `InputSinkOutput-SourceRestrictor`. It then asks the other three components to check the other parts of the constraint.

The constraint component `InputSinkOutputSourceRestrictor` checks first if the ports of the business component (connected to the `PipelineConstraintChecker`) are of some specific kind (`Input` and `Output`) by requesting the component `PortsKindRestrictor` to check this part of the constraint. A binding is thus defined between `InputSinkOutputSource-Restrictor` and `PortsKindRestrictor` through the ports `IO_PortRestriction`. This is an assembly binding, which is represented by the following description:

```
(01) <cl:AssemblyBinding glue="false" kind="constraintChecking">
(02)   <cl:Source>isisosbr.IO_PortRestriction</cl:Source>
(03)   <cl:Target>iopr.IO_PortRestriction</cl:Target>
(04)   <cl:UsedService name="arePortsOfKind">
(05)     <cl:Arg>Input</cl:Arg>
(06)     <cl:Arg>Output</cl:Arg>
(07)   </cl:UsedService>
(08) </cl:AssemblyBinding>
```

As we can observe in this listing (lines 5 and 6), parameters of the `arePortsOfKind()` service are initialized by the two values `Input` and `Output`. In this example, we need to check if ports are only of these two kind[3].

In the same way, a binding is defined between `InputSinkOutputSourceRestrictor` and `RolesKindRestrictor` in order to ask the latter component to check if the roles of connectors of the business component (connected to this constraint component) are of kind `Source` and `Sink`. The passed arguments to the service `areRolesOfKind()` are thus `Source` and `Sink`[4].

Then, the constraint component `InputSinkOutputSource-Restrictor` ends the execution of its provided service by checking the constraint stating that the input port of all sub-components of the business component should be connected to sink roles and output ports to source roles. This is shown in the component at the high left corner of Figure 7. We note the presence in the implementation of the service provided by this component, the invocations to the services provided by `PortsKindRestrictor` and `RolesKindRestrictor` (see the underlined expressions).

The delegation bindings are defined between the composite component `PipelineConstraintChecker` and its subcomponents. In Figure 7, there are four delegation bindings. As the constraint provided by `PipelineChecker` combines the different subcomponents with an `and` logical operator, all the constraints have to be checked for a `true` value.

As described, the constraint component, presented in this section, can be reused by any other composite component to check that its internal structure conforms with the *pipeline* architectural style.

6. TOOL SUPPORT FOR CONSTRAINT COMPONENT MODELING & CHECKING

We have developed an operational software suite, called CLACS-SCL, 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. modeling constraint-components in CLACS;

4. checking constraint-components;

5. generating SCL code starting from these descriptions and loading it in its running environment;

In order to implement these functionalities, we have used some existing Eclipse plugins, which are: the EMF[5] (Eclipse Modeling Framework) module which allowed us to define an

---

[3]In other contexts, we may need to check if they are of kind `InputOutput`, for example.

[4]With other business components, the parameters of this constraint can be initialized by other values, like `Trigger` and `Listener` for example.
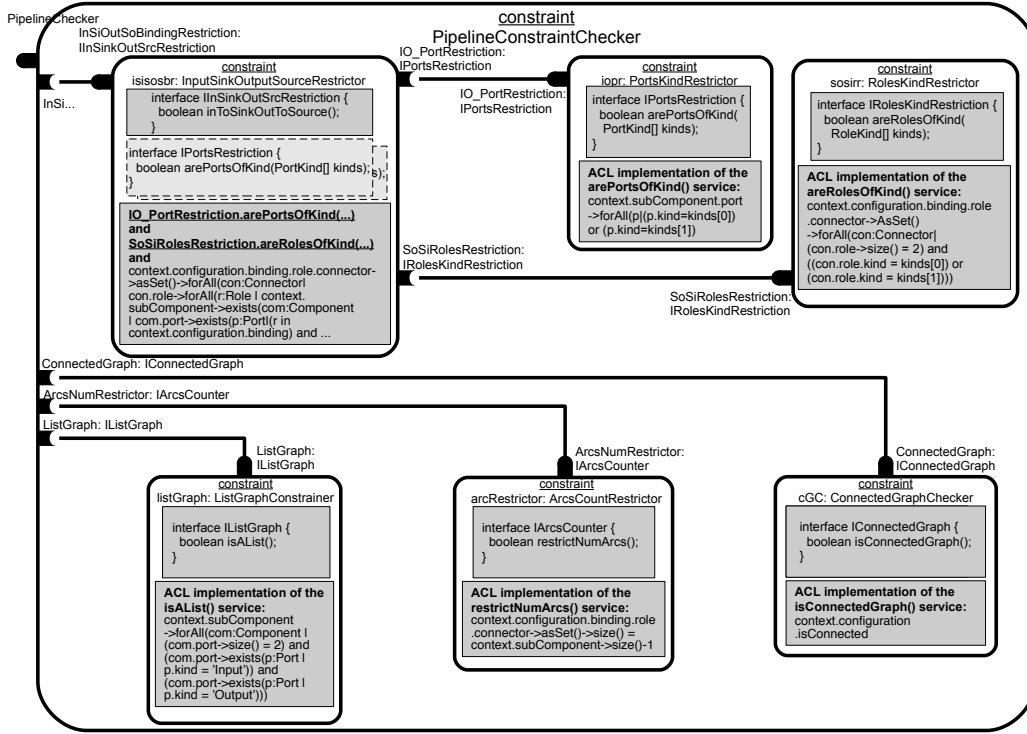
Figure 7: Example of a Constraint-Component Assembly

Ecore metamodel of CLACS to generate an editor, and the GMF[6] (Graphical Modeling Framework) plugin to give a graphical dimension to the editor. By parsing the files generated for a given architecture description, this editor allows an architect to check (item 2 above): i) if the referenced interfaces in port definitions exist (in the same file or in an external one imported in the same directory), ii) if the referenced component descriptors in component instances exist (in the same file or in an external one imported in the same directory); and iii) if the specified bindings link existing ports of existing component instances.

At architecture design stage, the interpretation of constraint-components is built upon the Eclipse OCL interpreter plugin[7]. Based on the XML-based CLACS architecture descriptions, this interpretation returns the (boolean) evaluation result of the architecture constraints encapsulated in constraint-components.

For more information about this tool, the reader is invited to visit the following website: http://code.google.com/p/clacs/

7. RELATED WORK

Different existing ADLs embed constraint languages. Acme [10] and Wright [3] are two representative examples of them. Wright is an ADL used to formalize architecture descriptions and more particularly connector specifications. This language provides the necessary language constructs for defining architecture constraints. The following example states in Wright that the architecture description should have a star topology.

$\exists center : Components \bullet$
$\forall c : Connectors \bullet \exists r : Role; p : Port \mid ((center, p), (c, r)) \in Attachments$
$\wedge \forall c : Components \bullet \exists cn : Connectors; r : Role; p : Port \mid ((c, p), (cn, r)) \in Attachments$

The first predicate indicates that there should exist one component (`center`) which is attached to all connectors of the architecture. The second predicate states that all components should be attached to a connector. This global constraint checks that every component in the architecture is connected to a single component representing the star's center.

Acme integrates *Armani* [16], 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. The following example shows an invariant and a heuristic in Armani.

```
Invariant Forall c1,c2 : component in sys.Components |
  Exists conn : connector in sys.Connectors |
  Attached(c1,conn) and Attached(c2,conn);
Heuristic Size(Ports) <= 5;
```

The invariant states that all components should be connected together. The set of bindings forms thus a connected graph. The heuristic indicates that the total number of ports should be less than or equal to 5.

As we can observe in the previous examples, constraints in Acme and Wright do not represent first-class entities for composition. Assembling Armani or Wright constraints is not straightforward, because there are no language constructs provided for this goal. In fact, these languages were not originally designed for this aim. In addition, the expressions presented above are generic examples of constraints. They do not apply to a specific context of a given architecture description. They represent 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 [11] and component specification patterns [2] 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 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 [11], model elements are UML class entities and in [1] constraints target the functional (behavioral) aspect of components.

Our proposal should also be compared with existing works on Quality of Service (QoS) specification and composition in the context of service-oriented architectures. [23] describes RBSLA, a language for describing service level agreements (SLAs). This language is based on RuleML[5] and allows the definition of predicates on the required QoS of a given service. As architectural constraints in CLACS, SLAs in this language formalize some descriptions that are usually specified as text documents. They can be composed to build more complex expressions in predicate logic. In addition, SLAs are saved as XML documents in the same way as our constraint-components are serialized in XMI format [20]. Other similar interesting languages include SLAng[6] and IBM's WSLA[7]. In SLAng, QoS constraints are defined in OCL. SLAng constraints apply on service models defined in EMOF [17] and limit the possible behaviors of services. In WSLA, QoS contracts can be specified as XML documents. The focus in this language is on the automatic monitoring of SLAs. In the same vein, SCA (Software Component Architecture) specification [22] proposed the SCA Policy Framework [21] as a way to describe constraints and QoS expectations from component designs. These are called policies and specify conditions under which service components run and interact. Originally, the constraint language presented in our paper has been designed to document architectural choices that answer to component's quality requirements [26]. Simi-

larly, in RBSLA, WSLA, SLAng and the SCA specification, constraints are related to services' quality. Nevertheless, the kinds of quality attributes addressed in these languages are not the same. QoS deals with runtime attributes (availability, confidentiality and performance, for example), however in our work we address static attributes (such as for example the portability attribute for the façade pattern and the maintainability attribute for the pipeline style of the examples of Section 2). In addition, these two kinds of non-functional documentation (SLAs and policies from one side and component-constraints from the other side) have not the same uses. SLAs and policies are contracts between service requestors and service providers. Constraint-components are contracts between component architects and component evolvers [26].

8. CONCLUSION AND FUTURE WORK

According to the OMG's *Reusable Assets Specification* [18], "reusable assets" are artifacts that provide a solution to a recurrent problem in a given context. In this paper, we presented architecture constraints as recurrent non-functional solutions to recurrent documentation problems, to address the customizability and reusability challenges presented in the introduction. Architecture constraints are "white-box" assets that can be customized for a given application context. "Variability points" represent the architectural elements to be constrained. "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 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[8]. The development process obtained in this work starts with component architecture design and documentation with CLACS, and ends with component implementation and execution with SCL and its runtime environment.

Our aim in the future is to build a repository of classified architecture constraints, and make it available for component-based software architects. 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 introspection components. Introspection capabilities, such as getting the list of sub-components or the ports of the business components, can be generated from the ADL's metamodel. These components can be customized by developers by adding checking services

---

for architecture constraints, as explained in this paper. This model will be enriched with some reflection capabilities, such as adding new ports or bindings to support dynamic architecture reconfiguration.

At the tool level, we plan in the near future to work on the creation of a component repository for constraint-components. This can be achieved by extracting information provided in CLACS descriptions to build some high-level documents which could be processed by existing indexation techniques in order to build access points for constraint-components.

9. REFERENCES

[1] J. Ackermann. Formal description of ocl specification patterns for behavioral specification of software components. In *MODELS Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, 2005.

[2] 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.

[3] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997.

[4] Eclipse. Java emitter templates (jet). Eclipse Board Web Site : http://www.eclipse.org/modeling/m2t/?project=jet#jet

[5] Eclipse. Eclipse modeling framework (emf). Eclipse Board Web Site : http://www.eclipse.org/modeling/emf/

[6] Eclipse. Graphical modeling framework (gmf). Eclipse Board Web Site : http://www.eclipse.org/modeling/gmf/

[7] Eclipse. Object constraint language plugin. Eclipse Board Web Site : http://www.eclipse.org/modeling/mdt/

[8] 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.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Sofware*. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc., 1995.

[10] 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.

[11] M. Giese and D. Larsson. Simplifying transformations of ocl constraints. In *Proc. of MODELS'05)*, Montego Bay, Jamaica, October 2005.

[12] A. Jansen, P. Avgeriou, and J. S. van der Ven. Enriching software architecture documentation. *Journal of Systems and Software*, 82(8):1232–1248, 2009.

[13] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proc. of WICSA'05*, 2005.

[14] P. Kruchten. An ontology of architectural design decisions in software intensive systems. In *Proceedings of the 2nd Groningen Workshop Software Variability*, pages 54–61, 2004.

[15] P. Kruchten, R. Capilla, and J. C. Duenas. The decision view's role in software architecture practice. *IEEE Software*, 26(2):36–42, 2009.

[16] R. T. Monroe. Capturing software architecture design expertise with armani. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2001.

[17] OMG. Meta object facility (mof) 2.0 core specification. OMG's Website: http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-15.pdf

[18] OMG. Reusable asset specification, v2.2. OMG's Website: http://www.omg.org/cgi-bin/doc?formal/2005-11-02

[19] OMG. Object constraint language specification, version 2.0. OMG's Website: http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf

[20] OMG. Meta object facility(mof) 2.0 xmi mapping specification, version 2.1.1. OMG's Website: http://www.omg.org/cgi-bin/doc?formal/07-12-01.pdf

[21] OSOA. Open soa. sca policy framework v1.00. http://osoa.org/display/Main/Service+Component+Architecture+Specifications

[22] OSOA. Open soa. service component architecture specifications. http://osoa.org/display/Main/Service+Component+Architecture+Specifications

[23] A. Paschke. Rbsla a declarative rule-based service level agreement language based on ruleml. In *Proc. of CIMCA-IAWTIC'06*, pages 308–314. IEEE CS, 2005.

[24] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[25] A. Tang, M. A. Babar, I. Gorton, and J. Han. A survey of the use and documentation of architecture design rationale. In *Proc. of WICSA'05*, Pittsburgh, Pennsylvania, USA, November 2005. IEEE CS.

[26] C. Tibermacine, R. Fleurquin, and S. Sadou. On-demand quality-oriented assistance in component-based software evolution. In *Proc. of CBSE'06*, pages 294–309, Vasteras, Sweden, June 2006. Springer LNCS.

[27] C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *In Journal of Systems and Software (JSS), Elsevier*, 2010.

[28] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, March/April 2005.