A three-level component model in component-based software development

Huaxi (Yulin) Zhang, Lei Zhang, Christelle Urtado, Sylvain Vauttier, Marianne Huchard

A Three-level Component Model in Component Based Software Development

Huaxi (Yulin) Zhang
Dept. Math/Info
Université Toulouse 2
Toulouse, France
Zhang@irit.fr

Lei Zhang
Research Center of Automation
Northeastern University
Shenyang, China
zl.org.cn@gmail.com

Christelle Urtado
LGI2P / EMA
Nîmes, France
Christelle.Urtado@mines-ales.fr

Sylvain Vauttier
LGI2P / EMA
Nîmes, France
Sylvain.Vauttier@mines-ales.fr

Marianne Huchard
Lirmm, Umr 5506
Cnrs and Univ. Montpellier 2
Montpellier, France
huchard@lirmm.fr

ABSTRACT

Component-based development promotes a software development process that focuses on component reuse. How to describe a desired component before searching in the repository? How to find an existing component that fulfills the required functionalities? How to capture the system personalization based on its constitutive components' customization? To answer these questions, this paper claims that components should be described using three different forms at three development stages: architecture specification, configuration and assembly. However, no architecture description language proposes such a detailed description for components that supports such a three step component-based development. This paper proposes a three-level ADL, named Dedal, that enables the explicit and separate definitions of component roles, component classes, and component instances.

Keywords

Component-based development, Software architecture, Architecture description language

1. INTRODUCTION

Component-based software development (CBSD) consists in two activities: the development of software components for reuse and the development of software applications by component reuse. The first activity can be managed by classical software development processes, with an analysis, a design and then a coding phase. The produced software modules, encapsulated as component classes, are then stored and indexed in repositories to be reused later on. The second activity corresponds to a more specific and still scarcely studied development process. We propose an architecture-centric development process that aims at defining the structure of an application as a set of reused components and a set of connections between them, using a dedicated Architecture Description Language (ADL). This process is structured in three steps, in which architecture definitions are gradually refined, from abstract to concrete representations.

1. After a classical analysis step, architecture specification first captures design decisions as ideal architectures imagined by architects to meet the requirements. Specifications do not describe complete component types but only component roles (usages). These roles are used to search for matching component classes in repositories. Specifications and roles are thus key concepts to effectively integrate component reuse in the development process.
2. Architecture configurations are then described to capture implementation decisions, as the architects select specific component classes from the repository to implement component roles.
3. Finally, architecture assemblies define how component instances are created and initialized to customize the deployment of architectures in different execution contexts.

Our process is supported by a three-level dedicated ADL, entitled Dedal, which enables the explicit and separate definitions of architecture specifications, configurations and assemblies. This way, a single abstract architecture definition can be refined into many concrete architecture definitions, to foster not only the reuse of components but also of architectures. The refinement relationships between these separate architecture representations — *i.e.,* the relationships between the component roles, classes and instances they are composed of — are proposed to control and verify the global coherence of these multi-level architecture definitions.

The remaining of this paper is organized as follows. Section 2 introduces our proposed architecture-centric, reuse-based development process. Section 3 presents the different

component description levels supported in Dedal, our proposed ADL to support this development process. Section 4 presents the different architecture description levels which can be expressed in Dedal, along with the refinement relations between them. Section 5 introduces the tool suite of Dedal. Section 6 discusses the related works. It studies how existing ADLs are suitable component based software development. Section 7 concludes with future work directions.

2. SOFTWARE ARCHITECTURES IN CBSD

2.1 A Development Process for Component Reuse

CBSD is characterized by its implementation of the "reuse in the large" principle. Reusing existing (off-the-shelf) software components [8] therefore becomes the central concern during development. Traditional software development processes cannot be used as is and must be adapted to component reuse [7, 6]. Figure 1 illustrates our vision of such a development process which is classically divided in two:

- the component development process (sometimes referred to as software component development *for* reuse), which is not detailed here. This development process is the producer of components that are stored in repositories for later consumption by the component reuse process.
- the CBSD process (referred to as software development *by* component reuse) that describes how previously developed software components can be used for software development (and how this reuse process impacts the way software is built).

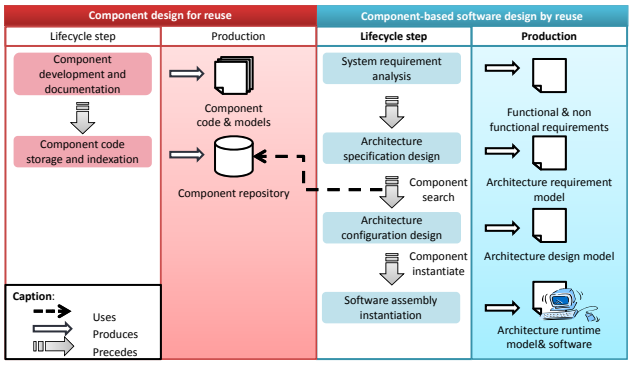

Figure 1: Component development and component-based software development processes

The CBSD process, cannot be realized in one phase. Different concerns are managed sequentially at different design level, from the more abstract to more concrete one. Indeed, architects and programmers cannot put all the information of a component in one description without separated these information. During the development process, these information of components is refined and enriched at each different development stage. In this paper, we proposed a CBSD process, which deliberately focuses on the produced artifacts: architecture models of the software for each development step for each development step[1].

[1]For simplicity's sake, it is also exclusively " reuse-centered" and does not describe how components should be: adapted if no existing component perfectly matches specifications, developed from scratch if no component is found that matches or closely matches specification, tested and integrated, and physically deployed.

In this CBSD process, software is considered to be produced by the reuse of components that have previously been stored and indexed in a component repository. It decomposes in three steps each of which produces a description that models the view of the architecture at this development step [24]:

1. *Architecture specification design.* After a classical requirement analysis step, architects establish the abstract architecture specification (*architecture requirement model*). They define which functionalities should be supplied by components, which interfaces should be exported by components, and how interfaces should connect to build a software system that meets the requirements. All the constituents of this architectural models are abstract (as-wished) and partial.

2. *Architecture configuration design.* In a second step, architects create architecture configurations (*architecture design models*) that define the sets of component implementations (classes) by searching and selecting from the component repository. Abstract component types from the architecture specification thus become concrete (as-found) component types in architecture configurations.

3. *Software assembly instantiation.* In a third step, configurations are instantiated into component instance assemblies (*architecture runtime models*) and deployed to executable software applications.

The claim of this paper is that an architectural description should correspond to each of the three steps of the CBSD process. In other words, components in architectures should be described from all abstract component, concrete component and component instance point of views. These descriptions should reflect the architect's design decisions at each step of the development cycle and be expressed using an adequate ADL.

2.2 Example of a Bicycle Rental System

Figure 2 shows the example used throughout the paper: the architecture specification of a bicycle rental system (BRS). A *BikerGUI* component manages the user interface. It cooperates with a *Session* component which handles user commands. The *Session* component cooperates with the *Account* and *Bike&Course* components to identify the user, check the balance of its account, assign him an available bike and then calculate the price of the trip when the rented bike is returned. In the following sections, we will use a part of this system in the dotted area to illustrate our concepts and ADL syntax.

3. COMPONENT REPRESENTATIONS IN THE THREE LEVELS OF DEDAL

Dedal models architectures at three separate abstraction levels, each of which contains different forms of components and connectors. For now, Dedal mainly focuses on modeling components. At the specification level, components

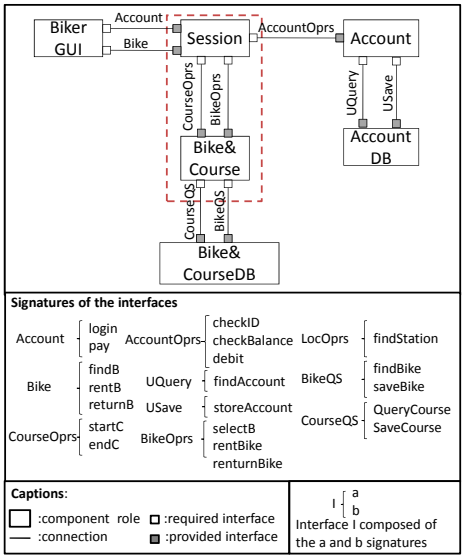

Figure 2: BRS abstract architecture specification

are modeled as roles which are requirement models for concrete component search. These specifications thus are abstract and partial. At the configuration level, components are modeled as (whole) component classes which realize the specifications. Several component classes might correspond to a single component role as there might exist several concrete realizations of a single specification. At the assembly level, concrete component classes are instantiated into component instances that represent runtime components and their parametrization. Figure 3 shows a complete example of components at three levels. In this section, we detailed introduce the different forms of components in these different levels.

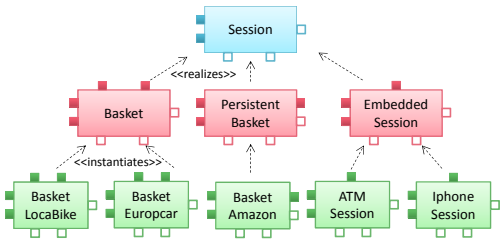

Figure 3: The *Session* component role, some possible concrete realizations and some of their instantiations

3.1 Components in Abstract Architecture Specifications

Component roles model abstract component types in that they describe the roles components should play in the system. A component role lists the minimum list of interfaces (both required and provided) the component should expose and the component behavior protocol that describes the behavior of the component in the architecture (dynamics of the architecture). As they define the requirements of the architect (its ideal view) to guide the search for corresponding concrete components, component roles are abstract and partial component representations (*e.g. Session* component role on Fig. 3). The syntax can be found in Fig. 4.

```
component_role::=
  component_role component_role_name
    ( required_interfaces interface_list )?
    ( provided_interfaces interface_list )?
    ( role_behavior component_behavior )?
    ( MinInstanceNbr PositiveInteger )?
    ( MaxInstanceNbr PositiveInteger )?
interface_list::=
   interface_name ( ; interface_name )*
```

Figure 4: Syntax of component role

- *Interface.* The *interfaces* are the connection points that the component should expose. They can be *provided* or *required*. An interface is composed by its name, direction (provided or required) and its implementation class, as shown in Fig. 5.

```
interface::=
  interface interface_name
    implementation implementation_class
```

Figure 5: Syntax of interface

- *Role behavior.* A *role behavior* is the protocol that describes the expected behavior of a component in an architecture (the behavior protocol is often referred to as the dynamics of the architectures). Dedal uses the protocol syntax of SOFA [19] to describe component role behavior as regular expressions[2]. Other formalisms could have been used instead; the notation chosen is interesting as it is compact and is implemented as an extension of the Fractal component model we use for our experimentations, with companion verification tools. Component protocols capture the behavior of components describing all valid sequences of emitted function calls (emitted by the component and addressed to neighbor components) and received function calls (received by the component from neighbor components).

- *Cardinality.* The precise cardinality of component instances are described in component role descriptions using **minInstances** and **maxInstances**. They define the minimum and maximum numbers of component instances that are to be instantiated from the component class which implements this component role. For example, the *BikeGUI* component role has a maximum number of component instances of *15*, as shown in Fig. 6.

Dedal chooses to describe component roles outside abstract architecture specifications, so as they can be reused from a specification to another (this would not be possible if they were embedded). Figure 6 shows the descriptions of the *BikeCourse* and *BikeCourseDB* component roles. They contain the SOFA-like descriptions of their behavior.

3.2 Components in Concrete Architecture Configurations

At configuration level, components are modeled in two ways with *component types* and *component classes*. Figure 7 provides a close-up view of the relationships between a component role (that model an abstract and partial view

[2] `!i.m` (*resp.* `?i.m`) denotes an outgoing (*resp.* incoming) call of method `m` on interface `i`. `A+B` is for `A` or `B` (exclusive or) and `A;B` for `B` after `A` (sequencing).

```
component_role BikeCourse
 required_interfaces BikeQS; CourseQS
 provided_interfaces BikeOprs; CourseOprs
 component_behavior
   (!BikeCourse.BikeOprs.selectBike,
   ?BikeCourse.BikeQS.findBike;)
   +
   (!BikeCourse.CourseOprs.startC,
   ?BikeCourse.CourseQS.findCourse;)
 MaxInstanceNbr 3
component_role BikeCourseDB
 provided_interfaces BikeQS; CourseQS
 component_behavior
   !BikeCourseDB.BikeQS.findBike;
   +
   !BikeCourseDB.BikeOprs.findCourse;
component_role BikeGUI
 required_interfaces Account; Bike
 component_behavior
   ?BikeGUI.Account.login;
   ...
 MaxInstanceNbr 15
interface BikeQS
 implementation fr.ema.locaBike.BikeQS
```

Figure 6: Component role descriptions of *BikeCourse*, *BikeCourseDB*, and *BikeGUI*; Interface description of *BikeQS*

of a required component), a component type that models the complete type of some existing concrete implementation, a component class that represent the concrete component implementation and a parametrized component instance.

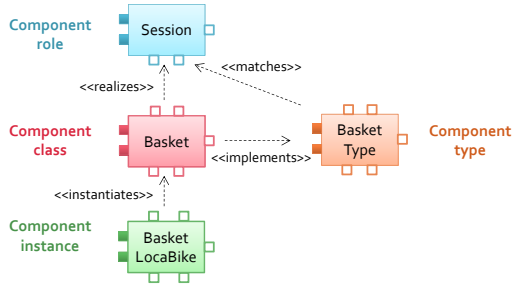

Figure 7: Relationships between component roles, component classes, component types and component instances

3.2.1 Component types

Component types represent the full types of at least one (maybe several) existing component implementations. They are defined by describing the interface set and the behavior of these component classes. Component types are reusable as they can be implemented by multiple component classes which possess the same interfaces and component behavior. The *BasketType* component type description of Fig. 8 is an example of component type description.

3.2.2 Component classes

Component classes represent concrete component implementations. Each component class points to the component type it implements. Component classes can either be primitive or composite.

Primitive component classes (*e.g. Basket* as described in Fig. 10) define the reused components by describing their interfaces, behavior, version and implementing class. Existing models usually do not include links to the implementing class as they assume there is a single implementation.

```
component_type BasketType
 required_interfaces BikeOprs; CourseOprs;
   AccountOprs; CampusOprs; AccessoryOprs
 provided_interfaces Account; Bike
 component_behavior
   (!BasketType.Bike.findB,
   ?BasketType.BikeOprs.findB;)
   +
   (!BasketType.Account.login,
   ?BasketType.AccountOprs.checkID;)
   . . .
```

Figure 8: Description of the *BasketType* component type

In Dedal, components can thus have several distinct implementations (which can be useful to have implementations versioned in such cases as software product lines management). The *BikeTrip* component class description of

```
primitive_component_class::=
 primitive_component_class component_class_name
   implements component_type_name
   content implementation_class
   ( attributes attribute_list )?
   versionID revision_numb
   ( pre_version pre_version )?
   ( motivation motivation )?
   ( condition condition )?
attribute_list::= attribute ( ; attribute )*
attribute::= type attribute_name
```

Figure 9: Syntax of primitive component class

Component class versions are documented by their revision numbers, their previous versions' revision numbers and by the motivations of the changes that entail their derivation from their previous versions. Motivations can either be *corrective* if the evolution aims at fixing some bug or *perfective* if the evolution aims at increasing the performance of the component[3].

```
component_class Basket
 implements BasketType
 content fr.ema.locaBike.Basket
 versionID 1.0
 attributes string company; string currency
```

Figure 10: The *Basket* (primitive) component class description

Composite component classes differ from primitive components in that their implementation is not defined by a single class but by an embedded architecture configuration, *i.e.,* a set of connected inner components. The composite component class definition further defines how the interfaces of the composite component are mapped to corresponding unconnected interfaces of its inner components thanks to delegation connections. As for simple provided interfaces and required interfaces in composite components, delegated interfaces are implementations of the corresponding provided and required interfaces in the corresponding component role. Explicit delegation declarations can be found in almost all the hierarchical ADL models, such as Darwin [20], Unicon [16], and SOFA2.0 [5]. Dedal's syntax for these can be seen in Fig 11. Figure 13 gives an example of the composite component class *BikeCourseDBClass* where the *BikeQS* provided interface of the *BikeData* component inside the *BikeCourseDBConfig* configuration is del-

[3]Motivations are used for gradual component version substitution as described in [23].

egated as a provided interface of the composite component that implements the *BikeQS* interface of the *BikeCourseDB* component role. Figure 12 shows a graphical representation of the same *BikeCourseDBClass* component.

```
composite_component_class::=
   composite_component_class component_class_name
    implements component_type_name
    content configuration_identifier
    delegated_interfaces delegated_interface_list
    ( attributes attribute_list )?
    versionID revision_numb
    ( pre_version pre_version )
    ( motivation motivation )?
    ( condition condition )?
configuration_identifier::=
 configuration_name ( revision_numb )
delegated_interface_list::=
 provided | required inner_interface
    as outer_interface
 ( ; provided | required inner_interface
    as outer_interface )*
```

Figure 11: Syntax of the composite component class

Both primitive and composite component classes can export an **attribute** list (as exemplified on Fig. 10 and 13). Attributes are not mandatory but can be declared as observable / visible properties for component classes so as to be able to set assembly constraints on attribute values in the instantiated component assembly level.

3.3 Components in Instantiated Component Assemblies

Component instances document the real artifacts that are connected together in an assembly at runtime. They are instantiated from the corresponding component classes. They might define constraints on components' attributes that reflect design decisions impacting component states (attribute values) over time. They also set the initial component state by initializing attributes values.

```
component_instance::=
 component_instance component_instance_name
  instance_of component_class_identifier
  init_state attribute_value_list
  current_state attribute_value_list
attribute_value_list::=
 attribute_name = attribute_value
 ( ; attribute_name = attribute_value )*
attribute_value::=
 "an attribute value of the correct type"
```

Figure 14: Syntax of component instance

```
component_instance BasketLocaBike
 instance_of Basket (1.0)
 init_state company="LocaBikecurrency";
  currency=="Euro."
component_instance BikeCourseDBLoca
 instance_of BikeCourseDBClass (1.0)
```

Figure 15: Component instance descriptions of *BikeTripC1* and *BikeCourseDBClassC1*

By default, component classes can be instantiated into multiple component instances. When more precise cardinality information is needed, it is expressed in component role descriptions using **minInstances** and **maxInstances** that define the minimum and maximum numbers of component instances that are permitted to instantiate from the component class which implements this component role. By this means, component classes do not include this configuration-dependent information and remain reusable. In the assembly level, assembly constraints that restrain the valid number of instances will be checked against the cardinality information defined in the component role (in the specification level). There is no rule to constrain the name of component instances of a given component class.

In conclusion, the components in architectures can be found in Figure 16.

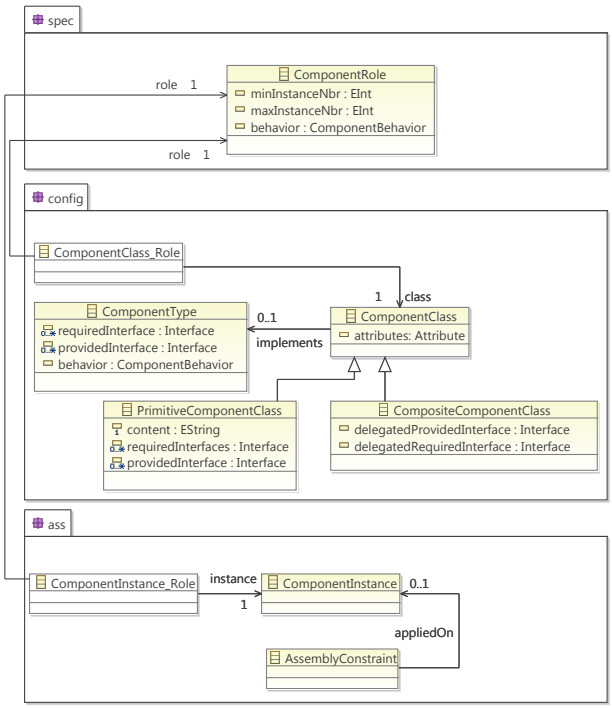

Figure 16: The metamodel of components in Dedal

4. THREE LEVELS OF ARCHITECTURE DESCRIPTION IN DEDAL

In this section, we briefly present the three architecture descriptions in Dedal based on the above component descriptions.

4.1 Abstract Architecture Specifications

Abstract architecture specifications (AASs) are the first level of software architecture descriptions. They provide a generic definition of the global structure and behavior of software systems according to previously identified functional requirements. They model the requirements expressed by the architect to serve as a basis to search for concrete component to create concrete architecture configurations. These architecture specifications are abstract and partial: they do not identify concrete component types that are going to be instantiated in the software system. They only describe the "ideal" component types from the application point of view. In Dedal, an AAS is composed of a set of **component roles**, a set of connections and its architecture behavior.

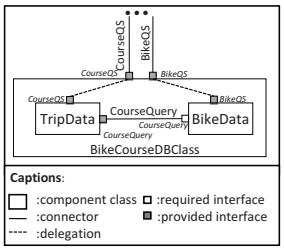

Figure 12: Graphic view of the *BikeCourseDBClass* composite component class and inner configuration

```
component_class BikeCourseDBClass
 implements BikeCourseDBType
 using BikeCourseDBConfig (1.0)
 delegated_interfaces
  provided BikeData(1.0)[BikeDB].BikeQS
   as BikeCourseDBType.BikeQS;
  provided TripData(1.0)[CourseDB].CourseQS
   as BikeCourseDBType.CourseQS
 attributes string company
 versionID 1.0
```

Figure 13: Description of the *BikeCourseDBClass* composite component class and inner configuration

4.2 Concrete Architecture Configurations

Concrete architecture configurations (CACs) are the second level of system architecture descriptions. They result from the search and selection of real component types and classes in a component repository. These component types must match abstract component descriptions from the architecture but need not to be identical; compatibility is sufficient. Component classes must be valid implementations of their declared component type. CACs describe the architecture from an implementation viewpoint (by assigning component roles to existing component types). Architecture configurations thus list the **concrete component and connector classes** which compose a specific version of a software application.

4.3 Instantiated Component Assemblies

Instantiated software component assemblies (ISCAs) are the third level of software architecture descriptions. They result from the instantiation of the component classes from a configuration. They provide a description of runtime software systems and gather information on their internal states. Indeed, this description level enables the record of state-dependent design decisions [22]. ISCAs list **the component and connector instances** that compose a runtime software system, the attributes of this software system, and the assembly constraints the component instances are constrained by.

```
assembly BRSAss
 instance_of BRSConfig (1.0)
 component_instances
  BikeTripC1; BikeCourseDBClassC1
 assembly_constraints
  BikeTripC1.currency="Euro.";
  BikeCourseDBClassC1.company=BikeTripC1.company
 version 1.0
  component_instance BikeTripC1
   instance_of BikeTrip (1.0)
  component_instance BikeCourseDBClassC1
   instance_of BikeCourseDBClass (1.0)
```

Figure 17: Component assembly description of the BRS

Assembly Constraints Assembly constraints define conditions that must be verified by attributes of some component instances of the assembly, to enforce its consistency. Such assembly constraints are not mandatory. Dedal permits to define two types of constraints that must all be enforced and that either are.

- *Logical constraints.* Logical constraints are regular expressions that are written using one or more logical operators among *and* (&&), *or* (||) and *not* (!) in our Dedal definition. To be verified, logical constraints must be evaluated at true.
- *Relational constraint.* Relational constraints can be used in two situations: (1) to declare the relation between an attribute and a given constant value, or (2) to specify the relation between the values of two distinct attributes. The relation operators are admissible are *less than* (<), *greater than* (>), *less than or equal to* (<=), *greater than or equal to* (>=), *equals* (==) and *different from* (!=).
- *Instance constraints.* The number of component instance for a component role can be refined in the assembly constraint to meet the different requirements of different runtime systems. They are expressed using **MinInstanceNbr**, **MaxInstanceNbr** and **InstanceNbr**, that represent the minimum, maximum and exact number of component instances.

Such simple assembly constraints are illustrated on the example of Fig. 17 where the value of the *currency* attribute of component *BikeTripC1* is fixed to *Euro* and where the value of the attribute *company* of the *BikeCourseClassDBC1* component must be maintained identical to the value of attribute *company* of component *BikeTripC1*. Another example that involves cardinalities would be expressed as the assembly constraint *InstanceNbr(BikeGUI)==10* that means that exactly ten component instances of the *BikeGUI* component role should be instantiated in this system. The cardinality information of the *BikeCourse* component role is stored in its specification (see Fig. 6).

```
assembly_constraint::=
 logical_constraint | relational_constraint
logical_constraint::=
 ( ! assembly_constraint ) |
  ( assembly_constraint ( || | && )
assembly_constraint )
relational_constraint::=
 ( instance_attribute ( == | != | > | < | >= | <= )
  ( instance_attribute | attribute_value ) )
instance_constraint::=
 ( ( MinInstanceNbr | MaxInstanceNbr | InstanceNbr )
  ( Component_role_name ) == PositiveInteger )
instance_attribute::=
 component_instance_name . attribute_name
```

Figure 18: Syntax for assembly constraints

However, in our work, assembly constraints are only listed without conflict detection among them, such as the logical conflict or the relational conflict.

5. IMPLEMENTATION OF DEDAL

The Dedal ADL presented in this paper has been implemented in the Arch3D tool suite. The language has been implemented twice: as an XML-based A DL and as a Java-based ADL[4]. The tools also propose a component model which enables to instantiate and manipulate corresponding assemblies at runtime which is coded as an extension of Julia, the open-source java implementation of the Fractal component platform[5].

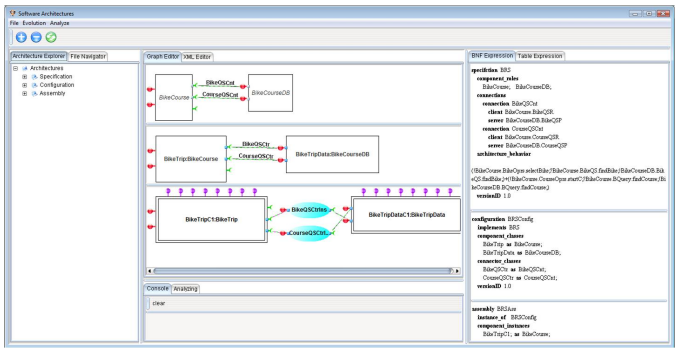

Figure 19: GUI view of BRS example

6. RELATED WORKS

We surveyed representative ADLs to compare their support of components descriptions at different abstraction levels.

6.1 Specification level

Abstract component type (Component role). Abstract component type is abstract component type which just describes the required interfaces of this component in this software system. However, all of these ADLs include concrete component type descriptions, which are suitable for traditional development. The component classes are usually exactly designed and programmed according to concrete component types. C2 [18, 17] is an exception as it provides a subtyping component type theory, which can almost be considered as a quai-abstract component type and facilitate reuse of component specification by instantiating it into different components.

6.2 Configuration level

Component types. All existing ADLs have their concrete component type definition, which specifies the interfaces of components. Some of them further supply the component behavior information, like C2, Wright [1, 2, 3] and SOFA 2.0 [19, 5, 13].

Connector type. Existing ADLs support three connector types: (1) implicit, such as Darwin and SOFA2.0, (2) explicit and predefined, such as C2 and Unicon [21, 20], and (3) explicit and customized, such xADL2.0 [9, 10, 11] and Wright.

Component classes. In existing ADLs, component classes are often described with their component types, which often specify the interfaces and the behavior of component classes. Most works state that all components should have a reference component type. Darwin [16, 15] and Unicon are different as they treat composite component description both as component type and class.

Composite component classes. Hierarchical composition support in existing ADL can be of three types.

- *Explicitly hierarchical composition*: The entire system is treated as a composite component. The hierarchy is explicitly described in component classes. Darwin, Unicon and SOFA2.0 are the representative works.
- *Implicitly hierarchical composition*: Composition is described in the component type. Wright is such an ADL. However, in Wright, component types often preferably described in configurations[6].
- *Explicit non-hierarchical composition*: The C2 is a unique ADL for describing composition. In C2, the communications of components are completely cut by connectors, thus it supposes that if some components are totally cut by a top connector and a bottom connector, and then this configuration can be seen as a composite component. The interfaces of this composite component are the interfaces of both connectors.
- *Complex composition model*: xADL2.0 permits necessary component composition at two levels: configuration and assembly. In the configuration level, the composition is described in the component type. In the assembly level, the composition is directly described in component instance. In both two levels, the composition is embedded in a specific container called *subArchitecture*.

Implementation. The implementation information is seldom included in ADLs, as they intend to be independent from implementation. However, for real architecture design, the implementation information is really important to discriminant between different component classes. FractalADL and Unicon enable to add implementation data to component classes. Unicon is more advanced, as it can support variable implementation, which can be specified during instantiation according to different requirements.

Attributes. All these works do not have their own attribute definition in their component description. However, as FractalADL and xADL 2.0 [9] are extensive ADLs, architects can easily customize these information by adding controllers in FractalADL [4, 14] implementation or xADL 2.0 [9] DTD definition.

6.3 Assembly level

The assembly (runtime) description of software architectures becomes more and more important as dynamic evolution requirements grow. In order to make the connection between configurations (component classes) and assemblies (component instances), there are two methods: mapping configurations to assemblies, or modeling assemblies.

Most dynamic ADLs *i.e.,* C2, SOFA 2.0 and FractalADL use the first choice. This often implies that other tools are necessary to support the mapping. The link is very fragile to preserve and update. Once the link between them is broken or expired, the architecture erosion and drift can occur.

Furthermore, mappings cannot represent all the facets of runtime systems, such as the parametrized attributes when

[4]Detailed information can be found in http://www.irit.fr/~Yulin.Zhang/Dedal.html

[5] `http://fractal.ow2.org/`

[6]In Wright, component types can be defined in *styles*

ADL	Abstract component type	Component type	Component behavior	Component class	Component instance
C2SADEL	Conceptual component : references to external component type	***Conceptual component*** : Extensible, component type.	***behavior*** : Message-based protocols.	No concrete type definitions. Component instances defined with types	---
Wright	---	***Component*** : Extensible, concrete component type.	***Computation*** : CSP-based behavior.	***Component*** : Concrete component types. No references to implementations.	---
Darwin	---	***Component*** : Extensible, concrete component type.	---	***component*** : Concrete component types. No references to implementations.	---
Unicon	---	***interface*** : Built-in, concrete component type.	---	***Component*** : Concrete component types, with multiple references to Implementations (variants).	---
SOFA 2.0	---	***frame*** : Extensible, concrete component type.	***protocol*** : Defined by behavior protocols	Concrete component types. Support references to implementations.	---
Fractal ADL	---	No explicit component types. Component definitions reused and extended in other definitions	set of client and server interfaces. Signature defined as a reference to an implemented type	***definition*** : component definition (set of interfaces). Can be reused to model other components (as a class of component)	Independent definition. Can refine a base definition used as a type
xADL 2.0		***componentType*** : Extensible, concrete component type.		***componentType*** : Concrete component type definition. Can refer to an implementation	Independent definition. Contain a reference to a component type

Table 1: Comparison of components in different ADLs

ADL	Primitive component	Implementation	Attributes	Composite component	Delegation
C2	***instance***	—	—	Explicit, non-hierarchical composition	Derive from two wrapped connectors.
Wright	***Instance***	—	—	Implicit, hierarchical composition: Composite configuration embedding in ***computation*** of component type.	***Bindings***
Darwin	***inst***ance	—	—	Explicit, hierarchical composition	***bind***
Unicon	***instance***	Implementation constraining.	—	Explicit, hierarchical composition	***Bind***
SOFA 2.0	***instance***	—	—	***architecture***: Explicit, hierarchical composition.	***Delegate*** and ***Subsume***
Fractal ADL	***component***	java class	extended	Explicit, hierarchical composition.	***binding***
xADL 2.0	***component***	—	—	***subArchitecture***: Implicit, hierarchical composition. Embedded in component types for configuration level; Embedded in component instance in assembly.	***signatureInterfaceMapping***: in component type level; ***interfaceInstanceMapping***: assembly level.

Table 2: The comparison of primitive and composite component in ADLs

instantiating components or runtime state of component. Thus an ADL that embraces the implementation architecture (component instances) is really needed. From this side, xADL 2.0 and AADL [12] are more complete, as they have their own assembly runtime architecture description.

7. CONCLUSION

This paper proposes a three step component-based development process to ease the reuse of components and architectures. To support component-reuse centric development, a three level component model is proposed by explicitly separating different level information on components into component roles, component classes and component instances. Based on the proposed component model, component design decisions can thus be precisely captured and traced throughout the development process. A three-level architecture is also proposed based on the component model. The three-level syntax of Dedal supports the expression of requirements by the means of abstract and partial component roles that are used as the main conceptual support for the search of reusable components to be included in configurations. The model of the runtime system (the instantiated component assembly) is rich enough to serve as the basis of a full evolution process [25].

We plan to develop this work in two directions. The first perspective for this work is to enrich and experiment the use of Dedal to manage component- based software product lines. We want to enrich the Dedal language to support fine grained product line information such as variability and optionality etc. The second perspective is to develop a quality extension of Dedal to make it support embedded and critical system development.

Acknowledgements

This work has been partially financed by the French ANR-10-BLAN-0219 CUTTER project.

8. REFERENCES

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[2] R. Allen, D. Garlan, and R. Douence. Specifying dynamism in software architectures. In *Proceedings of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Switzerland, September 1997.

[3] R. J. Allen. *A formal approach to software architecture*. PhD thesis, 1997. Chair-David Garlan.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[5] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06*, pages 40–48, Seattle, USA, 2006. IEEE Computer Society.

[6] M. R. V. Chaudron and I. Crnkovic. *Software Engineering; Principles and Practice*, chapter Component-based Software Engineering, pages 605–628. John Wiley & Sons, 2008.

[7] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. In *ICSEA '06*, page 44, Papeete, French Polynesia, october 2006.

[8] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A classification framework for software component models. *IEEE Trans Software Eng*, 37(5):593–615, October 2011.

[9] E. M. Dashofy, A. V. der Hoek, and R. N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA '01*, pages 103–112, Washington, DC, USA, 2001.

[10] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *ICSE '02*, pages 266–276, Orlando, Florida, 2002. ACM Press.

[11] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, 2005.

[12] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

[13] P. Hnetynka, F. Plasil, T. Bures, V. Mencl, and L. Kapova. SOFA 2.0 metamodel. Technical report, Dep. of SW Engineering, Charles University, December 2005.

[14] M. Leclercq, A. E. Ozcan, V. Quema, and J.-B. Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07*, pages 209–219, 2007.

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, Sitges, Spain, September 1995.

[16] J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.

[17] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE'99*, pages 44–53, Los Angeles, CA, May 1999.

[18] N. Medvidovic, R. N. Taylor, and E. J. Whitehead. Formal modeling of software architectures at multiple levels of abstraction. In *In Proceedings of the California Software Symposium 1996*, pages 28–40, April 17, 1996.

[19] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[20] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.

[21] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *ICCDS '96*, pages 2–10, Annapolis, Maryland, 1996.

[22] M. Shaw and D. Garlan. *Software architecture:*

perspectives on an emerging discipline. Prentice-Hall, Inc., 1996.

[23] H. Y. Zhang, C. Urtado, and S. Vauttier. Connector-driven process for the gradual evolution of component-based software. In *ASWEC'09*, Gold Coast, Australia, April 2009.

[24] H. Y. Zhang, C. Urtado, and S. Vauttier. Architecture-centric component-based development needs a three-level ADL. In M. A. Babar and I. Gorton, editors, *ECSA'10*, volume 6285 of *LNCS*, pages 295–310, Copenhagen, Denmark, August 2010. Springer.

[25] H. Y. Zhang, C. Urtado, and S. Vauttier. Architecture-centric development and evolution processes for component-based software. In *SEKE'10*, Redwood City, USA, July 2010.