Architecture-Centric Component-Based Development Needs a Three-Level ADL

Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier

LGI2P / Ecole des Mines d'Alès, Nîmes, France
{Huaxi.Zhang,Christelle.Urtado,Sylvain.Vauttier}@mines-ales.fr

Abstract. Architecture-centric, component-based development intensively reuses components from repositories. Such development processes produce architecture definitions, using architecture description languages (ADLs). This paper proposes a three step process. Architecture specifications first capture abstract and ideal architectures imagined by architects to meet requirements. Specifications do not describe complete component types but only component roles (usages). Architecture configurations then capture implementation decisions, as the architects select specific component classes from the repository to implement component roles. Finally, architecture assemblies define how components instances are created and initialized to customize the deployment of architectures in their own execution contexts. This development process is supported by a three-level ADL which enables the separate definition of these three representations. The refinement relationships between these architecture representations are also discussed.

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 the reuse of components. 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 processes. 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 language called an Architecture Description Language (ADL). This process is structured in three steps, through which architecture definitions are gradually refined, from abstract to concrete representations. After a classical analysis step, architecture specifications first capture 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. Specification and roles are thus key concepts to integrate component reuse effectively in the development process.

M. Ali Babar and I. Gorton (Eds.): ECSA 2010, LNCS 6285, pp. 295–310, 2010.

Second, architecture configurations capture implementation decisions, as the architects select specific component classes to implement component roles. Finally, architecture assemblies define how components instances are created and initialized to customize the deployment of architectures in different execution contexts. Our process is supported by an three-leveled dedicated ADL, called 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 relationship 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. It studies how existing ADLs are suitable for it. 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 6 concludes with future work directions.

2 Software Architectures in CBD

2.1 A Development Process for Component Reuse

Component-based software development is characterized by its implementation of the "reuse in the large" principle. Reusing existing (off-the-shelf) software components therefore becomes the central concern during development. Traditional software development processes cannot be used as is and must be adapted to component reuse [1, 2]. Figure 1 illustrates our vision of such a development process which is classically divided in two:

- the component development process (sometimes referred to as 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.
- and, the component-based software development process (referred to as component-based software development *by* 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).

The proposed component-based software development process deliberately focuses on the produced artifacts (architecture descriptions, as models of the software) for each development step. For simplicity's sake, it is also exclusively "reuse-centered" and does not describe how components should be developed

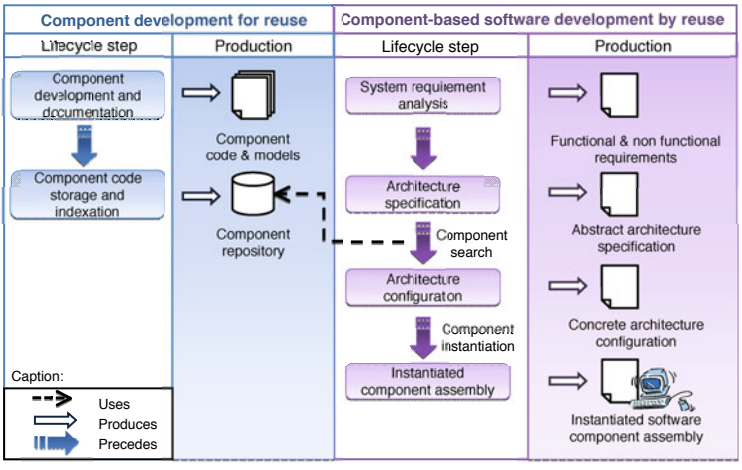


Fig. 1. Component-based software development process

from scratch if no component is found that matches or closely matches specifications, adapted if no existing component type perfectly matches specifications, tested and integrated or, physically deployed.

In this component-based software development 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:

1. *Model of requirements.* After a classical requirement analysis step, architects establish the abstract architecture specification. 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.
2. *Model of design.* In a second step, architects create architecture configurations that define the sets of component implementations (classes) by searching and selecting from the component repository. Abstract component types from the architecture specification then become concrete component types in architecture configurations.
3. *Model of runtime.* In a third step, configurations are instantiated into component instance assemblies 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 component-based software development process. In other words, architectures should be described from all specification (model of requirements), configuration (model of design) and assembly (model of runtime) point of views. These three descriptions should reflect the architect's design decisions at each step of the development cycle and be expressed using an adequate

ADL. State-of-the-art ADLs have been analyzed from this perspective, trying to answer the following questions (that provide a taxonomy for comparison):

- Do existing ADLs support multiple view representations?
- If so, are these views used to reflect successive development steps?
- In cases where several descriptions of a given architecture coexist, which development step can they be associated to?
- Which information on software is captured? In which view / level representation?

2.2 Expressiveness of Existing ADLs

A software system architecture [3] gathers design decisions on the system. It is expressed using an ADL which, in most cases, provides information on the structure of the software system listing the components and connectors the system is composed of. Quality attributes are sometimes provided (*e.g.* xADL [4]). The dynamic behavior of systems is often described (*e.g.* C2SADEL [5], Wright [6], SOFA [7]) but their descriptions are not homogeneous as various technologies (*e.g.* message-based communication, CSPs, regular expressions) are used.

When systems are too complex to easily be described, two classical mechanisms can be used to split descriptions into smaller ones. *Hierarchical decomposition* enables to view the system at various granularities (*e.g.* Darwin [8], SOFA [7] or Fractal ADL [9]). Systems are composed of sub-systems that can further be described at a finer level. *Thematic decomposition* amounts to consider the system from distinct viewpoints (*e.g.* syntactic and behavioral diagrams of UML [10]). Whole systems are seen from several partial viewpoints that make each description focus on some system attributes.

Systems can also be described at various steps of their life-cycles. To our knowledge, no ADL really includes this "time" dimension. Some works such as UML [10] or Taylor *et al.* [3] implement or describe close notions. UML makes it possible to describe object-oriented software at various life-cycle steps but this capability is not transposed in their component model. Taylor *et al.* [3] distinguish two description levels for architectures at design and programming time, respectively called perspective (or as-intended) and descriptive (or as-realized) architectures. However, as far as we know, they do not propose any ADL or metamodel to concretely implement these two architecture descriptions. Garlan *et al.* [11] propose a three-layer framework (task, model and runtime layers) and points out the importance of three levels for dynamic software evolution management. Beside their having close notions, these existing works do not propose such descriptions that would follow the three identified steps of component-based software development.

We then examine the representative ADLs to see which levels of architecture descriptions are supported (as shown in Tables 1 and 2). As far as we know, the studied ADLs unfortunately do not enable the three levels that correspond to lifecycle steps to be all described. This analysis results in requirements for the language presented in this paper:

Table 1. Expressiveness of existing ADLs — Modeling of the three lifecycle steps

ADL	Specification	Configuration	Assembly
C2SADEL	✓	✓	×
Wright	×	✓	×
Darwin	×	✓	×
Unicon	×	✓	×
SOFA 2.0	×	✓	×
FractalADL	×	✓	×
xADL 2.0	×	✓	✓

Table 2. Expressiveness of existing ADLs — Component representations

ADL	Abstract component type	Concrete component type	Component class	Component instance
C2SADEL	×	✓	✓	×
Wright	×	✓	✓	×
Darwin	×	✓	✓	×
Unicon	×	✓	✓	×
SOFA 2.0	×	✓	✓	×
FractalADL	×	✓	✓	×
xADL 2.0	×	✓	✓	✓

1. No ADLs presented in Table 1 is tailored to CBD. Switching to such a reuse-centered development process shall impact the description language.
2. No ADLs presented in Table 1 models component types in an abstract way in order to support the search and selection of concrete component in component repositories. Concrete components in architecture configurations should not be strictly identical to abstract component types described in their architecture specification. As components pre-exist, the specification should define abstract (ideal) and partial component types while configurations describe concrete (satisfying) components that are going to be used (as claimed by Taylor *et al.* [3]).
3. Connectors should not necessary be explicit but the architect should have the possibility to explicit them when needed. Explicit connectors model specific connection types and can be reused from one design to another. However, in most situations, connectors can be system-generated and thus remain implicit for simplicity's sake.
4. Most ADLs do not model the running system (assembly level) or component instances, except xADL 2.0. ADLs should include some description on how components classes are instantiated and what are the characteristics of the running assemblies (constraints on component state values).
5. Components should possibly be primitive (implemented by an implementation class) or hierarchically composed of components (implemented by a configuration).

6. Component types should be reusable. This implies that their description is modularized (outside architectures).
7. Both structural and behavioral viewpoints should be provided for both components and architectures.

2.3 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 a 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 to illustrate our concepts and ADL syntax.

The two following sections present Dedal, the proposed ADL which spans the three levels of architecture descriptions. Dedal enables the description of abstract architecture specifications, concrete architecture configurations and instantiated component assemblies. It also supports a controlled architecture evolution process the description of which is out of the scope of this paper (see [12] for this aspect).

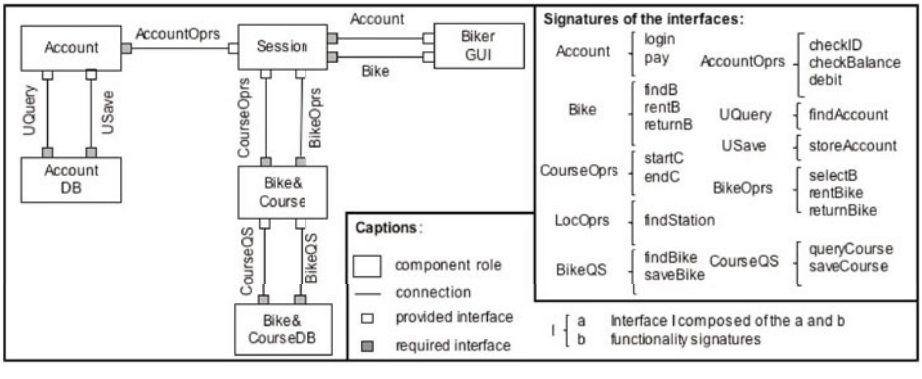


Fig. 2. BRS abstract architecture specification

3 Component Descriptions 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 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.

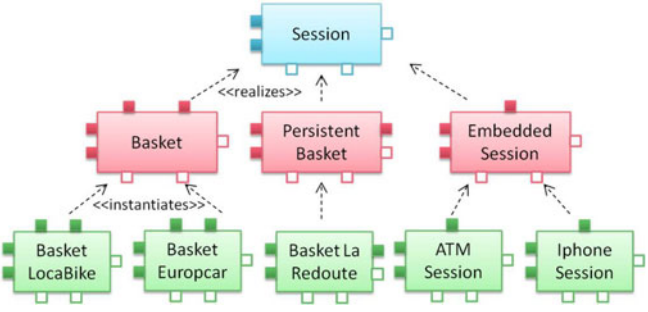


Fig. 3. The *Session* component role, some possible concrete realizations and some of their instantiations

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 parameterizations. Figure 3 shows a complete example of components at three levels.

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). Dedal uses the protocol syntax of SOFA [7] to describe component behavior as regular expressions[1]. 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 used for or experimentation, with companion verification tools. Component protocols capture the behavior of components in their context 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). As component roles are abstract component specifications, Dedal modularly describes them outside architecture specifications, so as they can be reused from a specification to another (which would not be possible if they were embedded). Figure 4 shows the description of the *Session* component role. This description contains (a part of) the SOFA-like description of its behavior.

[1] `!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` (sequence).

```
component_role Session
 required_interfaces BikeOprs; CourseOprs; AccountOprs
 provided_interfaces Account; Bike
 component_behavior
  (!Session.Bike.findB,
  ?Session.BikeOprs.findB;)
  +
  (!Session.Account.login,
  ?Session.AccountOprs.checkID;)
  . . .
```

Fig. 4. *Session* component role

3.2 Components in Concrete Architecture Configurations

At configuration level, components are modeled in two ways with *component types* and *component classes*. Figure 5 provides a close-up view of the relationships between a component role (that model an abstract and partial view 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 parameterized component instance.

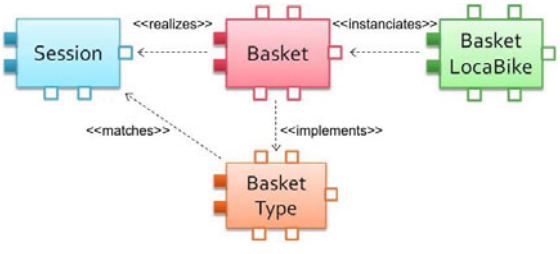


Fig. 5. *BikeCourseDBClass* composite component

Component types represent the full types of at least one (maybe several) existing component implementations. They are defined by describing the interfaces and 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. 6 is an example of component type description.

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. 7) define the reused components by describing their interfaces, behavior, version[2] and implementing class. Existing models usually do not include links to the implementaing

[2] This information (as well as all the versioning information included in other descriptions later on) serves evolution management purposes that are not described in this paper. For more information, the interested reader might refer to [12].

```
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;)
    . . .
```

Fig. 6. Description of the *BasketType* component type

```
component_class Basket
 implements BasketType
 using fr.ema.locaBike.Basket
 attributes string company; string currency
```

Fig. 7. The *Basket* (primitive) component class description

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

Fig. 8. The *BasketLocaBike* component instance description

class as they assume there is a single implementation. In Dedal, components can thus have several implementations (which can be useful to have implementations versioned in such cases as software product lines management).

Composite component classes will be introduced in Sect.4.2. Both primitive and composite component classes can export an attribute list (as exemplified on Fig. 7 and 11). 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 Software 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.

4 Three Levels of Architecture Description in Dedal

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. Types of concrete components need not be identical to abstract types. As CBD processes favors component reuse, component type compatibility should be more permissive than strict identity but still guarantee safety of use. Compatible concrete component types can, for example, provide more functionalities than strictly specified (extra functionality will remain unused) or provide more generic functionalities (use of polymorphism of object-oriented languages)[3].

```
specification BRSSpec
 component_roles
  BikeCourse; BikeCourseDB
  ...
 connections
  connection connection1
   client BikeCourse.BikeQS
   server BikeCourseDB.BikeQS
  connection connection2
   client BikeCourse.CourseQS
   server BikeCourseDB.CourseQS
  ...
 architecture_behavior
  (!BikeCourse.BikeOprs.selectBike;
  ?BikeCourse.BikeQS.findBike;
  !BikeCourseDB.BikeQS.findBike;)
  +
  (!BikeCourse.CourseOprs.startC;
  ?BikeCourse.CourseQS.findCourse;
  !BikeCourseDB.CourseQS.saveCourse;)
  ...
 version 1.0
```

Fig. 9. AAS of the BRS (partial)

In Dedal, an AAS is composed of a set of component roles, a set of connections and its architecture behavior. Figure 9 provides an example of the AAS for the BRS. For readability reasons, this description represents only a small part of the BRS AAS depicted in Fig. 2. **Connections** make interactions between two components possible. They define which component interfaces are bound together. *connection1* and *connection2* from Fig. 9 are such connections. **Architecture behaviors** describe the protocols of complete architectures – meaning all possible interactions between their constituent components. As for component protocols, the syntax used is that of SOFA protocols[4]. Compatibility

[3] The reader further interested about component compatibility can refer to authors' work on component repositories [13] and component substitution [14].

[4] `!c.i.m` (*resp.* `?c.i.m`) denotes an outgoing (*resp.* incoming) call of method `m` on interface `i` of component `c`.

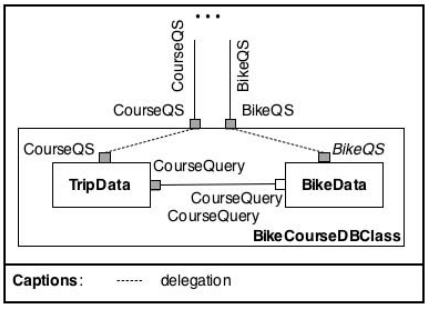
between individual component protocols and the protocol of their containing architecture as well as compatibility between the protocols of two connected components is not studied in this work as we interface our language with corresponding mechanisms (trace inclusion) from SOFA. Figure 9, that describes the BRS architecture specification, contains the BRS architecture protocol. The reader can intuitively check that the protocol of the *BikeCourse* component role is compatible with ("included" in) the protocol of the BRS architecture.

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 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. The architecture of a given software is thus defined by a unique AAS and possibly several CACs. For a given software, each architecture configuration must conform to the architecture specification. This means that each component or connector class used in an architecture configuration must be a legal implementation of the corresponding component role or connection in the architecture specification. Figure 10 describes the architecture configuration of the BRS. The explicit description of **connector classes** is possible (as exemplified on Fig. 12) but not mandatory. In cases where they are implicit, we consider connectors as generic entities which are provided by containers (execution environments) in which configurations are deployed. Connections are automatically administered by containers at runtime to manage the instantiation of configurations. In cases where connectors are explicitly added, their descriptions define the specific connector classes that reflect design choices and that must be used to manage special communication, coordination, and mediation schemes. **Composite component classes** are components the implementation of which is not a simple class but a complete configuration that differ from the above described configurations in that it has some unconnected interfaces. The composite component class concept enables hierarchical composition of architectures which has been identified

```
configuration BRSConfig
 implements BRSSpec (1.0)
 component_classes
  BikeTrip (1.0) as BikeCourse;
  BikeCourseDBClass (1.0) as BikeCourseDB
 version 1.0
```

Fig. 10. A possible CAC for the BRS




```
component_class BikeCourseDBClass
 implements BikeCourseDB
 using BikeCourseDBConfig (1.0)
 delegated_interfaces
  provided
   BikeCourseDBConfig.BikeData.BikeQS
   as BikeCourseDB.BikeQS
  provided
   BikeCourseDBConfig.TripData.CourseQS
   as BikeCourseDB.CourseQS
 version 1.0
 attributes company
```

Fig. 11. The *BikeCourseDBClass* composite component class and its description

```
specification BikeCourseDBSpec
 component_roles
  BikeDB; CourseDB
 connections
  connection ConnectionCourseQuery;
   client BikeDB.CourseQuery
   server CourseDB.CourseQuery
 version 1.0
```

```
configuration BikeCourseDBConfig
 implements BikeCourseDBSpec (1.0)
 component_classes
  BikeData (1.0) as BikeDB;
  TripData (1.0) as CourseDB
 connector_classes
  CourseQuery (1.0) as
ConnectionCourseQuery;
 version 1.0
```

Fig. 12. Descriptions of the *BikeCourseDBSpec* abstract specification and of the *BikeCourseDBConfig* inner configuration

as an effective means to manage system complexity and concretely implement reuse (as whole configurations can be considered as coarser grained components). Composite component classes further define how unconnected interfaces from the inner configuration can be delegated to interfaces of the composite component. As for provided and required interfaces in primitive components, delegated interfaces are implementations of the corresponding provided and required interfaces in the corresponding component role. Figures 11 and 12 give the example of the composite component class *BikeCourseDBClass* that implements the *BikeCourseDB* role where the *BikeQS* provided interface of the *BikeData* component inside the *BikeCourseDBConfig* configuration is delegated as a provided interface of the composite component that implements the *BikeQS* interface of the *BikeCourseDB* component role. Figure 11 shows a graphical representation of the same *BikeCourseDBClass* component.

Conformance between an AAS and a CAC is a matter of conformance between component roles and the component classes that supposedly implement them. Many conformance relations could be defined, from stricter to very loose ones. On the one hand, we defend that reused components need not be exactly identical to specifications because being too strict in this matter might seriously decrease the number of reuse opportunities. On the other hand, it is expected from a conformance relation that it enables verifications that guarantees good chances that the thought component combination will execute. The rule of the thumb that can be used is that concrete components must provide at least what

is the specification declare it provides and require less than what the specification already requires. This translates into rules for interfaces and rules for behavior protocols:

- the provided interfaces list of the concrete component class must contain all the interfaces specified in the component role,
- all the required interfaces of the concrete component class must be specified in the component role,
- the behavior of a component class includes (in the sense of trace inclusions) the behavior specified in the component role.

Variations on these rules can further consider interface specialization rules as in [13]. Figure 7 shows an example of a concrete component class (*BikeTrip*) that has a required interface (*LocOprs*) that is not in the specification (*BikeCourse* component role) it conforms to. In the case of composite components, delegated interfaces of provided (*resp.* required) direction are considered exactly as if they where provided (*resp.* required) interface of primitive components. Indeed, when considered externally, composite components can be seen as if they where primitive. Figure 7 provides an example of the *BikeCourseDBClass* composite component class, that conforms to the specification of the *BikeCourseDB* component role.

4.3 Instantiated Software Component Assemblies

Instantiated software component assemblies (ISCAs) are the third level of system 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 [15]. 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. Figure 13 gives the description of a software assembly that instantiates the BRS architecture configuration of Fig. 10.

```
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)
```

Fig. 13. Component assembly description of the BRS

The explicit description of **connector instances** is possible but not mandatory. In cases where they are implicit, we consider them as generic entities which are provided by containers (execution environments) in which configurations are deployed. In cases where connector instances are explicitly added, their descriptions define the specific attributes that reflect implementation choice to meet different situation. 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. **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. For now, Dedal only permits to list several constraints that must all be enforced and that either:

- limit the possible values for an attribute to a given constant,
- restrain the cardinality of some connection end (*i.e.,*the number of instances of the component class that stands at the end of the connection in the configuration) to a given constant,
- or, enforce equality of the values of two distinct attributes that pertain to two distinct component instances of a given component assembly.

Such assembly constrains are illustrated on Fig. 13 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(BikeTrip)=2* that mean that exactly two component instances of the *BikeTrip* component class should be instantiated in this system. The cardinality of the *BikeTrip* component class is recorded in the *BikeCourse* component role specification. These constraints are very simple and do not yet enable the expression of alternatives, negation, nor the resolution of possible conflicts. Such extended assembly constraint management is one of the perspectives for this work for which we plan to take inspiration from systems that manage architectural styles as constraints sets [6, 16].

Conformance between a CAC and an ISCA is quite straightforward. All component instances of the assembly must be an instance of a corresponding component class from its source configuration (and reciprocally). Conformance also includes the verification that attribute names used in an assembly constraint of some component assembly pertain to the component classes the

components of the assembly are instances of. For example, the assembly constraint *BikeTripC1.currency="Euro."* of Fig. 13 has the conformance process check whether the *BikeTrip* component class (from which *BikeTripC1* is instantiated) possesses a *currency* attribute.

5 Implementation of Dedal in the Arch3D Tool Suite

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 ADL and as a Java-based ADL. The tools also propose a component model which enables to instantiate and manipulate corresponding assemblies at runtime which is extended as an extension of Julia, the open-source java implementation of the Fractal component platform[5]. Our extension of the Fractal platform tools has two purposes: to support the explicit and separate representation of specifications and configurations and, to embed these representations in the component model. The three architecture representations are then available and manipulable at runtime, also providing a full support for evolution management. The *Arch3D Editor* tool provides a graphical console to create, view and modify Dedal-based Fractal architectures. Architects can simultaneously display the different representations of an architecture and work on them.

6 Conclusion

Dedal enables the explicit and separate representations of architecture specifications, configurations and assemblies. Architecture design decisions can thus be precisely captured and traced throughout the development process. 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 instanciated component assembly) is rich enough to serve as the baseis of a full evolution process [12]. A perspective for this work is to experiment the use of Dedal to manage component-based software product lines.

References

1. Crnkovic, I., Chaudron, M., Larsson, S.: Component-based development process and component lifecycle. In: Proc. of the Intl. Conf. on Software Engineering Advances, Papeete, French Polynesia, October 2006, p. 44 (2006)
2. Chaudron, M., Crnkovic, I.: Component-based Software Engineering. In: Software Engineering; Principles and Practice, pp. 605–628. Wiley, Chichester (2008)
3. Taylor, R., Medvidovic, N., Dashofy, E.: Software Architecture: Foundations, Theory, and Practice. Wiley, Chichester (January 2009)

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

4. Dashofy, E., van der Hoek, A., Taylor, R.: A highly-extensible, XML-based architecture description language. In: Proc. of 2nd WICSA Conf., Amsterdam, The Netherlands, pp. 103–112 (2001)
5. Medvidovic, N., Rosenblum, D., Taylor, R.: A language and environment for architecture-based software development and evolution. In: Proc. of ICSE Conf., Los Angeles, USA, May 1999, pp. 44–53 (1999)
6. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6(3), 213–249 (1997)
7. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. 28(11), 1056–1076 (2002)
8. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes 21(6), 3–14 (1996)
9. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: 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)
10. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
11. Garlan, D., Schmerl, B., Chang, J.: Using gauges for architecture-based monitoring and adaptation. In: Proc. of Working Conf. on Complex and Dynamic Systems Architecture, Brisbane, Australia (December 2001)
12. Zhang, H. Y., Urtado, C., Vauttier, S.: Architecture-centric development and evolution processes for component-based software. In: Proc. of 22nd SEKE Conf., Redwood City, USA (July 2010)
13. Aboud, N.A., Arévalo, G., Falleri, J. R., Huchard, M., Tibermacine, C., Urtado, C., Vauttier, S.: Automated architectural component classification using concept lattices. In: Proc. of the Joint WICSA/ECSA Conf., Cambridge, UK (September 2009)
14. Desnos, N., Huchard, M., Tremblay, G., Urtado, C., Vauttier, S.: Search-based many-to-one component substitution. J. Softw. Maint: Res. Pract. 20(5), 321–344 (2008)
15. Shaw, M., Garlan, D.: Software architecture: perspectives on an emerging discipline. Prentice-Hall, Englewood Cliffs (1996)
16. Cheng, S.W., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste, P.: Using architectural style as a basis for system self-repair. In: Proc. of 3rd WICSA Conf., Montreal, Canada, August 2002, pp. 45–59 (2002)