A Multi-Paradigm Approach to Describe Software Systems

ADEL SMEDA, MOURAD OUSSALAH, and TAHAR KHAMMACI
LINA, Université de Nantes
2, Rue de la Houssinière, BP 92208
44322 Nantes Cedex 03, France
Tel: +332 51125963, Fax: +332 51125812

Abstract: - As software systems grow, their complexity augments dramatically. In consequence their reusability and their evolvability are becoming a difficult task. To cope with complexity sophisticated approaches are needed to describe the architecture of these systems. Architectural description and modeling is much more visible as an important and explicit analysis design activity in software development. During the last decade software architecture research community has proposed number of Architectural Description Languages (ADL) that designed to represent complex software system’s architectures. In the other hand, object-oriented modeling has become widely used and appreciated by the industrial community. In this article we present a multi-paradigm approach, based on object-oriented modeling and architectural description, to describe complex software systems. The main contribution of this approach is defining connectors as first-class entities to deal with complex interactions among components. The approach improves reusability by enhancing extensibility, evolvability, compositionality, and understandability of complex systems. It assigns number of operational mechanisms to support that.

Key-Words: - architectural description, object-oriented modeling, architecture description languages, components, connectors, configurations, component-based systems.

1 Introduction
Object-oriented modeling describes systems as a collection of classes interact with each other. Interactions in object-oriented are merely procedure calls (or function invocations) and shared data access. To handle the communications of a complex system with heterogeneous components procedure calls and shared data access are not adequate. Indeed complex and composite connectors are needed to support complex interactions and to overcome the compatibility problem. Architectural description approach came further by separating computations (components) from relations (connectors). In fact architectural description styles are motivated by software connectors, styles such as pipe and filter, event-driven, message-based, and data flow are based on different types of connectors. In spite of that, software architecture community does not have defined explicitly what the nature of a connector is. Connectors are often considered to be explicit at the level of architecture, but implicit in a system’s implementation [1]. In addition some Architecture Description Languages (ADL [2] [3]) model connectors as components (e.g. the notion of a “connection component” in Rapide [4]). Hence the current level of support for connectors is still far from the one components have.

We think that connectors should be treated as first-class entities because of two types of reasons: technical and application [1].

Technical
Technical problems are caused by combining the notion of interaction with the notion of computation. They can be summarized in four points:
Lack of abstraction: the lack of abstraction of the mechanisms used to model connectors does not permit connectors to have a true place in the paradigm of components. Moreover this absence of abstraction prevents connectors from updating and evolving their concepts and their code. Therefore it is often required to understand the whole functionalities of a component to distinguish the implicit connectors, which are buried in the component (connectors and their semantics are mainly based on components).
Increase the complexity of a component: the existence of many interactions among components in a system contributes to a significant increase in the complexity of the components. And each new interaction adds more complexity to the components.
Hard to use: the usability of connectors (in the form of properties of components) is proved to be difficult. Relations among components are not fixed; a connector may be used differently by different kinds of components.
Lack of reusability: as components and connectors are amalgamated, it becomes difficult to identify the behavior intrinsic of a component, and this harms the reuse and the evolution of components and connectors.

Application
Various applications can be modeled more easily by having components and connectors explicitly separated, including:
Configuration management: The components can be organized in several ways: 1. they can be composed of other components (hierarchy of composition) 2. they can exist in several versions (hierarchy of derivation) 3. they can have several representations or point of views. The combination of these various hierarchies often requires complex propagation mechanisms [5]. For example: If a component of a composite component becomes invalid the composite component should be invalid. The creation or destruction of a version of a component can involve the creation (or the destruction) of a new version of the composite component.
Distributed systems: If components are localized at different nodes of a distributed system, the cooperation of the distant components is determined by the semantics of the connectors, which connect them.
Subsystems Coupling: If subsystems need to be coupled, connectors must be defined among the components of the subsystems. The updating or changing operated in one of the subsystems must then be propagated via these connectors to the other subsystems. Hence as a result:
The consequences of the activities (functionalities) of a component are not specific properties to this component, but properties of the connectors that connect these components.
The interactions among components (i.e. connectors) often need to adapt to the requirements of a specific environment. Even within only one system, various concepts possibly need to add their specific interaction rules to the rules, which already allowed.

2 Component-Object based Software Architecture (COSA)
Object-oriented modeling and architectural description have many things in common. In fact the two have been built based on similar concepts, which are abstraction and components interaction. In architectural description components and connectors are the foci of the system, where components are abstraction of modules and connectors are description of the communication and the interactions among those components. While in object-oriented systems classes are given as an abstraction of data, and functions serve as interfaces to that data, also associations are provided to describe the relations and the communications among classes and objects.

In terms of architecture in general the similarity between the two fields is obvious [6]. In terms of intentions, the two approaches are aimed toward reducing costs of developing applications and increasing the potential for related product family [7], [8] hence encouraging reusability and component based programming. The two have their foci shifted from lines of code to coarser grained architecture elements and their overall interconnection structure.

The differences between the two approaches can be summarized in eight points:
1. Level of granularity. Object-oriented’s level of abstraction is the class’s level. Meanwhile architectural description’s level of abstraction is the component’s level which is much higher than class’s.
2. Definition of interactions. In object-oriented interactions are merely defined through procedure calls and function invocations. In architectural description interactions are defined using connectors, even though they are not explicitly presented.
3. Interfaces. In architectural description interfaces are entities of interactions described explicitly through ports and roles. They go beyond simple function calls, permitting description of complex forms of communication.
4. Definition of a system’s global architecture. Contrary to object-oriented modeling architectural description allows the definition of a system’s global architecture before completing the construction of its components (its implementation).
5. Support for non-functional properties. Object-oriented modeling does not provide a direct support to analyze non-functional properties, e.g. performance, safety, security, schedulability, portability, conformance to standards, and reliability. For architectural design such properties are as important as what is being computed.
6. Community support. Object-oriented is well known practiced by a broad community of engineers and developers. Architectural description remains an ad hoc concept known only to the academic community.
7. Standardization of concepts. Object-oriented community has presented UML (Unified Modeling Languages [9]) as a standard notation for most of the object-oriented modeling
notations. While architectural description community is much less cohesive, leading to a different number of notations and approaches (and perhaps contradictable).

8. Refinement to source code. Architectural description provides only high-level models, without ways to relate those models to source code. As noted in [6] such linkages are important to preserve the integrity of the design.

2.1 COSA basic elements

COSA describes systems in terms of types and instances. Components, connectors, and configurations are types that can be instantiated to build different architectures. The basic elements of COSA are: components, connectors, interfaces, configurations, constraints, and properties. Figure 1 shows a meta-model of the approach. In the following we introduce them briefly:

Components represent the computational elements and data stores of a system. Each component may have multiple interfaces with multiple ports. Each interface consists of a set of points of interactions between the component and the external world that allow the invocation of the services. A component may have many implementations. Our description of components is similar to the other software architecture approaches [2][3][5][7][8].

Connectors represent interactions among components. They provide the link for architectural designs. They are explicit architectural entities that bind components together and act as mediators between them, their functions separate them from components. Connectors in COSA will be explained clearly in section 2.2.

Interfaces in COSA are first-class entities. They provide connection points between a component and a connector, consequently between that component and another component. Likewise, they define how the communication between two components can take place. A component interface connection point is called port and a connector interface connection point is called role. In addition to ports and roles interfaces could have
other communication services such as broadcasting, auxiliary interfaces, etc. Interfaces also could be assigned number of constraints and properties. An interface could transfer data either parallel or serial and its communication mode could be synchronous or asynchronous.

**Configurations** represent a graph of components and connectors and describe how they are fastened to each other. In COSA configurations are classes that can be instantiated several times giving different architectures of a system. A configuration may have an interface (or interfaces) that defines a number of ports, and each port are bound to one or more ports of the internal components. In general, configurations may be hierarchical: components and connectors may represent subconfigurations that have internal architectures. They are referred to as *representations*.

**Properties** represent additional information (beyond structure) about the parts of an architectural description. Typically they are used to represent anticipated or required extra-functional aspects of an architectural design. There are two types of properties: functional properties and non-functional properties. Functions that relate to the semantics of a system and represent the requirements are called functional properties. Meanwhile non-functional properties represent additional requirements, such as safety, security, performance, and portability.

**Constraints** are specific properties, they define certain rules and regulations that should be met in order to ensure adherence to intended component’s/connector’s uses.

### 2.2. Connectors in COSA

A COSA connector is represented by a set of named *roles* and a *glue* specification. In principle, a *role* is an interface of the connector intended to be tied to a component interface. In the context of the frame, a *role* is either a *provide* role or a *require* role. A *provide* role serves as an entry point to the component interaction represented by the connector type instance and it is intended to be connected to a *require* interface of a component (or to a *require* role of another connector). Similarly, a *require* role serves as an outlet point of the component interaction represented by the connector type instance and it is intended to be connected to a *provide* interface of a component (or to a *provide* role of another connector). The number of roles within a connector denotes the *degree* of a connector type. For example in a client-server a connector type representing procedure call interaction between client and server entities is a connector with degree of two. More complex interactions among three or more components are typically represented by connector types of higher degrees.

The *glue* specification describes the functionality that is expected from a connector. It could be just a simple protocol links the roles or it could be a complex protocol that does various operations including linking, conversion of data format, transferring, adapting, etc. In general the *glue* of a connector represents the connection type of that connector. Connectors can also have an internal architecture that includes computation and

![Diagram of connectors in COSA](Fig. 2. Structure of connectors in COSA)
information storage. For example a connector would execute an algorithm for converting data from format A to format B or an algorithm for compressing data before it transmits them.

Figure 2 presents a meta-model illustrates the structure of COSA connectors. Figure 3 shows the definition in an explicit way.

Class Connector connector-name{
  Interface {
    Roles {…..} // define roles
    ...... } // other interfaces services (auxiliary
    ...... } // interfaces
  Glue { // define glue
    Connection-Type {…..} // communication
    ...... } //, conversion, etc.
  Define-Composition { Classes { define
    components; }
    ...... } //subconnectors and
    ...... } //subcomponents
  Binding { define binding between roles}
  Properties {…..} // glue’s properties
    ...... } // connector’s properties
  Constraints {…..} // connector’s constraints
}

Fig. 3 COSA definition of connectors.

As a simple illustrative example, figure 4 shows a client-server system described using COSA. In the example there are two components a client which sends requests demanding servers and a server which provides servers. There is also a connector (RPC) which acts as a mediator between the two components. The topology of this system is declared by instantiating the configuration client-server and built by instantiating the components (client, server) and the connector (RPC) and by listing a set of attachments. The example shows only one architecture (arch-1), more architectures with different topologies can be instantiated similarly.

Class Configuration client-server {
  Interface {Port External {External-protocol;} }
  Class Component server {
    Interface {
      Port provide {provide-protocol;}
      external-port { External-port-protocol ;}
      Properties { connection-mode=sync;
        data-type =format-1;
        max-clients=2; }
    }
  }
  Class Component client {
    Interface {
      Port request {sent-request;}
      Properties { data-type =format-2; }
    }
  }
  Class Connector RPC{
    Interface {
      Roles {participator-1, participator-2 }
    }
    Glue {
      Connection-type {
        read participator-1;
        convert from format-1 to format-2;
        write participator-2;}
      Properties {bidirectional; }
    }
    Properties {max-roles =2; }
    Binding { server.external-port to External;}
  }
  Instance client-server arch-1 {
    Instances {
      S1: server;
      C1: client;
      C1-S1: RPC; }
    Attachments {
      C1.request to C1-S1. participator-1;
      S1.provide to C1-S1. participator-2; }
  }
}

Fig. 4 Describing a client-server system using COSA.

4 Related Work

A number of perspectives concerning co-existing architectural description and object-oriented modeling have been already proposed [6] [10] [11]. In the work of Medvidovic [11], the intention is to investigate the possibility of using the Unified Modeling Language as a starting point for bringing architectural modeling into wider, industrial use. Some strategies for modeling software architectures using UML have been suggested like using UML “As it is”, constraining the UML meta model using UML’s built-in extension mechanisms or extending the UML meta model to directly support the needed architectural concepts. The goal of the first strategy is to assess the support provided by UML for the needs of architectural modeling and to compare directly the modeling power provided by UML to that of an ADL. The second strategy involves using OCL to specify constraints on existing meta-classes of UML’s meta model and treats UML as a core notation that is extended in order to support specific architectural concerns. The third strategy has been eliminated because it violates the key requirement that the resulting notation must be conform to the UML standard and could become compatible with UML-compliant tools.

The goal of the Garlan’s work [6] [10] is to enumerate and evaluate different options an architect has in selecting UML modeling constructs to represent architectural structure (i.e. components, connectors, systems, and styles). In this study, the authors investigate four strategies for modeling architectural elements: 1) UML classes as component types and components objects as their instances, 2) UML classes as component types and UML classes as component instance, 3) UML components as component types and UML component instances as component instances, 4) UML subsystems as component types and subsystem instances as component instances. In [12] Garlan has considered the UML Real-time
Profile as a particular variant on these strategies, based on the mapping of ADL concepts to UML Real-Time. We can notice that the first perspective has focused on non-structural aspects of architectures (behaviors, interactions and constraints), while the second perspective has concentrated on structural aspects of architectures (components, connectors, ...). In the other hand, our work is concerning with applying some operational mechanisms from object-oriented systems to architectural description and see what these mechanisms bring more to software architecture in terms of reusability.

5 Conclusion
Describing complex software systems based on off-the-shelf reused components requires not only well-defined components but also well-defined connectors. Since the architecture of a software system depends not only on components but also on their interactions, connectors should be raised to the level of components. In this article we have presented a multi-paradigm approach of software architecture based on object-oriented modeling and architectural description. It defines connectors the same way components are defined, by separating their interfaces from their behaviors. By defining connectors as first-class elements we can specify number of operational mechanisms for their reuse and evolution [1]. For our future work we are defining what we call an active connector. An active connector has the ability to adapt to changes and to propagate those changes. It does certain actions after an event occurs and a given constraint is satisfied. We think that this type of connectors permits defining more explicitly complex software systems.

References:


