A Graphical Representation of COTS-based Software Architectures [*]

Raúl Monge[†], Carina Alves[‡], and Antonio Vallecillo[§]

[†] Dpto. Informática. Universidad Técnica Federico Santa María. Valparaíso, Chile.

rmonge@inf.utfsm.cl

[‡] Universidad Federal de Pernambuco, Brasil, and University College, Londres, UK.

C.Alves@cs.ucl.ac.uk

[§] Dpto. de Lenguajes y Ciencias de la Computación. Universidad de Málaga. Spain.

av@lcc.uma.es

Abstract

As commercial components emerge and commence to be used in real applications, the need to design such systems with a sound architecture becomes a critical issue. Traditionally, Architectural Description Languages have been used for that purposes, although their formality and unfriendliness have limited their use in industrial environments. On the other hand, the successful and widespread modeling notation UML has shown some limitations when used for representing Software Architectures. In this paper we describe a set of constructs and UML extensions that facilitate the description of Software Architectures, particularly those of applications developed from reusable COTS components.

1 Introduction

Complex software systems and applications require expressive notations for representing their architectures. Traditionally, specialized Architecture Description Languages (ADLs) have been used, which allow the formal description of the structure and behavior of the architecture of the application being represented [12]. However, the formality and lack of unified visual support of most ADLs have encouraged the quest for more user-friendly notations. In this respect, the general-purpose modeling notation UML is clearly the most promising candidate, since it is familiar to developers, easy to learn and to use by non-technical people, offers a close mapping to implementations, and has commercial tool support.

The current definition of UML does not offer a clear way for encoding and modeling the architectural elements typically found in architectural description languages. Basically, the problems are the semantic clashes between the UML and ADL concepts (e.g. the term component has slightly different meanings in both worlds), the lack of a clear mapping between the UML and the ADL concepts (as discussed in [5]), the vague semantics of UML versus the precise semantics required by ADLs [10], and the gaps that currently UML has for modeling and exploiting architectural styles, explicit software connectors, and local and global architectural constraints [13].

Several authors have studied the expressiveness of UML in order to represent software architectures, and the ways to adapt this notation to do so (in particular [5, 13].) The difficulties found made the OMG include an explicit requirement about supporting component-based development and run-time architectures of applications in the UML 2.0 Superstructure Request for Proposals (OMG document `ad/00-09-02`.) However, after looking at the initial proposals received, we feel that the results will still be insufficient for fully representing software architectures.

A software architecture is something more than the description of the structure and behavior of an application, and the constraints on them. There is also the need to associate QoS attributes to architecture descriptions,

[*]Work supported by CYTED subproject VII, code VII.18 (“WEST: Web-oriented Software Technology”)

as the only means to effectively express and exploit the extra-functional requirements of systems and applications [4, 20]. In general, two kinds of quality attributes can be distinguished: those observable at runtime (such as availability, accuracy, or timeliness), and those that show during the system's life cycle (such as maintainability, upgradeability, or openness.) Although these latter qualities cannot be observed and measured during runtime, they still need to be considered during the design of the system.

Our work has a special emphasis on describing software architectures of COTS-based applications, i.e. those developed by assembling commercial off-the-shelf (COTS) software components. In Component-Based Software Development (CBSD), components interact following a client-server model, and using RPCs as the basic communication mechanism. The architectures of the applications that we try to represent follow this model, although we shall see how our notation can be used to represent other kinds of architectures too.

In CBSD, the selection process of COTS components plays a critical role. This process initially decomposes the application requirements into a hierarchical criteria set, which usually includes components' functionality, extra-functional requirements, architecture constraints, and other non technical factors such as vendor guarantees and legal issues. During the selection process, the properties of each COTS component candidate are identified and assessed according to this set of evaluation criteria. Of course, the application software architecture should allow the description of the properties of its constituent components as well as traceability of requirements, i.e., how the system QoS and extra-functional properties are decomposed into the individual component properties, and viceversa—how the components' attributes and properties implement or guarantee a given system property.

In this paper we propose a set of constructs that facilitate the description of COTS-based Software Architectures. These constructs allow the graphical representation of the application structure, together with annotations for describing the behavior and runtime quality attributes of the architectural elements. For that we shall build on the standard UML notation, defining these constructs as extensions to it, along the lines of the work by Selic and Rumbaugh, which defines UML extensions for modeling real time systems [23]. We will try to respect the standard UML semantics as much as possible, hence building a notation which is consistent with the current UML standard.

The structure of the paper is as follows. After this introduction, Section 2 defines the constructs we propose for representing the structure of a software architecture. Then, Section 3 discusses how to express some of the application attributes and its elements, in particular the operational semantics of the components and connectors, the choreography that defines the protocol access to their services, and some of the runtime quality attributes of the applications. Section 4 presents an example that illustrates our approach, and Section 5 compares our work with other related proposals. Finally, Section 6 draws some conclusions and points out some further research activities.

2 Structure

The software architecture of a system defines its high-level structure, exposing its organization as a collection of interacting components. Focusing on the architectural structure, five basic concepts can be identified: components, connectors, systems, properties, and styles [5].

- *Components* represent the computational elements and data stores of a system. Components have multiple interfaces, called *ports*, each one defining a point of interaction with its environment. A component may have several ports of the same type.

- *Connectors* represent interactions among components. They provide the glue for architectural designs and they are first class citizens in ADLs. Therefore, they deserve explicit modeling treatment. Connectors have interfaces that define the *roles* played by the participant components in the interactions.

- *Systems* are collections of inter-related components and connectors. In general, systems may have a hierarchical structure, whereby subsystems may have their own internal structure. In order to be composed, systems may also expose interfaces and properties.

- *Properties* describe some of the extra-functional aspects of components and systems, typically used to represent QoS requirements of an architectural design. In general, it is desirable to associate properties to architectural elements (components, connectors, or systems) in order to allow traceability, or even to be

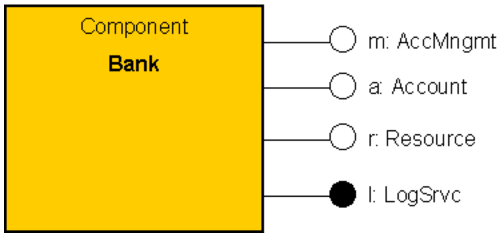


Figure 1: Component Bank in the short notation.

able to infer system properties from the properties of its constituent parts (e.g. some ADLs allow to calculate the system throughput and latency based on performance estimates of the constituent components and connectors.)

- Finally, an architectural *style* represents a family of related systems, that share a common set of architectural elements (components, connectors, property types) together with rules for composing instances of those elements.

The following describes in detail how these elements are modeled in our approach.

2.1 Components

In order to use the term *component* in a consistent way, we need to differentiate between UML components, ADL components, and software components. These three concepts have in general slightly different meanings. UML components are replaceable, modular units of deployment, that encapsulate implementation and expose a set of interfaces [21]. ADL components do not represent units of deployment, but units of computation. Besides, they typically represent *logical* components (e.g. business or EDOC components), whilst UML and software components represent *physical* components (e.g. EJB, CCM, COM+ components or .NET assemblies.) For software components we will adopt Szyperski's definition: binary units of possibly independent production, acquisition and deployment that interact to form a functioning system, with explicit interfaces and context dependencies only [25]. Unlike UML components, software components can be grouping elements, since they can be composed of atomic components and frozen resources [25].

The adjective COTS will refer to a special kind of (usually large grained) software components, which are [15]:

– sold, leased, or licensed to the general public

– supported and evolved by the vendor, who retains the intelectual property rights

– available in multiple, identical copies, and

– used without modifications of their internals.

The inabilities for UML components to represent ADL components have been thoroughly described elsewhere [5, 13]. In our context, we try to describe Software Architectures made up from software components, and therefore ADL components will correspond to software components, thus sharing common characteristics.

Component services are described in terms of *ports*. A port is a named interface of a component. Interfaces are sets of operations and events that define the services provided or required by the component. Thus, a component may optionally expose the same interface more than once using different port names.

Ports are the entities used by connectors to wire up components according to the system architecture. We will distinguish between two kinds of ports: *provided* ports define the services that a component offers, while *required* ports are those that describe the external services used by the component during execution.

Components accept two kinds of notations. In the short one, components are shown using the UML classifier box with the keyword "component", and the ports will be denoted by lollipops. White lollipops will represent provided interfaces, while black (solid) lollipops will represent required interfaces (Fig. 1). In the

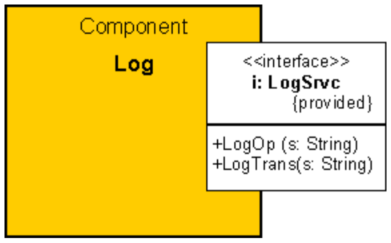


Figure 2: Component Log in the extended notation.

extended notation, the classifier box can contain the artifacts that form part of the component, and the ports will be represented by the full classifier notation for an interface, protruding out of the component box. The interface will show the signature of the operations supported or required. In order to differentiate between provided and required interfaces, the corresponding tags "{provided}" or "{required}" will have to appear besides the name of the interface, thus qualifying it.

Figure 1 represents a component called Bank, which offers three ports of types AccMngnt, Account, and Resource, and requires one (LogSrcv). Figure 2 shows a component called Log, which provides a port named i of type LogSrcv, expressed in the extended notation. If a component offers several ports of the same type, they have to be assigned different names. For instance, a,b:X names two ports which offer the services defined by interface X. Arrays can be used to name ports. Thus, o[1..10]:Y represent 10 ports of type Y and p[1..*]:Z can be used to represent an unbounded set of ports of type Z.

Components can be composed of a set of atomic components (i.e. binary, executable or script files), and by a set typed frozen resources, as described in [25]. Each of these *artifacts* may be deployed in a *node* (a run-time physical object that represent a computational resource.) Thus, we allow representation of component internal parts—artifacts—inside a component box using the standard UML notation. The nodes where artifacts or components are to be deployed can also be represented, by means of a constraint associated to the corresponding element which specifies the identifier of the node.

2.2 Connectors

A connector defines a relationship between two or more components. This may be realized by something as simple as a direct procedure call, or by something as complex as a complete network-based multicast virtual circuit with adaptation and encryption.

The end points of connectors are called *roles*, which are attached to the corresponding components' ports. Roles, like ports, are typed so that a connector end can only be attached to a port that has a compatible type (both at the signature and behavioral levels.) Likewise, we will differentiate between incoming and outgoing roles, depending whether the services they contain are provided or required (no mixtures of provided and required operations are allowed in our model.)

In ADLs, connectors are first-class, typed entities with potentially rich specifications, and different from components. However, the notion of connector usually disappears in CBSD, since either they are direct calls and therefore supported by the underlying communication platform (e.g. the ORB—no matter how complicated it is), or they have some sort of computation and hence they are implemented by components.

In our proposal we will distinguish between components and connectors. Trivial connectors are denoted by directed Relationship arrows, with optional multiplicity at the ends. Other kinds of connectors (a complete taxonomy of software connectors is described in [14]) are denoted by dashed-lines boxes, similar to the ones used for components (but surrounded by dashed lines instead of solid lines). In the short notation, roles will be denoted by lollipops, whilst in the extended notation the full interface of the role will be described inside a dashed-line box protruding out of the connector box. Again, white lollipops represent provided interfaces, and solid ones represent required interfaces (Fig. 3). The same tags used for component interfaces in the extended notation can be used for connector interfaces to differentiate between required and provided interfaces.

By representing in a very similar way both components and connectors we try to highlight the similarities between both entities, while respecting their semantic differences: using dashed lines in the connector boxes

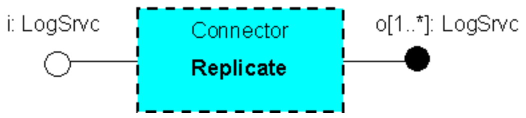


Figure 3: Connector Replicate in the short notation.

tries to resemble the concept of collaborations that connectors carry along. In addition, non-trivial connectors will have to be finally implemented by software components, and this graphical representation tries to address this fact.

2.3 Systems

Systems are collections of components and connectors, that implement a piece of functionality. Systems may have also interfaces, so they can be connected to other systems. In this case, it is necessary to explain and represent the mapping between the internal and external interfaces. The elements of this mapping are usually called *bindings* [5].

For representing systems we will use the UML *subsystem* element, since it has the semantics that we want our systems to have (see Fig. 6).

2.4 Properties

Quality attributes of the architectural elements of our model (components, connectors, ports, roles, systems, and styles) will be specified by 3-tuples (name, type,value), along the lines of the ODP standard [8]. UML tagged values will be used for that (see Section 3.3).

2.5 Styles

Architectural styles will be represented by constraints that define the set of rules that any given system (and its architectural elements) should fulfill in order to comply to a given style. Examples of styles include data-flow architectures based on graphs of pipes and filters, blackboard architectures, and n-tiered systems. OCL is the obvious candidate to express these constraints, since it is formal and it allows to express restrictions on any UML element using navegability.

3 Behavior

As we have said before, the description of the software architecture of a system or an application goes beyond the representation of its internal structure. In this section we shall discuss some of the functional and extra-funcional attributes that can be expressed with our proposal: behavior, choreography, and quality attributes.

Those properties will describe the specific characteristics of each architectural elements, and will be associated to them using UML's notes with the stereotypes <<behavior>>, <<choreography>>, and <<properties>>, respectively. Standard UML determines the way and notation in which the behavior and choreography of a system are described, in terms of the corresponding state charts and sequence diagrams. However, we did not want to restrict ourselves to any particular notation. Thus, the tagged value {notation = ...} should be included in all behavioral and choreography descriptions, specifying the particular notation in which they are written.

It is important to notice that in the provided ports and roles, the behavior, choreography and properties refer to the "actual" values of these characteristics. However, in required interfaces they mean the "expected" characteristics of the required service. When connecting a provided service to a required one (e.g. a connector end to a port or two ports by a trivial connector), their attributes should be tested for compatibility.

Figure 4 shows a graphical representation of the three attributes of port Account of component Bank.

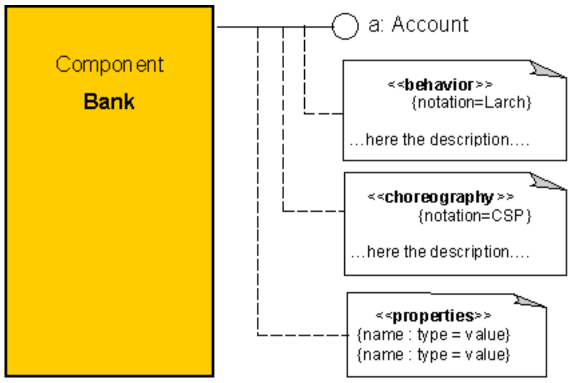


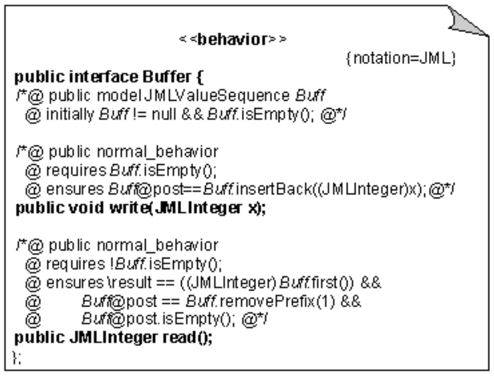
Figure 4: Representation of component attributes.

3.1 Behavior

The behavior of an architectural element defines the semantics of its provided and required operations. Since software components are black boxes, their behavior is determined by their interfaces, i.e. the types of their ports. Same applies for connectors, whose behavior is determined by the behavior of their roles. Thus, we will attach behavioral descriptions to ports and roles only, but not to any other architectural element.

A note with the stereotype <<behavior>> will be used for specifying behaviors. The notation in which the behavior is specified is not fixed a priori, it can be done using pre/post conditions expressed in OCL or Larch, state charts, or any other notation. A tagged value "{notation=...}" will indicate the notation used.

As a very simple example, consider an interface called Buffer, that describes the services provided by a one-place buffer, which implements operations write(), read(), and isEmpty(). Of course, the behavior of all these operations should be properly qualified so the correspond to a one-place buffer, and not to any general buffer. The following text corresponds to the behavior of the buffer expressed in JML [11].

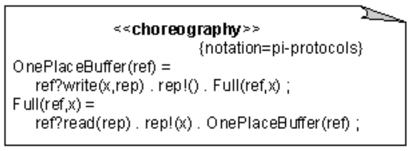


```
                    <<behavior>>
                                     {notation=JML}
public interface Buffer {
/*@ public model JMLValueSequence Buff
  @ initially Buff != null && Buff.isEmpty(); @*/

/*@ public normal_behavior
  @ requires Buff.isEmpty();
  @ ensures Buff@post==Buff.insertBack((JMLInteger)x);@*/
public void write(JMLInteger x);

/*@ public normal_behavior
  @ requires !Buff.isEmpty();
  @ ensures \result == ((JMLInteger) Buff.first()) &&
  @         Buff@post == Buff.removePrefix(1) &&
  @         Buff@post.isEmpty(); @*/
public JMLInteger read();
};
```

3.2 Choreography

Behavioral specifications are not enough for specifying the semantics of components. Part of this specification is the set of "protocols" they implement. A protocol specifies what messages a component sends and receives when it collaborates with other components, and the partial order in which those messages can be sent and received—which is now commonly called the *choreography*. Message sequence charts, Petri nets, or process algebras (such as CCS, CSP or π-calculus) can be used for describing choreographies.

In our proposal, choreographies can be attached to ports and roles by means of notes with the stereotype <<choreography>>. Again, the tagged value "{notation=...}" will indicate the notation used.



```
<<choreography>>
                    {notation=pi-protocols}
OnePlaceBuffer(ref) =
    ref?write(x,rep) . rep!() . Full(ref,x) ;
Full(ref,x) =
    ref?read(rep) . rep!(x) . OnePlaceBuffer(ref) ;
```

As an example, Figure 3.2 shows the description of the choreography of the one-place buffer interface defined above, written using a π-calculus-based notation [3].

Usually, protocols can be decomposed into dyadic sessions, each one defining the interactions between pairs of interfaces, i.e. each session describes the *rôle* a component plays in its collaboration with other components. Hence, it is enough to assign choreographies to single interfaces (ports or roles). However, there are situations in which it is important to describe how the separate sessions interleave, and therefore a global choreography should be specified (cf. [3]).

For instance, we may have a component Bank that supports interfaces Account and Resource. The first one supports operations getBalance(), deposit(), and withdraw(), while the second interface provides the typical operations for two-phase commit transactions: begin(), prepare(), commit(), and rollBack(). Now, if component ATM makes use of component Bank, we may be interested not only in checking that the operations in the interfaces are separately called in the right order, but to know which Account operations are subject to transactions and which ones are not. Please note that this information is missing if only the pairwise interactions are shown. Therefore, the choreography of the ATM should be globally specified. This can be done in our approach by attaching a <<choreography>> element to a component, a connector, or a system, in addition to the <<choreography>> notes that describe the protocols obeyed by their individual ports or roles (these ones describe pairwise interactions only).

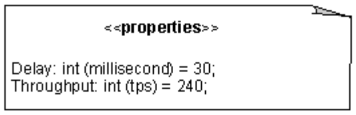
3.3 Quality attributes

QoS is an abstraction covering aspects of non-functional behavior of a system. A *QoS characteristic* represents some aspects of the QoS of a system object that can be identified and quantified. Examples of QoS characteristics are delay, response time, stability and usability of information, freshness and accuracy of information, availability, and so on [9]. The QoS characteristics in OMG request for proposal about UML Profile for modeling QoS and fault tolerance [19] are classified as:

- Time-related (delay, freshness)
- Importance-related (priority, precedence)
- Capacity-related (throughput, capacity)
- Integrity related (accuracy)
- Fault-tolerance (mean-time between failure, mean-time to repair, number of replicas).

A *QoS requirement* expresses one or more values associated with the characteristic(s), the nature of the value (i.e. upper limit, threshold, target, etc.), actions to be taken as result of reaching limit/target/etc., the nature of any agreement reached during negotiation, and so on. In the context of ODP systems, a QoS requirement is expressed using the QoS relation structure. A *QoS relation* states mutual obligations of an object and its environment. QoS relations specify two kind of QoS requirements: *expected* and *provided* QoS characteristics. The first one specifies what the object expects from the environment and the second one what the object guarantees to the environment. QoS relations are fundamental building blocks for the expression of QoS requirements. Provisions are met as long as expectations are met.

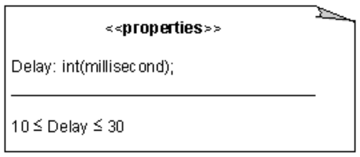
In our work we propose that provided QoS requirements are specified as properties related to the provided ports of a component or connector. Similarly, expected QoS requirements are related to the required ports. Each property is specified as a 3-tuple (<QoS-name>, <QoS-type>, <QoS-value>). For example, the requirement that the maximum expected delay is 30 milliseconds and the expected throughput is of 240 transaction per second can be specified as:



```
<<properties>>

Delay: int (millisecond) = 30;
Throughput: int (tps) = 240;
```

Please note that units can be specified in brackets besides the type of the variables.

It is also possible to include ranges and other expressions describing the possible values of the variables. For example, the relation that the maximum delay is 30 milliseconds and the minimum delay is 10 milliseconds can be specified as:

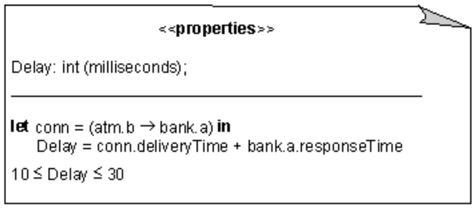


```
<<properties>>

Delay: int(millisecond);
------------------------------
10 ≤ Delay ≤ 30
```

Global properties about QoS can be related with end-to-end requirements and *local properties* with the requirements of the objects that compose a system. A key concept about QoS is that end-to-end requirements dictate internal requirements [9]. This means that there is a direct relationship between the end-to-end and the internal requirements. For example, the OMG Audio/Video Specification [16] states that network-level requirements are determined by application-level requirements. So, an important question for us is if it there is such a mapping between global and local properties, and if it exists, how it is specified.

A set of QoS relations applying to a composition of components is composable if and only if the conjunction of obligations logically implies the conjunction of expectations. Furthermore, a desirable property about QoS for a system architecture is that it should be possible to derive the QoS of a composition of components from the QoS of its individual components. If QoS requirements apply to a composition of objects, a refinement mechanism can be helpful in order to derive the set of QoS requirements applying to the individual components of the composition.

Our notation also allows us to express global properties of the system in terms of relations or equations that involve local properties of components or connectors. For example (see Section 4), in the following figure the property at the ATM's (automatic teller machine) expected port Account could be related with the delay of the bank and the connector between bank and atm.



```
<<properties>>

Delay: int (milliseconds);
------------------------------
let conn = (atm.b → bank.a) in
    Delay = conn.deliveryTime + bank.a.responseTime
10 ≤ Delay ≤ 30
```

These relationships implement some kind of traceability, describing how local properties influence global properties.

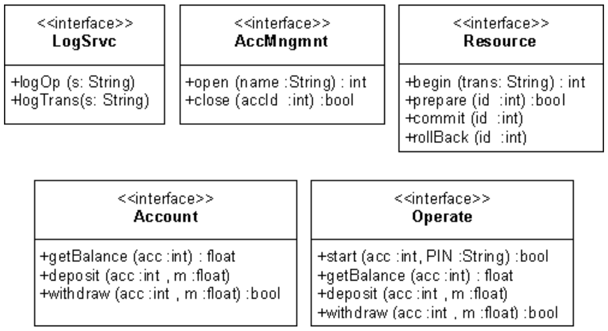
There may be many alternative notations for specifying QoS properties. For example, OCL is suggested in [19] and Lamport's Temporal Logic of Actions (TLA) is used in [9]. Therefore, the notation tag—as for behavior or choreography—can be used to specify the type of notation in which the quality attributes are described.

In our work, the proposed UML extensions for specifying quality attributes of software architectures allow us to define in a flexible way the properties associated to the architectural elements of the system. Open issues are the notation finally used for specifying the QoS attributes, and how global properties are related with local properties in a composed system—for instance, the choreography that specifies the messages exchanged between the elements should play a vital role in these relationships. Besides, tools can be developed for consistency checking between global and local properties. The correct identification of architectural style or some pattern of a object composition can also help establishing the relationships between global and local properties. Thus, for example, if we have a pipeline of interconnected components, the delay time can be calculated as the sum of the individual delays of components, plus the delivery time of the connectors that join them.

Finally, there is the distinction pointed out in [20] between those quality attributes observable at run-time (such as availability, accuracy, or timeliness), and those that show during the system's lifetime (testability, upgradeability, or openness). In this paper we have concentrated on the first kind of attributes. How to incorporate information about the second kind of attributes is the subject of a forthcoming paper.

4 Example application

The selected example application is a simplified version of the Automatic Teller Machine (ATM) problem, described in [24] as one of the candidate model problem in software architectures. In this application, a banking network should be designed such that customers have accounts in banks and can access them either directly or by using ATMs. The system requires appropriate record keeping and fault-tolerance provisions.

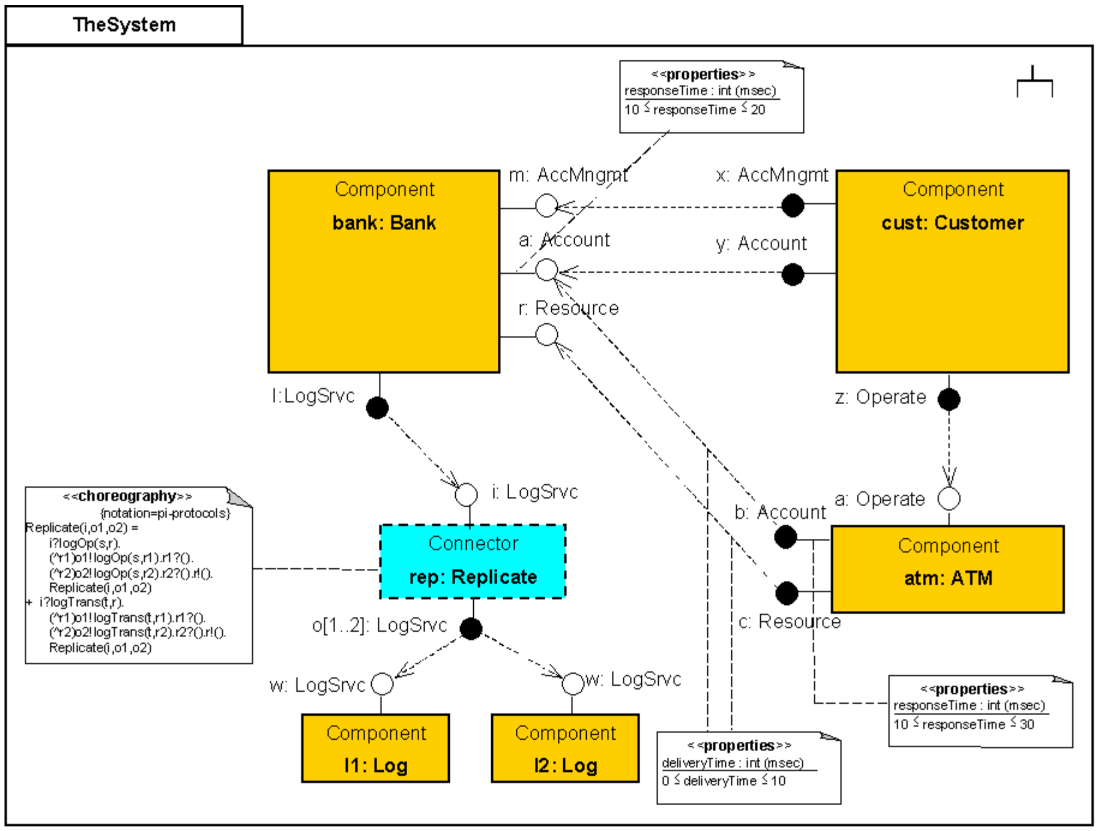


<<interface>> LogSrvc
+logOp (s: String)
+logTrans(s: String)
<<interface>> AccMngmnt
+open (name :String) : int
+close (accId :int) :bool
<<interface>> Resource
+begin (trans: String) : int
+prepare (id :int) :bool
+commit (id :int)
+rollBack (id :int)
<<interface>> Account
+getBalance (acc :int) : float
+deposit (acc :int , m :float)
+withdraw (acc :int , m :float) :bool
<<interface>> Operate
+start (acc :int, PIN :String) :bool
+getBalance (acc :int) : float
+deposit (acc :int , m :float)
+withdraw (acc :int , m :float) :bool

Figure 5: Example system: Interfaces.

For designing the architecture of the system we have identified four kinds of components and one connector (Fig. 6). The services provided and required by each components are described in Fig. 5. Component Bank will take care of the accounts, using interface AccMgnmt for creating and closing accounts, and interface Account for normal account operations. Interface Resource is used to provide a transactional service when accessing the accounts from ATMs. In order to implement the first requirement of the problem (record keeping) we have defined component Log, which provides interface LogSrvc for logging both transactions and operations. The second requirement (fault-tolerance) has been implemented by replicating the logs and by using transactions in operations between the ATMs and the bank accounts. Connector Replicate is in charge of duplicating the log outputs of component Bank.

Apart from the architectural elements, we have included in Fig. 6 four notes for illustrating their use. The first one specifies the choreography of the connector, describing its dynamic behavior. As we can see, the connector first log into its output o[1], then into o[2], and then replies back to its calling component. Other behaviors could have been alternatively specified, such as for instance one in which the connector first replies to its calling component, and then it start logging into its outputs (either sequentially or in parallel.) The other three notes describe QoS attributes. One specifies that the connections between the ATM and the bank should have a delivery time between 0 and 10 milliseconds for carrying the requests and responses (independently from the mechanism communication, ORB, or platform they are implemented in). The note attached to the provided port a of component bank specifies the actual response time range for the operations in that port. Finally, the last note describes a required QoS attribute of the interface Account that component atm uses. The note imposes the requirement that it should have a response time in the range of 10 and 30 milliseconds for each of its requested operations (no less, no more). As we mentioned before, this notation is devised for allowing checking tools prove that the composition is consistent, i.e. a tool can check that the expected properties of the atm are met by the corresponding services provided by the bank and the connector that joins them.



```
Replicate(i,o1,o2) =
    i?logOp(s,r).
    (^r1)o1!logOp(s,r1).r1?().
    (^r2)o2!logOp(s,r2).r2?().r!().
    Replicate(i,o1,o2)
+  i?logTrans(t,r).
    (^r1)o1!logTrans(t,r1).r1?().
    (^r2)o2!logTrans(t,r2).r2?().r!().
    Replicate(i,o1,o2)
```

Figure 6: The software architecture of the example system.

The specification of the choreography of the components and connectors, apart from describing their dynamic behavior, also provides very useful information when reasoning about the QoS properties of the system. For instance, in Fig. 6 we can see the messages exchanged between the component bank and the two log components through connector Replicate. Using this information, and the response time of each log components, we could calculate the expected response time of the port l:LogSrvc of component bank.

5 Related work

There are two different research lines that can be related to ours: the use of UML for describing Software Architectures, and the documentation of COTS components for describing their functional and extra-functional aspects so they can be incorporated into existing software architectures.

In the first place, several authors have pointed out the problems that UML currently has for modeling Software Architectures, as well as the possible ways to overcome them using the standard UML 1.x facilities [5, 10, 13, 22]. As a common conclusion, there is no solution without extending UML. In this sense, let us see what UML 2.0 is able to offer, although in the initial responses to the Superstructure RFP some problems still remain.

On the other hand, the work by Bran Selic and Jim Rumbaugh about modeling real-time systems with UML [23] is probably the closest to our work, since they reduce the domain (to real time systems) and introduce some non-standard UML constructs that allow an effective way to represent the software architecture of such systems.

There is also some work done in the UML profile for EDOC [18], which fits into the more ambitious goals pursued by the Model Driven Architecture (MDA) initiative by OMG [17]. MDA tries to address the fact that historically the integration between the development tools and the deployment into the middleware frameworks has been weak. This is now beginning to change by using key elements of MDA, namely specific models and XML DTDs that try to cover the complete life cycle of software systems, as well as *profiles* that provide mappings between the models used in various life cycle phases. The facilities provided by XMI for data exchange between different models are used in the MDA for marrying the worlds of modeling (UML), metadata (MOF and XML) and middleware (UML profiles for CORBA, Java, EJB, etc.). XMI also provides developers focused on implementation in Java, Visual Basic (VB), HTML, etc., a natural way of taking advantage of the software architecture and engineering discipline when a more rigorous development process is desirable. One of the key elements in this chain is the use of UML for modeling systems.

With regard to the documentation of COTS components, there are a number of proposals that try to enhance the information that they currently convey. To mention a few, IBM is working on a proposal for documenting their large grained components (`www.ibm.com/software/components`), and there are several nice proposals from SEI (`www.sei.cmu.edu/`) claiming better component documentation. Bastide et al. [2], and Canal et al. [3] are among the authors that propose IDL extensions for dealing with protocol information in CORBA environments, using formal languages—Petri nets and π-calculus, respectively. Finally, some other works [1, 6, 7] try to deal with the extra-functional properties of components, adding this kind of information to commercial components, with the goal of relating it to the architectural specification of the applications.

6 Conclusions

The need for expressing the software architecture of a complex system is a critical issue. However, most architectural description languages are either too formal to be effectively used in industrial environments, or too simple and imprecise to provide useful information and allow the use of checking tools. Our proposal tries to address these issues, offering extensions to UML for the graphical representation of software architectures. We allow the description of the system's structure in terms of its constituent components and connectors, as well as the specification of their behavior. Quality of service properties can also be described in our approach by means of QoS attributes.

In this paper we have basically focused on the graphical representation of the behavior and QoS attributes of the system and its architectural elements, but without determining a precise notation for writing them. The search of the appropriate notations for the specification of the individual elements, as well as the development of tools to support them is a matter of further research, specially those that allow the automatic consistency checking of the composed systems.

Future activities also include the consideration of other kinds of extra-functional requirements—such as security, or mantainability—, how to include them into the system's graphical description, and how they can be mapped to the local properties of components. Other open issue is the integration of our graphical representations with proper tools and mechanisms, so they can form part of the software development's life cycle. For instance, how to translate the information provided in the graphical description of the system into XML or other intermediate languages, and how to use it to search for the appropriate COTS components living in repositories, using the service currently offered by some COTS traders [7, 6].

References

[1] Carina F. Alves, Nelson S. Rosa, Paulo R. F. Cunha, Jaelson F. B. Castro, and George R. R. Justo. Using non-functional requirements to select components: A formal approach. In *Proc. of the Fourth Iberoamerican Workshop on Requirements Engineering and Software Environments (IDEAS'01)*, San José, Costa Rica, April 2001.

[2] Rémi Bastide, Ousmane Sy, and Philippe Palanque. Formal specification and prototyping of CORBA systems. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 474–494. Springer-Verlag, 1999.

[3] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, José M. Troya, and Antonio Vallecillo. Extending CORBA interfaces with protocols. *The Computer Journal*, 44(5):448–462, October 2001.

[4] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 1999.

[5] David Garlan, Shang-Wen Chen, and Andrew J. Kompanek. Reconciling the needs of architectural descriptions with object-modeling notations. To appear in *Science of Computer Programming*, 2002.

[6] Luis Iribarne, José M. Troya, and Antonio Vallecillo. Trading for COTS components in open environments. In *Proc. of the 27th Euromicro Conference*, pages 30–37, Warsaw, Poland, September 2001. IEEE CS Press.

[7] Luis Iribarne, Antonio Vallecillo, Carina Alves, and Jaelson Castro. A non-functional approach for COTS components trading. In *Proc. of the Fourth Workshop on Requirements Engineering (WER'01)*, Buenos Aires, Argentina, November 2001.

[8] ISO/IEC. RM-ODP. Reference Model for Open Distributed Processing. Rec. ISO/IEC 10746-1 to 10746-4, ITU-T X.901 to X.904, ISO/ITU-T, 1997.

[9] ISO/IEC. Open Distributed Processing – Reference Model – Quality of Service. Commitee Draft ISO/IEC 15935, ISO, 1998.

[10] Cris Kobryn. Modeling components and frameworks with UML. *Communications of the ACM*, 43(10):31–38, October 2000.

[11] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999. JML web page: `http://www.cs.iastate.edu/~leavens/JML.html`.

[12] Nenad Medvidovc and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[13] Nenad Medvidovic, David S. Rosenblum, Jason E. Robins, and David F. Redmiles. Modeling software architectures in the unified modeling language. To appear in *ACM Transactions on Software Engineering and Methodology*, 2002.

[14] Nikunj Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proc. of ICSE 2000*, pages 178–187, Limmerick, Ireland, June 2000.

[15] B. Craig Meyers and Patricia Oberndorf. *Managing Software Adquisition: Open Systems and COTS Products*. Addison-Wesley, 2001.

[16] OMG. *Audio/Video Streams Specification v1.0*. Object Management Group, January 2000. OMG document `formal/00-01-03`.

[17] OMG. *Model Driven Architecture. A Technical Perspective*. Object Management Group, January 2001. OMG document `ab/2001-01-01`.

[18] OMG. *A UML Profile for Enterprise Distributed Object Computing V1.0*. Object Management Group, August 2001. OMG document `ad/2001-08-19`.

[19] OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, Request for Proposal*. Object Management Group, January 2002. OMG document `ad/2002-01-02`.

[20] Otto Preiss, Alain Wegmann, and Jason Wong. On quality attribute based software engineering. In *Proc. of the 27th Euromicro Conference*, pages 114–120, Warsaw, Poland, September 2001. IEEE CS Press.

[21] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[22] Bran Selic. On modeling architectural structures with UML. In *Proc. of the UML'2001 Workshop on Software Architectures and Requirements Engineering (STRAW'01)*, Toronto, Canada, May 2001.

[23] Bran Selic and Jim Rumbaugh. Using UML for modeling complex real-time systems. Available at `http://www.rational.com/media/whitepapers/umlrt.pdf`, March 1998.

[24] Mary Shaw, David Garlan, Robert Allen, Dan Klein, John Ockerbloom, Curtis Scott, and Marco Shumacher. Candidate model problems in software architecture. Available at `http://www.cs.cmu.edu/afs/cs/project/compose/www/html/ModProb/`, January 1995.

[25] Clemens Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, 1998.