Domain specific modeling and analysis

Jacob, J.F.

Chapter 5

Component Coordination in UML

Authors: Frank de Boer, Marcello Bonsangue, Joost Jacob

5.1 Introduction

Modeling is an essential part of large software projects. The Unified Modeling Language (UML) has become the de-facto standard language for specifying, modeling and documenting software systems, visualizing software systems. The basic innovative ideas of UML, which are the main reasons for its popularity, are the unification of the concepts and notations used in the life-cycle of software development as well as the recognition of the importance of modeling and analysis as a means to improve quality. UML consists of a number of diagrams used for expressing the goals of the system (use case diagrams), for specifying the structure of the system (class diagrams) and the behavior of the system (state diagrams, activity diagrams, sequence diagrams).

In this paper we introduce a formal model of components in UML. This model has been developed in the context of the European IST project OMEGA. The aim of this project is the correct development of real-time embedded systems based on formal techniques. The approach followed is based on a formal semantics of a suitable subset of UML which includes class and state diagrams, a version of the Object Constraint Language, use case diagrams, and live sequence charts (an extension of UML's sequence diagrams [DH01]). The semantics of the UML subset, here called Ω-UML, is defined in terms of a formal interleaving semantics obtained by associating with each model of Ω-UML a symbolic transition system [DJPV03].

Our component model generalizes the basic concepts of object-orientation by providing additional structuring and abstraction mechanisms which allow a modeling discipline and the application of formal techniques based on interfaces. More specifically, it allows to structure the class diagrams of a UML model into components and to abstract from the internal details of these encapsulated class diagrams. Because of the encapsulation provided by components we can compose them hierarchically in a natural manner.

In this paper we also discuss the formal semantics of our component model. First we discuss the formal relation between a system of components and the underlying UML class diagrams. This relation is defined in terms of a reduction which 'compiles away' the additional structuring and abstraction mechanisms provided by components. However, we also show how we can describe the externally observable behavior of a component at a high-level of abstraction and compositionally in terms of its structuring and abstraction mechanisms. This latter view provides the formal justification of the modeling to interfaces discipline and it provides a formal basis for the application of formal techniques to components.

Furthermore, we discuss different coordination patterns in the context of our component model. First we show how high-level components can be used to model the low-level coordination patterns underlying the computational model of Ω-UML. These coordination patterns form an intricate combination of the asynchronous communication supported by an event-driven computational model (along the lines of the Actor model [Agh86]) and the synchronous communication supported by the usual rendez-vous mechanisms of operation calls in object-orientation. Finally, we show how to generalize our component model to a model of component coordination based on mobile channels which allow a clear separation of concerns between coordination and computation.

This paper is structured as follows: Section 2 introduces the component model. In Section 3 we discuss the formal semantics of our component model. Section 4 then proceeds with a discussion of how to model the low-level coordination patterns underlying Ω-UML by means of high-level inter-component coordination. Finally, Section 5 discusses a generalization to mobile channels. In Section 6 we draw some conclusions.

5.2 A component model

In this section, we introduce an extension of UML addressing the area of component-based software systems. Following Szypersky [Szy02], we see a software component as a *unit of composition* with well-defined interfaces, that can be independently developed and subject to composition by third parties. In the context of UML, this means that we consider a component as a mean to provide a high-level software abstraction like that of a *module*, which encapsulates its internal structure and which provides interfaces specifying the provided and required operations. The rationale is to provide a structuring and abstraction mechanism which allows a modeling discipline based on interfaces.

More technically, a *component* is a UML classifier, which is intended to be self-contained and re-usable during development and deployment. It is identifiable by a name but it has no attributes and operations. It cannot be instantiated or be part of associations, but it can be generalized since it has a type, defined by the set of its provided and required interfaces. This means that a component is a *unit of substitution* that can be replaced by a component that offers at least the same provided interfaces and demand at most the same required interface.

A *component interface* is just a UML interface, that is, a non-instantiable classifier with operations and attributes. We distinguish between two kind of component interfaces: *required interfaces* and *provided interfaces*. A provided interface specifies a set of operations that the component offers to the environment. A required interface specifies a set of operations that are needed by the component to guarantee the correct functionalities of some provided interfaces. We allow for generalization relations among component interfaces.

A component is also a package, and therefore a structural em unit of abstraction of the classes realizing its behavior. Other UML elements may be owned by a component. In particular, other components may be owned by a component allowing for hierarchical specifications. Encapsulation of the internal structure is guaranteed because interaction points with its environment are exclusively defined via ports, the software concept equivalent of the hardware port on a board.

A *port* is a class and also an interface, that is, it is an instantiable interface. A components owns a set of ports, and each port owns a set of the component provided interfaces, and a set of the required interfaces. The same component interface may be owned by more than one port. The set of ports defines the border between the internal implementation of the component and its environment. Internal classes may realize a port or depend on a port.

Incoming communications defined in the provided interface of a port are handled within instances of an internal class of the component realizing that port. If a class realizes a port it realizes also one of its provided interfaces. We assume that at most one internal class may realize a provided interface of a port (but we allow for different classes to realize the same provided interface if each class realize a different port). A port introduces an indirection, and each request of instantiation for that port is resolved at run-time by instantiating an internal class realizing a provided interface owned by the port (which class is resolved statically by the type of the object expected by the requester from the instantiation). This indirection mechanism abstract from the actual implementation of an operation and allows for a very late binding of an operation implementation with its declaration in a component interface. We call *port instances* the object instance of internal classes instantiated by a port.

If an internal class depends on a port then it depends also on one of its required interfaces. From the environment point of view, outgoing communications of an object instance of an internal class are identified with communications from the port owning the required interface on which that class depends. Communications at the border (that is, between two port instances) are observable.

A component has two structural vies: a black-box view and a white-box view. In the *black-box view*, only the component provided and required interfaces and their grouping into ports is visible. Optionally, behavioral elements such as a state machines may be attached to each port, to define more explicitly a sequence of operation calls. For a black-box component it must hold that every type or class used in a provided interface must be declared in one of the provided or required interface of the component itself. This self-containment property, together with not allowing generalization across the border, ensures a complete encapsulation of the internal implementation. Notationally, a black-box component is drawn as a classifier rectangle with in the right hand corner a component icon: a rectangle with two smaller rectangles protruding from its left hand side. Ports are shown as small squares on the edge of the component rectangle, with association to interfaces, shown as labeled ball and socket for the provided and required interface interface, respectively. Provided and required interface can also be shown more explic-

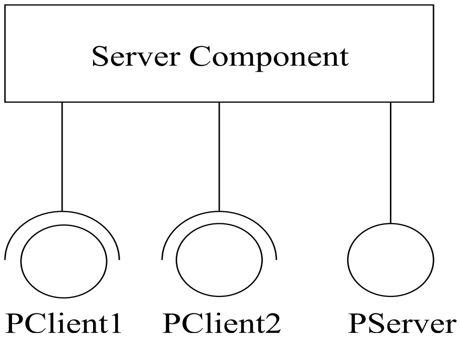

Figure 5.1: A black-box view of a component

itly as the classifier rectangles. Figure 5.1 shows an example of a component with two ports: one with a provided interface and another with a provided and a required interface.

Black-box views of components are used in *component system diagrams* to visualize the structural connections in a component-based system. There can be dependency relations from a provided interface of a component to a required interfaces of another component (but not vice-versa). Optionally, a coarser specification of the structural collaboration can be made using *connectors*. A connector is a specialized association between ports, used to indicate that all required interfaces of a port must be compatible with the provided interfaces of the other connected port. The wiring between components in a system is used at run-time for the instantiation across components. If an internal objects (i.e., an instance of an internal class) requests the instantiation of a port P with interface I on which it depend, then this request is resolved at deployment time in a request for instantiation of the port Q with a provided interface wired with the required interface I of P. In other words, a port through its required interfaces act as placeholders for port names that become known only at deployment time, when connectors or dependency relations are statically fixed in a component system diagram.

Figure 5.2 shows a component system diagram. Dependency relations between provided and required ports implicitly given by the ball-in-socket notation. The association between a port of component B and one of component C is a connector: the set of provided and required ports of those ports must be compatible.

In the *white-box view*, the internal elements of a component are revealed,

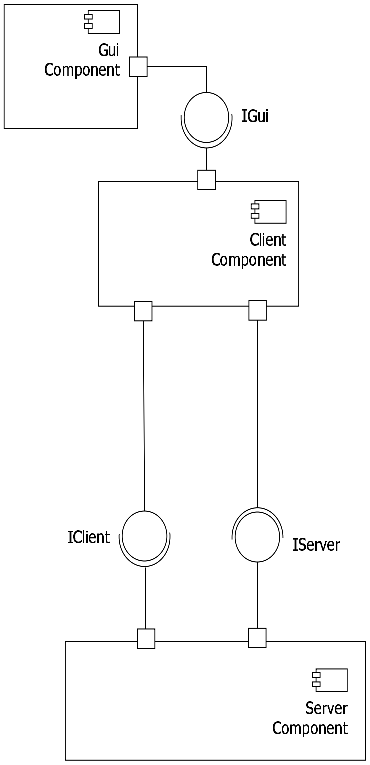

Figure 5.2: A component system diagram

in order to show the implementation of the external behavior of the component ports. To this end, dependency and realization relations must be given between internal classes or internal components and the component ports: a dependency relation provides information about an internal class implementation in terms of its required services. An internal class can *depend* on several ports. Since ports are UML interfaces we can draw dependency relations from an internal class to a port, with the intended meaning that the class depend on one of the required interface of that port. On the other hand, a required component interface or a port cannot depend on something inside a component. That would be a design error since the component can supply the needed services by itself. Since ports are UML interfaces, we can draw realization relations from internal classes to ports. An internal class can *realize* several port, but port can only realize provided interfaces. A connector between the port of an internal component and an external port is used to graphically show the export of a port of an internal component to the environment.

Figure 5.3 show the white-box view of component A: the upper port is connected with a port of the internal components B, while the other port is realized by class C. Class C depends on the same port and also on a provided interface of the internal component B.

Run-time component interaction configurations are modeled in architectural diagrams. They are snapshots that can be used to describe the initialization of a component system, invariant properties of the configuration, and others useful runtime characteristics. In architectural diagrams only instances of ports, ports and components are shown, together with their relationships. Instances of ports handle all interactions from the environment into the component they belong to, as well as the interaction between different port instances. Interaction from the inside of a component to the environment is handled by the ports of the component (and not by their instances).

In Figure 5.4 we show an architectural diagram of the component system depicted in Figure 5.2. Port instances are represented by filled squares, while port classes are denoted by plain squares. Arrows denote directed relations either between port instances (representing the possibility of executing operation calls from a port instance to another one), or from port classes to port instances (representing the possibility of executing operation calls from the internal of the component to the port instance of another component).

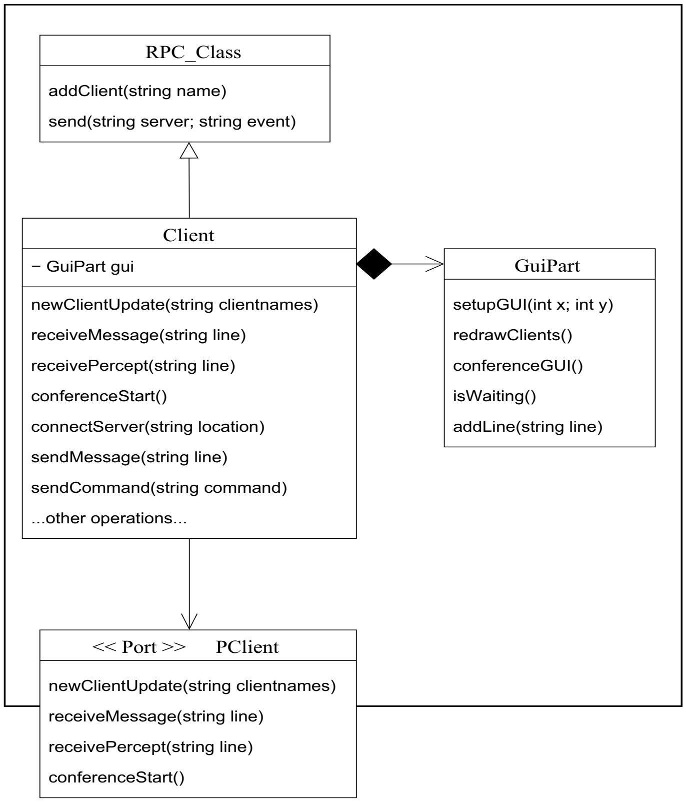
RPC_Class
addClient(string name)
send(string server; string event)
Client
− GuiPart gui
newClientUpdate(string clientnames)
receiveMessage(string line)
receivePercept(string line)
conferenceStart()
connectServer(string location)
sendMessage(string line)
sendCommand(string command)
...other operations...
GuiPart
setupGUI(int x; int y)
redrawClients()
conferenceGUI()
isWaiting()
addLine(string line)
<< Port >> PClient
newClientUpdate(string clientnames)
receiveMessage(string line)
receivePercept(string line)
conferenceStart()

Figure 5.3: A white-box view of a component

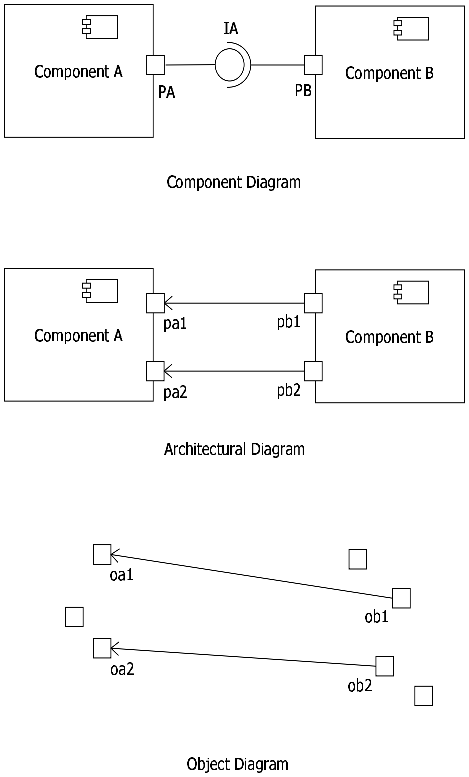

Figure 5.4: An architectural diagram of a component system

5.3 Ω-UML

Building embedded real-time systems of guaranteed quality, in a cost-effective manner, is an important technological challenge. There is a general agreement that a means to achieve this is a model-based approach. UML aims at providing an integrated modeling framework encompassing structural descriptions, as well as behavioral descriptions. Although there is a large number tools available that implement a dynamic semantics of UML, none of these tools integrates state-of-the-art formal validation tools, as required in many industrial sectors for a proper development process.

The aim of the European IST project OMEGA is the correct development of real-time embedded systems based on formal techniques. The approach followed is based on a formal semantics of a suitable subset of UML which includes class and state diagrams, Object Constraint Language, use case diagrams, and live sequence charts (an extension of UML's sequence diagrams [DH01]). The semantics of the UML subset, here called Ω-UML, is a formal interleaving semantic obtained by associating with each model of Ω-UML a symbolic transition system [DJPV03]. Due to space restrictions, we only sketches here the main concepts of Ω-UML and their intuitive semantics.

Similarly to the standard UML [SWB03], classes in Ω-UML may be either active or passive. An *active object* (i.e., an instance of an active class) is like an event-driven task, which processes its incoming requests in a first-in-first-out fashion. It has only one thread of control, so that active objects are internally sequential. As only one message may be treated at a time, there must be a mechanism for queuing the calls. For a *passive object* only one operation may be active at a time, and, although passive, it has certain degree of control over the invocations made toward them, as explained later in this section. Furthermore, in Ω-UML all objects are assumed to be *reactive*, that is, their behavior can be made dependent on the current state of the system.

In Ω-UML *state-machines* are used to describe the computational behavior of the instances of a class. These state machines are composed of transitions which are labeled by a (guarded) *trigger* and an *action*. A trigger specifies the reception of an *operation call*. An action involves assignments to the attributes, class instantiation and operation calls.

The semantics model of Ω-UML captures three different kinds of inter-object communication: share variables (via public attributes), synchronous (via triggered operation calls, i.e. operation calls whose return value depend on the current state of the system) and asynchronous (via signal events).

The execution of a *synchronous* operation call involves a *rendez-vous* between the sender and the receiver of the call: First, the sender and receiver of the call have to synchronize on the execution of an operation call by the sender and a corresponding trigger by the receiver. Such a synchronization results in the sending of the actual parameters which are stored by the receiver in the corresponding formal parameters of the operation. During the execution of the operation by the receiver, the sender is suspended. Upon termination of the call, the return value is send back to the sender, after which both sender and receiver resume their own execution.

On the other hand, an *asynchronous* operation call is stored in the *event-queue* of the receiver. The execution of a trigger involving an asynchronous operation consists of checking whether a corresponding operation call appears as the first element of the event-queue (of the receiver) and storing its actual parameters in the formal parameters. It suspends otherwise.

The above communication mechanisms between active and passive objects is coordinated by means of *activity groups*. Activity groups are run-time components which are created dynamically. Each object belongs to an unique activity group, and each activity group contain exactly one active object at each time.. The objects of an activity group share both one thread of control and the event queue. The sharing of control means that at most one object of the group is executing. Control is passed on by a synchronous operation call to another object belonging to the same group. On the other hand a synchronous call to an operation of an object not belonging to the same activity group suspends the executing object and a fortiori its activity group. An asynchronous operation call to an object will be stored to the event queue of its activity group.

5.3.1 Components in Ω-UML

In order to represent all the aspects important for real-time system design at an appropriate level of abstraction, Ω-UML incorporates the component model introduced in Section 5.2. The model allow the definition of visibility and communication constraints needed for the design of large systems and provide a basis for proper abstraction, compositional refinement and verification.

To properly model the communication mechanism of a component-based system, the action language of Ω-UML is extended to allow actions for port instantiation. They are of the form

$$x := new(P)$$

where P is the name of a port owned by a component and x is an attribute of type compatible with one of the required interface owned by the port P. This action can be executed by any object instance of an internal class depending on the port P.

A component system diagram can be reduced to a large class-based Ω-UML model, using information from the white-box and black-box view of each component involved in the diagram. The basic idea is to recursively (because of the hierarchical structure of a component) transform each component into a the class it encapsulate. The classifiers for ports, component interfaces and components are omitted from this diagram, as well as all relations on which they are involved. Action for port instantiation $x := new(P)$ are transformed into action for class instantiation $x := new(C)$, where C is the name of the class indirectly instantiated by the port to which P is connected.

The basic semantic model obtained in this way is not compositional with respect to the concept of component. It is formalized in terms of a translation relation on object-diagrams which specify for each existing object the values of its attributes and the values of some system variables which encode some relevant control information (such as the current state in the associated statechart). Transitions are labeled by external events or by a label indicating an internal computation step. An external event is of the form

callee.op(caller,parameters)

denoting the call of a (synchronous or asynchronous) operation op by the caller to the callee with a list parameters, or

caller.return-op(callee, actual-parameters)

denoting the return from *callee* of a synchronous operation call *op*, initially called by *caller*. This semantics defines a very fine-grained notion of observability, making roughly every choice point and every interaction between distinct objects observable.

A more abstract notion of observability should be preferred, to capture only those interactions between the component and its environment. This can be obtained by transforming the labels of above transition system as follows. If the callee is a port instance then the transition step should be observable. In case the caller is also a port instance, the label should not change, otherwise *caller* should be renamed to the port on which its class depend. All other steps should be treated as internal steps. By means of a corresponding projection operation on traces of events we can define the traces of a system of components in a compositional manner along the lines of the compositional trace semantics of CSP [SS99] (generalized to components and object-orientation).

5.4 Intra-component coordination

In this section we discuss how our component model can be used in UML to express the coordination mechanism between active and passive classes as prescribed by Ω-UML by means of activity groups. That is, we see a component as a *unit of reaction*, which statically determines the run-time coordination of a reactive system with active and passive objects. Instances of an active classes are like Real-Time Operating System tasks, having their own thread of control, and running concurrently with other instances of active classes. Instances of passive classes are more like sequential objects whose operations are under the control of the active object that controls the caller.

In this section we will concentrate on the basic communication mechanism of UML: objects communicate and synchronize only via synchronous operation calls. For modeling purposes this suffices because asynchronous operation calls can be modeled by synchronous ones.

The basic idea is to replace an activity group from a model of Ω-UML by a component with a port which describes the coordination model of this activity group, that is, port instances will generate and coordinate activity groups. Since activity groups do not provide any encapsulation, the port will own a provided interface for each class specifying the activity group. The port instance is the only object that external objects can interact with, in the style of ordinary message interaction: p.operation(parameters). Further, the port instance p does not have attributes that are accessible by external objects.

Interactions between an external object and an internal object are delegated via the component port instance p: The external object is now interacting with p instead of with an internal object directly: every operation call

callee. op(parameters)

to an internal object callee is transformed into a call

p.op (caller, callee, parameters)

which involves the storage of the corresponding call to the operation op in the pending request table discussed below and which is immediately followed by a transition with a trigger

return-op(return-value)

which will involve the return from the call op by the callee and the reception of the return value.

On the other hand, the internal objects have to be modified so the port-instance can be informed about what triggers they will accept in their current state. We do so by adding to every state of an internal object a loop consisting of a *poll*() trigger and a return statement that returns the names of the triggered operations that can be accepted in the state.

Finally, a component port is an instance of a port class with an attribute which stores operation calls from external objects to objects of the activity group. This pending request table has four columns: every row has entries that consist of the caller object, the callee object, the operation name, and a sequence of actual parameters. Furthermore a port will contain other attributes for expressing certain relevant information about the state of the activity group, e.g., its objects, the executing object, etc.. This additional information will be used in the method *select* that selects an object from the set of internal objects for execution. A port has also a method *choose* that can select an entry in the pending request table. These two methods together will implement the coordination mechanism used and will involve a particular scheduling policy.

The behavior of a port consists of two main loops: one for receiving operation calls from external objects and storing these in the pending request table and one for dispatching calls from the pending request table and for forwarding these calls to an internal object and returning the result to the external object.

The first loop starts with a transition for each operation op that consists of a corresponding trigger

op(parameters)

where the list of parameters contains the caller and callee of the call. This trigger is followed by a local computation step which involves a corresponding update of the pending request table. After this local update the call to the operation op of the port-instance is completed. Note however, as described above, that the caller object, after completion of the call to the operation op of the port-instance will wait for a call to the operation return-op which will coincide with the completion of the operation op by the internal object.

The second loop starts with the selection of a call in the pending request table. The port-instance subsequently calls the corresponding operation of the callee. When this call is completed the port-instance resumes its activity by sending the received return value to the initial caller object.

5.5 Inter-components coordination

Reactive systems are systems that reacts continuously to their environment, at a speed imposed by the latter [HP85]. Among reactive systems are most of the industrial real-time systems, like control systems and signal processing systems. These systems are distributed in their own nature: think for example at the different location of the sensors and actuators of a system. Furthermore they are subject to temporal requirements concerning both the input rate and the response time. This requirements must be taken into account when modeling a system, for example, by considering an architectural design employing the *globally asynchronous locally synchronous* paradigm [Cha85]: communication within a unit of distribution may be synchronous, whereas communication between different unit of distribution must be asynchronous.

The semantic model of Ω-UML is rich enough to support communication through shared attributes, operation calls, and signals. Synchronous operation calls use rendez-vous as communication mechanism. It involves a synchronization between the sender and the receiver of the operation call for the message (and parameters) passing, an asynchronous establishment of the rendez-vouz, and another synchronization between the sender and receiver for passing the return values. Rendez-vous lead to useless waiting time and reduce parallelism and efficiency [Fox88]. That is why in this section we restrict the communication model of Ω-UML so to support the globally asynchronous locally synchronous paradigm: all inter-component communication are purely asynchronous (via signal events), while intra-component communication is unrestricted. This way, components are *units of distribution* of reactive real-time systems.

A consequence of the inter-component communication by signal events is that each port instance (the receivers of all the signals directed to a components) must be equipped with an event-queue, so that incoming requests are processed in a first-in-first-out fashion. This is equivalent to say that communication between components is performed by means of send and receive primitives over a network of FIFO channels: The processes in the network are the ports and the port instances. while the channels are the event-queue associated with each port instance. An asynchronous operation call $p.signal(parameters)$ correspond to sending the structured signal $signal(parameters)$ to the channel p (the identity of the port instance owning the event-queue), while the trigger of the signal correspond to the reception of the first signal from the channel with sink attached to the process. Notice that channels can be dynamically created, and passed to other processes by means of a signal. Therefore, communications are performed over a dynamically reconfigurable networks of channels and processes.

In other words, by loose coupling the inter-component communication mechanism (here obtained by forbidding synchronous operation call) one obtain a system of dynamic processes communicating through mobile channels. There is, however, an asymmetry in the above coordination mechanism: channels are mobile only at their source. This is due to the fact that in UML, the triggering of an operation is implicitly directed to the event queue of the active object controlling it. If we relax this constraint, and introduce trigger operations directed to a channel (introducing, for example, a syntax for trigger operation similar to a CSP [Hoa85] read operation c?signal(parameters)) then we obtain a the coordination model of mobile channels proposed in [ABdB00, SAdBB03]: Processes can be created dynamically and have an independent activity that proceeds in parallel with all the other processes in the system and interact only by sending and receiving messages *asynchronously* via channels which are (unbounded) FIFO buffers. Channels are created dynamically. In fact, the creation of a process consists of the creation of a channel which connects it with its creator. This channel has a unique identity which is initially known only to the created process and its creator. As with any channel, the identity of this initial channel too can be communicated to other processes via other channels, so that which processes are connected by which channels, is completely dynamic, without any regular structure imposed on it a priori.

A compositional formal semantics based on histories of signals sent and received by each process has been given for the above coordination model [dBB00], together with a logic-based component interface description language that conveys this observable semantics [AdBB00]. This interface description language allows for deriving properties of a component-based system out of the logical interfaces of each port of the constituent components [AdBB00]. Finally, the model has also been implemented as a middleware for distributed communication and collaboration [SAdBB02].

5.6 Conclusion

In this paper we have presented a UML model for components to address architecture and component based development. Components are units of abstraction that can be independently developed, like classes or modules. Unlike classes, they components are also unit of encapsulation that can be extended by subtyping of the interfaces, but not by inheritance of their implementation. Component-based systems are described by means of two new UML diagrams: component system diagrams and architectural diagrams. Component system diagrams are for describing the structural dependencies among the provided and required interfaces of the components in a system, while architectural diagrams are for the description of a runtime configurations of the architecture of a component-based system.

Our model offers a coherent view for the design of architecture and component-based systems: components serve as a naming mechanisms for abstracting from the internal parts, interfaces as declaration mechanisms of services (either provided or required) and ports together with the dependency-realization relations as abstraction mechanisms of object interactions.

Contrary to the component concept used in deployment diagrams of UML 1.4 [SP99], our components are not units of instantiation and do not need to have a unique run-time identity. Our model of component is similar to the recently approved proposal by the U2 partners for UML 2.0 [OMG03], but components have no state, are not instatiable, and allow for the existence at run time of multiple ports with the same set of interfaces, each Port attached to the necessary number of runtime links. For example, in UML 2.0 a class is also a component, while this is not the case for our notion of component.

Our model has been largely influenced by the main concepts offered by architecture description languages (ADLs): components, ports, and configurations. A large number of ADLs have been proposed, some of them with a sound formal foundation. We only mention here Wright [AG97], Rapide [LKA$^+$95] and ACME [GMW97]. Closer to our architectural diagrams are the architectural descriptions provided by ROOM [SGW94] and UML-RT [Sel98] (the latter is in fact a UML profile interpreting ROOM concepts in terms of UML stereotypes).

Many models for components have been proposed in the last years, some informal and remaining within the realm of the existing UML (see for example [CD00]), and others founded on a logical and mathematical basis (e.g. Broy's component model based on streams of messages [BS01]. In [GCK02] and [MRRR02], several strategies for modeling components and other architectural concepts within UML are investigated, with as conclusion that these concepts are hard to describe in UML as it is. Similar to Broy's component model, the semantics of our model is also based on sequences of messages (like those used for the semantics of CSP [Hoa85]). However our components have dynamic aspects (e.g. Port instances) not fully covered by Broy's model.

Moreover our component model is a conservative extension of an object-oriented model and therefore it requires the addition of only a couple of extra concepts to the standard UML 1.4. It is interesting to note that these additional concepts are also required by the component model proposed for UML 2.0 by the U2 partners [U20]. As described above, however, the semantics of these concepts is different between the two models.

Acknowledgement The work reported in this paper has been funded by the European IST-2001-33522 project OMEGA.