A Combined Approach for Component-Based Software Design

C. R. Guareis de Farias, M. van Sinderen, L. Ferreira Pires, D. Quartel

Telematics Systems and Services, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{farias, sinderen, pires, quartel}@cs.utwente.nl

ABSTRACT. Component-based software development enables the construction of software artefacts by assembling binary units of production, distribution and deployment, the so-called software components. Several approaches addressing component-based development have been proposed recently. Most of these approaches are based on the Unified Modeling Language (UML). UML has been increasingly used in component-based development, despite some shortcomings associated with this language. This paper presents a methodology for the design of component-based applications that combines a model-based approach with a UML-based approach. This combined approach tackles some of the limitations associated with UML, allowing a better control of the design process. Our combined approach is illustrated using some excerpts from a case study carried out on a chat application.

Keywords. Software components, Component-based design, UML, AMBER.

1 INTRODUCTION

Component-based software development has emerged to increase the reusability and portability of pieces of software. Component-based development aims at constructing software artefacts by assembling (software) components. We define a component as a binary piece of software, self-contained, customisable and composable, with well-defined interfaces and dependencies.

Traditional object-oriented software development aims at enabling the reuse of object type definitions (object classes) at design and implementation levels. In contrast, component-based development aims at enabling the reuse of components at deployment level. Components represent complete pieces of functionality that are ready to be installed and executed in multiple environments, provided that a middleware platform that supports the execution of the components is available.

Some design methodologies addressing component-based development have been proposed recently. Most of them are based on the Unified Modelling Language (UML), c.f. [1, 5, 6, 7, 8]. UML [10] is a process-independent modelling language widely accepted in both academic and industrial settings. UML basically consists of a collection of diagrams used to model a system under different and often complementary perspectives.

Although UML has been increasingly used as the basis for such approaches, it still has some drawbacks that hinder its usage and effectiveness. So far, the support provided by UML for component-based development is limited. Both the UML component semantics and notation should be improved [9, 11]. A major change in UML with this respect is expected to occur with the release of the UML 2.0 specification, which is expected by the end of 2001.

The specification of complex behaviours using UML behaviour diagrams can be cumbersome [4]. These types of diagram provide roughly three general kinds of constructs to describe the relationship between states or activities: enabling, interleaving (parallelism) and synchronisation. Other types of relationship that would improve the modelling capabilities of UML, such as non-deterministic choice and disabling, are not supported. Further, the specification of complex interaction patterns using sequence diagrams often leads to diagrams of poor legibility.

Finally, the use of UML to model the service provided by an application and to decompose this service into a set of components is usually informal and intuitive. Therefore, it is difficult to formally assess whether that the achieved decomposition in terms of components complies with the required service.

This paper presents a methodology for the development of component-based applications that combines a model-based approach [13] with a UML-based approach [5, 6]. This combined approach aims at profiting from the advantages of both approaches: the abstraction power and formality associated with the use of an abstract architectural modelling language, called AMBER, and the diversity of concepts and public acceptance associated with UML. To exemplify parts of our methodology we use some excerpts from a case study on a simple chat application.

This paper is further structured as follows: section 2 introduces AMBER; section 3 provides an overview of our combined approach, while sections 4 to 6 detail it with the help of a running example; section 7 discusses some related work; finally, section 8 presents some final remarks and outlines some future work.

2 THE AMBER MODELLING LANGUAGE

This section introduces the formal modelling language that forms the basis of our combined approach to component design. This language is called Architectural Modelling Box for Enterprise Redesign (AMBER) [2, 12].

An AMBER model of a system consists of two separate sub-models, viz., an entity model and a behaviour model.

Entity model

An entity model represents relevant system parts at a given abstraction level and their interconnection. Two concepts are used in an entity model, viz., entity and interaction point.

An entity represents a system (part) that carries out some function or behaviour. An entity may be decomposed into smaller units, called sub-entities. An interaction point represents some mechanism, physical or logical, through which an entity can interact with other entities or with its environment. Consequently, an interaction point is shared by two or more entities or by one or more entities and their environment.

Figure 1 shows the graphical notation of the entity model concepts. An entity is represented by a rectangle with cut-off corners, while an interaction point is represented by an ellipsis that overlaps with the entities that share the interaction point or by separated ellipses interconnected by a line. Figure 1(a) depicts an entity E1 with a single interaction point. Figure 1(b) depicts the decomposition of the entity *E1* into the sub-entities *E2* and *E3*, all of them sharing the same interaction point.

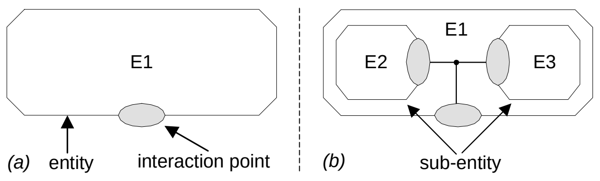

Figure 1. Entity model notation.

Behaviour model

A behaviour model represents the functionality or behaviour of each entity described in the corresponding entity model. Three basic concepts are used in a behaviour model, viz., action, interaction and causality relation.

An action represents an activity performed by a single entity, while an interaction represents a common activity performed by two or more entities. For simplicity reasons, the term action is used to refer both to actions and to interactions in the remaining of this section.

An action abstracts from how the result of the activity being modelled is established. However, the result established by an activity can be represented by attaching attributes to the corresponding action. Attributes of information, time and location represent values of information established in the activity, the time moment at which the activity is completed and the logical or physical location where the activity takes place, respectively.

The occurrence of an action represents the successful completion of an activity. In case an action occurs, the same result is established and made available at the same time moment and at the same location for all entities involved in the activity, otherwise no result is established.

Figure 2 depicts our graphical notation of an action and an interaction. An action (Figure 2a) is graphically represented as a circle (or ellipsis), while an interaction (Figure 2b) is graphically represented as a segmented circle (or ellipsis), one segment for each interaction contribution.

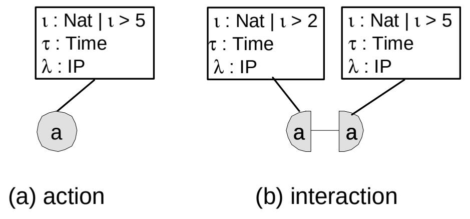

Figure 2. Action and interaction.

The information (ι), time (τ) and location (λ) attributes are represented within a textbox attached to the action. Constraints can be defined on the possible outcomes of the values of ι, τ and λ (after the symbol '|'). In case of an interaction, each involved entity can define its constraints, such that the values of ι, τ and λ must satisfy all constraints, otherwise the interaction can not happen. In case multiple values are possible for some attribute, a non-deterministic choice between these values is assumed.

A causality relation is associated with each action, modelling the conditions for this action to happen in terms of the occurrence or non-occurrence of other actions. An action only occurs when its enabling condition is satisfied.

Two basic kinds of causality relation between two actions, a and b, are the enabling relation, in which the occurrence of a enables the occurrence of b, and the disabling relation, in which the occurrence of a disables the occurrence of b, provided that b has not occurred yet. Furthermore, in case of the absence of any causality relation between two actions, these actions are independent (concurrent).

Basic causality relations can be composed using booleanlike operators. They can also be supplied with additional constraints, which further restrict the relation with conditions on attribute values of preceding actions. A probability attribute can also be added to each causality condition to model the probability the action happens when the enabling condition is satisfied.

Figure 3 shows some common action relations between two or more actions. A trigger represents a special kind of action which has its enabling condition is always satisfied.

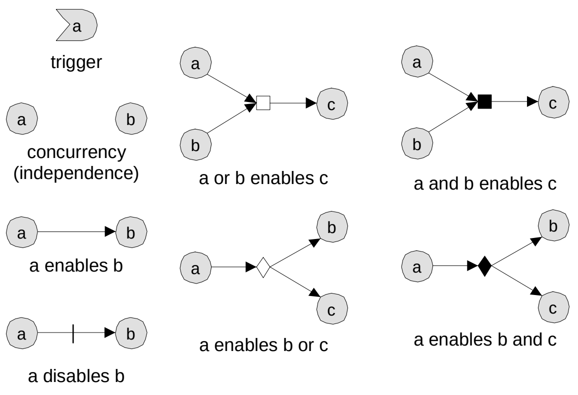

Figure 3. Common action relations.

AMBER provides a construct, called behaviour block, which allows one to better structure behaviour. A behaviour block is graphically represented as a rectangle with round corners. Similarly to entities, blocks can be decomposed into smaller units, called sub-blocks.

When actions connected through a causality relation are placed in separate behaviour blocks, an exit and an entry points are added at the block's edge (depending on the direction of the causality relation) to indicate that a condition in a behaviour block enables an action in the other behaviour block.

Blocks can also be used to represent repeated and replicated behaviours. A repeated behaviour indicates the occurrence of a similar behaviour over time, while a replicated behaviour indicates that a number of similar behaviours are executed in parallel.

We exempt ourselves from discussing AMBER further in this paper. Additional notational details to understand our approach are provided throughout the paper as needed.

Tool support

AMBER is supported by an integrated toolkit called Testbed Studio [2]. The support provided by Testbed Studio includes, a.o., support to visual representation of the models, analysis techniques (quantitative and functional), step-by-step simulation of the models and documentation.

3 PROCESS OVERVIEW

Our approach identifies four abstraction levels for the development of an application, viz., enterprise, system, component and object.

The enterprise level aims at capturing the vocabulary and other domain information of the system being developed. This level provides the most abstract description of the system being produced.

The system level delimits the system being developed from its environment. The environment of a system consists of any information system or human users that make use of the services provided by the system itself as well as other systems that provide some service used by the system being developed.

The component level represents the system in terms of a set of composed components. A component may be further decomposed in sub-components. A composite component is an aggregate of sub-components that, from an external point of view, is similar to a single component. If a composite component is part of the component composition, the design process of this component corresponds to the design process of an isolated system, in which the other components are part of the environment of this system.

The object level defines the internal structure of (simple) components. A component is structured using a set of related objects, which are implemented in a programming language. The development process of a component at the object level corresponds to traditional object-oriented software development processes and therefore we refrain from discussing it in detail in this paper.

Figure 4 depicts the layering structure of our approach.

The four abstraction levels are related to each other in different ways. For example, the system level corresponds to one possible mapping of the information captured at the enterprise level. Different systems can be generated based on the same information. The component level corresponds to a refinement of the system level, in which the system or a composite component is refined into a set of components. The object level corresponds to a refinement of the component level, in which each (simple) component is refined into a set of objects.

We can also abstract from a set of objects to form a component and abstract from a set of components to form the system or a composite component. However, it is not always possible to abstract from the system to obtain the complete description of the enterprise level because the information present at the system level may correspond only to a subset of the information at the enterprise level.

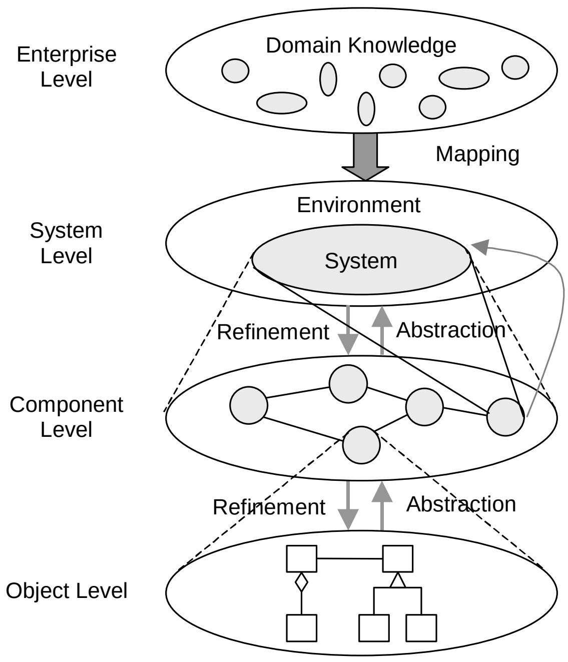

Figure 4. Development using abstraction levels.

Besides structuring into abstraction levels, we also consider different views at each one of these levels. Each view offers a different perspective of the system being developed. These perspectives are interrelated so that the information contained in one view can partially overlap the information contained in the others.

We identify three basic views, viz., structural, behavioural and interactional. The structural view provides information about the relations between concepts and/or the structure of entities and their interconnection. The behavioural view provides information about the behaviour of each entity in isolation, while the interactional view provides information about the cooperative behaviour of the entities as they interact with each other. Both the behavioural and the interactional views can be seen as dual views on the same aspect, viz., behaviour.

4 ENTERPRISE LEVEL

The enterprise level captures the vocabulary and other domain knowledge information related to the system being developed. The information is used both to communicate with the future or intended users of the system and to serve as the basis for delimiting the system with respect to its environment.

Different sets of concepts may be captured at this level according to the domain involved. For example, common concepts usually captured at the enterprise level are actors, activities, goals, processes, information, support services, etc. However, specific domains may require specific concepts, such as rules, policies and events.

The structural view at the enterprise level is captured using concept diagrams. A concept diagram consists of a UML class diagram in which an object class represents a concept and an association between classes represents a relationship between these concepts. Often there is no direct mapping between the identified concepts and their possible implementations.

A glossary of terms is developed in parallel with the development of the concept diagrams. The use of such a glossary aims at creating and maintaining a standard documentation of the concepts encountered in the domain of the system. The glossary should be maintained and updated as the development of the system continues.

While concept diagrams are useful to capture the structural relationship between concepts, these diagrams are not so convenient to capture behaviour. Therefore, the behavioural and structural views at the enterprise level are captured using AMBER models.

AMBER can be used in two different ways: to capture simple relationships between the identified activities and to capture possible sequences of activities. The identified activities are modelled as actions in AMBER, while any piece of information used by an activity is modelled as the information attribute of the corresponding actions.

Figure 5 depicts some AMBER models of the chat application at the enterprise level. Figure 5a shows, on the left, a model capturing the relationship between the actions register and connect (choice, i.e., the execution of one action disables the other and vice-versa). Figure 5a shows, on the right, that a choice is made after the occurrence of the action join among the actions send, receive, invite, answ-inv and leave. Figure 5b shows two possible sequences of actions.

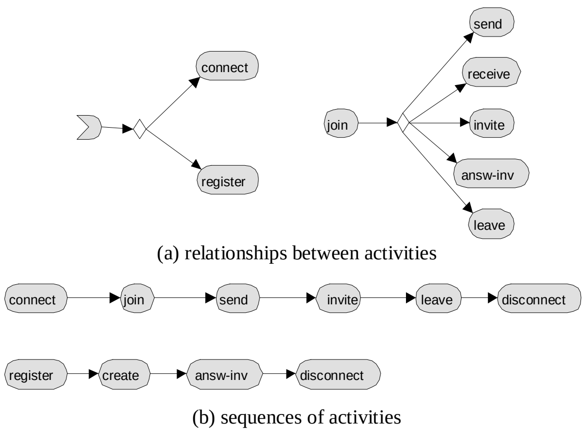

Figure 5. AMBER model of the enterprise level.

When expressing some possible sequences of actions in

AMBER, one should not be concerned with the amount of sequences provided, but with the added value concerning the understanding of the whole picture provided by each new sequence.

5 SYSTEM LEVEL

At the system level we describe the service provided by the application being developed. At this level we obtain a clear definition of the boundary between the system and its environment. External supporting services are identified at this level as well. These services are considered to be part of the system environment.

The structural view at the system level is captured using an AMBER entity model and a UML use case diagram. An entity model is used to capture the static relationship between the system and external supporting services, while a use case diagram is used to organise the user functional requirements.

To create an entity model at the system level, we map the actors identified at the enterprise level onto entities. The environment of the system being developed is mapped onto entities as well. Interaction points are defined between the identified entities.

These entities are then mapped onto actors in a use case diagram, while the activities identified at the enterprise level are mapped onto use cases. Each activity can be mapped to a separate use case or two or more related activities can be combined in a same use case. Although the description of a use case correspond to some behaviour, in the structural level we are concerned with how these pieces of behaviour relates to each other and with an associated actor. Later, the behaviour described by each use case forms the basis for capturing the behavioural and interactional views.

Figure 6 shows the entity model of the chat application at the system level. Two entities were identified: *Chat Application*, representing the chat application itself, and *Participant*, representing the environment of the chat application. A single interaction point, *ip1*, between the two entities is defined.

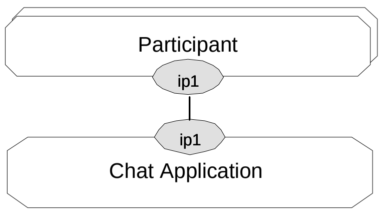

Figure 6. Entity model of the system level.

Both the behavioural and the interactional views are initially captured using the behaviour model from AMBER.

The first step towards creating a behaviour model at the system level is to represent the combined behaviour of the identified entities as a whole. In this step we abstract from the individual responsibilities of the entities while interacting by only considering a set of integrated interactions (modelled as actions) and the causality relations between them. This combined behaviour offers the most abstract representation of the service provided by the system and it is called integrated perspective.

Figure 7 shows the integrated behaviour model of the chat application at the system level. In this phase the actions and causality relations identified at the enterprise level are preserved and further combined.

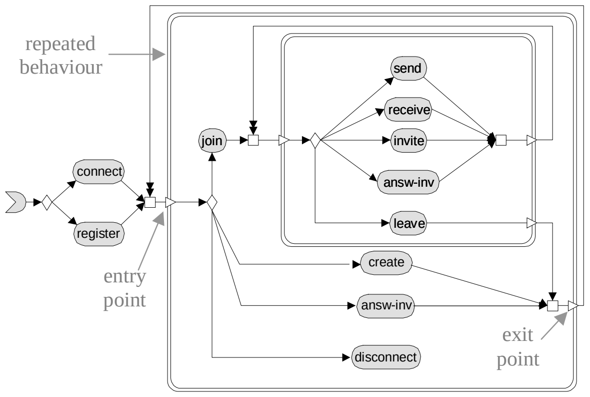

Figure 7. Integrated behaviour model at system level.

After describing the integrated perspective, we now consider the individual responsibilities of each one of the entities involved. At this point, we describe, for each entity, its interaction contributions and the causality relations between them. Such a description is called distributed perspective.

Thus, at the distributed perspective each action present at the integrated perspective is refined into an interaction and each causality relation is distributed over the involved entities. While describing the distributed perspective, we refrain from capturing unnecessary details on how to use the system or its internal structure. The internal behaviour of entities representing the system environment is of no further concern either.

The use of AMBER to model the behaviour of the application being developed at the system level (integrated and distributed perspectives) does not provide a clear separation between the behavioural and interactional views. In a single diagram we capture the behaviour of each entity in isolation plus the interactions between the entities. This can pose an extra burden, especially when multiple entities are involved in relatively complex behaviours.

Therefore, we suggest the use of UML diagrams to complement both the behavioural and interactional views. To complement the behavioural view we suggest the use of activity diagrams, while to complement the behavioural view we suggest the use of (non-standard) package sequence or collaboration diagrams. Package sequence and collaboration diagrams are not explicitly present in the UML notation guide nor supported by most UML tools, but they are allowed according to the UML metamodel [7].

Activity diagrams can be used to represent how the interactions relate to each other in the scope of an entity, while a package sequence/collaboration diagram can be used to represent the relationship between interactions in general.

To help capturing the interactional view we make use of some user-supplied usage scenarios. Each scenario describes different situations in which the application is used and usually involves the execution of several use cases.

6 COMPONENT LEVEL

The component level represents the system being developed in terms of a set of interconnected components. A component provides access to its services via one or more interfaces. These services usually can be customised by setting up some properties of the component.

When building a cooperative system from components, we do not need to know how these components are internally represented as objects. Actually, a component does not have to be necessarily implemented using an object-oriented technology, although this technology is generally recognised as the most convenient way to implement a component.

Components can be off-the-shelf, adapted from similar components or constructed from scratch. So far, most of the effort spent on building component-based applications concentrates on building new components. However, the more mature and widespread this technology becomes the more likely it is that this effort will move towards adapting similar components and reusing existing ones [3].

The structural view of an application at the component level is captured using AMBER entity model. This entity model corresponds to a refinement of the entity model captured at the system level, in which entities are refined into sub-entities, representing components, and interaction points are added to connect these entities. The entity model at this level is used to represent the static relationship between the identified components themselves and between the component and the application environment.

The UML use case diagram identified at the system level may also be refined and split among the identified components, such that these components correctly support the use cases. However, there is no rule of thumb on how to split and assign use cases to components. A good practice is to keep similar functionality in a single component and assign distinct functionality to separate components. Although similarity and distinction are subjective terms, sometimes it suffices to rely on the individual judgement and experience of the application designer. In case a use case is likely to be supported by two or more components, it is possible that this use case is too complex and that it should be refined in multiple simpler use cases.

The behaviour modelling (behavioural and interactional views) of the application at the component level follows an approach similar to the system level. Initially, both the behavioural and the interactional views are captured using AMBER behaviour model.

For each identified entity at the structural view, a behaviour is assigned. However, we are now interested in revealing not only the external (observable) behaviour but also the internal details of the system, i.e., how the interactions between components are refined and how actions are inserted to represent the activities performed by the components that form the system.

Similarly to the system level, the use of AMBER to model the behaviour of the application at the component level does not provide a clear separation between the behavioural and interactional views. Therefore we also use UML diagrams to complement the information captured by these views at the component level.

To complement the behavioural view we suggest the use of activity diagrams to represent how the interactions and actions of each component relate to each other in the scope of an entity. To complement the behavioural view we suggest the use of package sequence or collaboration diagrams to represent the relationship of the interactions between the identified components.

If composite components are involved, it may be necessary to produce several refinements of the components, each producing a detailed description of the components involved and their relationship. For example, in the design of the chat application at the component level we initially identified two composite components, viz., a client component and a server component.

Figure 8 shows parts of the behaviour model of the server component. Two sub-behaviours are depicted: on the left a sub-behaviour representing the connection management of s participant and on the right a sub-behaviour representing the management of invitations. Figure 8 shows internal details (activities modelled as actions in behaviour blocks) and how the interactions are refined. While at the system level the answer to an invitation is represented as a single interaction, at the component level this interaction is split in two separate interactions, one representing the notification and another representing the answer itself.

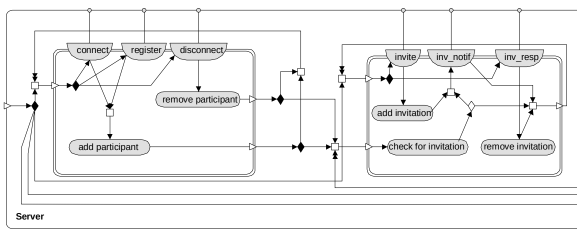

Figure 8. Server component behaviour model.

In the behaviour model of the client and server components, the identified sub-behaviours supply some clues on how the structure of the next component level can be defined. Each sub-behaviour is assigned to a different component (similar sub-behaviours may also be assigned to a single component). At each refinement of the component level the internal details are further revealed and the interactions are further refined. A detailed account of behaviour refinement is presented in [12,13].

Component modelling

In order to model the behaviour of a component using AMBER, including its interface, we need to establish some conventions, which are not part of AMBER.

A component may have one or more interfaces through which its services become available. However, each interface should be contained in a separate behaviour block. An interface contains operations. The execution of an operation is modelled as interactions. We distinguish between two different types of operations, viz., an invocation, which returns a value to the invoking component, and an announcement, which does not.

An invocation is modelled as a sequence of two interactions, viz., an invocation request and an invocation return. Optionally, you may have a third interaction representing the occurrence of an exception during the invocation process. An announcement is modelled as a single interaction between two components. A second (optional) interactional may represent the occurrence of an exception during the announcement process. In both cases, the occurrence of an exception indicates that the operation could not be completed successfully.

Figure 9 depicts the different types of operations between components C1 and C2. Figure 9a represents an invocation operation, while Figure 9b represents an announcement operation. The interaction modelling an exception in Figure 9b, although enabled, may not actually happen. This is represented by annotating the enabling relation with a question mark.

The following convention is used to model an invocation operation: the keywords invocation, return and exception should be added to the name of the operation in the interaction identifiers. These keywords represent the invocation, return and exception interactions, respectively. To model an announcement operation the keyword announcement should be added to the name of the interaction.

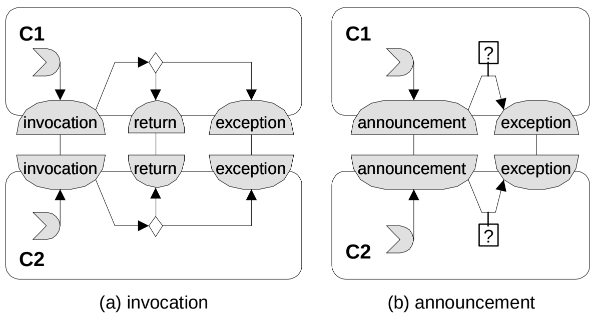

Figure 9. Interface operations.

Input parameters are modelled as structured information attributes of the invocation interaction, while output parameters are modelled as structured information attributes of the return interaction. To model the return value of the operation itself, we attach the keyword returnvalue to an attribute in the return interaction.

To model the occurrence and notification of events we also have a special notation, which is similar to an announcement operation. The keywords event announcement or event notification are added to indicate that the interaction corresponds to an event. The keyword event announcement indicates that a component is producing an event, while the keyword event notification indicates that a component is consuming this event.

The subscription to an event by a consumer component can be modelled as an invocation operation. This component passes in the invocation interaction a reference to one of its interfaces at which the occurrence of the event should be notified. For simplification purposes, we abstract from the subscription process and consider only the announcement and notification of the event itself.

In Figure 9 we also abstract from the platform supporting the communication between both components. ORB mediated communication modelling is discussed in [13].

7 RELATED WORK

The Unified Process [8] is a software development process based on UML. However, the Unified Process is not simple, on the contrary, it is very complicated. Actually, the Unified process is not really a development process but more like a process framework, since it describes best practices in software development but still has to be specialised to be suitable for different projects. Therefore, it lacks systemacticness and prescriptiveness. Nevertheless the Unified Process is very flexible and scalable, having being largely used in the software industry.

Another major problem faced by the Unified Process is its limited view of components. The Unified Process is not really a component-based software development process, but rather the use of components in this methodology is an afterthought, since it prescribes the development of a set of objects followed by their grouping into components.

Catalysis [1] is another complex software development process based on UML. Similarly to the Unified Process, Catalysis is much like a process template, which can be tailored according to the situation. Thus, Catalysis also lacks prescriptiveness. Nevertheless, Catalysis is flexible and scalable, being also very popular among software developers.

A major benefit from Catalysis relies on its explicit use of components. However, being a broad software development process, Catalysis is not completely component-oriented.

8 FINAL REMARKS

This paper presented a combined approach for the design of component-based applications. According to this approach, the development of an application is organised using four different abstraction levels. At each level, different views are used to capture structural, behavioural and interactional aspects of the application under development. These views seem to be the most relevant ones for application design. Still, we can have other views if necessary, such as test, functional or deployment views.

Our development process is not so generic and complete as Unified Process and Catalysis, however, our methodology is simpler. Further, our combined methodology does not rely exclusively on UML and therefore is not subjected to the shortcomings of this language if compared to other methodologies.

The use of the formal language AMBER to complement UML diagrams provides two major advantages. First, the concepts present in AMBER are simple, intuitive and close to the concerns of an application designer at the early stages of the development process. AMBER offers a high abstraction power and different constructs to represent behaviour. Second, the use of a formal modelling language enables a number of extra design activities, such as analysis, verification and simulation, to be carried out in parallel with the design process itself. These activities enhance the quality of the design and allow the detection and solution of problems as early as possible.

References

1. D'Souza, D. F. and Wills, A. C.: *Objects, Components and Frameworks with UML: the Catalysis Approach*. Addison Wesley, USA, 1999.
2. Eertink, H., Janssen, W., Oude Luttighuis, P., Teeuw, W. and Vissers, C.A.: A Business Process Design Language. In *1999 World Congress on Formal Methods (FM'99), Vol. I., Lecture notes in Computer Science (1708)*, Springer, pp. 76-95, 1999.
3. Grasso, M.P.: Distributed component systems: the next new computing model. *Application Development Trends*, 6(11), pp. 43-52, 1999.
4. Guareis de Farias, C.R., Diakov, N. and Poortinga, R.: Analysis of UML. *Amidst technical report, AMIDST/WP1/N006/V04*, 1999.
5. Guareis de Farias, C. R., van Sinderen, M., and Ferreira Pires, L.: A systematic approach for component-based software development. In *Proceedings of the Seventh European Concurrent Engineering Conference (ECEC 2000)*, pp. 127-131, 2000.
6. Guareis de Farias, C.R., Ferreira Pires, L. and van Sinderen, M.: A component-based groupware development methodology. In *Proceeding of the 4th International Enterprise Distributed Object Computing Conference (EDOC 2000)*, pp. 204-213, 2000.
7. Hruby, P.: Structuring Design Deliverables with UML. In *Proceedings of UML'98 International Workshop*, pp. 251-260, 1998.
8. Jacobson, I., Booch, G. and Rumbaugh, J. *The unified software development process.* Addison Wesley, USA, 1999.
9. Kobryn, C.: UML 2001: a standardization odyssey. *Communications of the ACM*, 42(10), 29-37, 1999.
10. Object Management Group: *Unified Modeling Language 1.3 specification*, 1999.
11. Object Management Group UML Revision Task Force: *OMG UML v. 1.3: Revisions and Recommendations*, 1999. Available at http://www.omg.org.
12. Quartel, D.: *Action relations: basic design concepts for behaviour modelling and refinement.* PhD thesis, University of Twente, Enschede, Netherlands, 1998.
13. Quartel, D.A.C., van Sinderen, M.J., and Ferreira Pires, L.: A model-based approach to service creation. In *Proceedings of the 7th International Workshop of Future Trends in Distributed Computing (FTDCS'99)*, pp. 102-110, 1999.