A Combined Component-Based Approach for the Design of Distributed Software Systems

C. R. Guareis de Farias*, L. Ferreira Pires, M. van Sinderen, 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 components. Several approaches to 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 of 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 of UML, allowing a better control of the design process.

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. In this scope, a component is a self-contained, customisable and composable binary piece of software, with well-defined interfaces and dependencies.

A component is a unit of deployment and distribution. 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. Special interest has been recently given to the (runtime) reconfiguration and migration of components in component-based systems, c.f., [1, 10].

Some design methodologies that address component-based development have been proposed recently. Most of them are based on the Unified Modelling Language (UML) [11], c.f. [2, 6, 7, 8]. Although UML has been increasingly used as the basis for such development methodologies, 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, 12]. A major change in UML with this respect is expected to occur soon with the release of the UML 2.0 specification.

The specification of complex behaviours using UML behaviour diagrams can be cumbersome [5]. These types of diagram provide roughly three kinds of constructs to describe the relationships 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 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 [14] with a UML-based approach [6]. This combined approach aims at profiting from the benefits of both approaches: the abstraction power and formality associated with the use of the abstract architectural modelling language AMBER [3, 13], and the diversity of concepts and public acceptance of UML.

This paper is structured as follows: section 2 introduces AMBER; section 3 introduces the main elements of our combined approach, while sections 4 to 6 describe our approach in more detail; section 7 discusses some related work and presents some final remarks.

* Supported by CNPq (Brazil).

Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'01)
1071-0485/01 $17.00 © 2001 IEEE

2. The AMBER modelling language

AMBER [3, 13] stands for Architectural Modelling Box for Enterprise Redesign. An AMBER model of a system consists of two separate sub-models: an entity model and a behaviour 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 that carries out some function or behaviour. An entity may be decomposed into sub-entities. An interaction point represents some mechanism, physical or logical, through which an entity can interact with its environment (including other entities).

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.

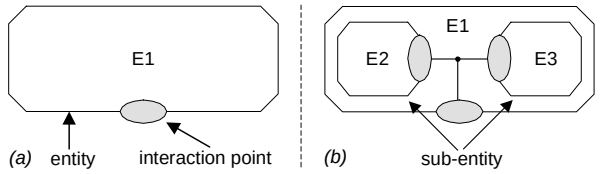


Figure 1: Entity model notation

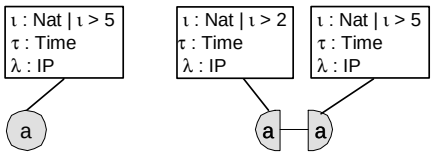
A behaviour model represents the functionality or behaviour of each entity described in the corresponding entity model. Three basic concepts are used in a behaviour model, viz., action, interaction and causality relation.

An action represents an activity performed by a single entity, while an interaction represents a common activity performed by two or more entities. The term action is used to refer to both actions and interactions, whenever desirable for conciseness.

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.



(a) action (b) interaction

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 cannot happen.

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.

The 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. Further, in case of absence of a causality relation between two actions, these actions are independent (concurrent).

Basic causality relations can be composed using boolean operators. Causality relations can also contain constraints that restrict the occurrence of actions based on the attribute values of preceding actions. A probability attribute can also be added to each causality condition to model the probability that the action happens in case 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 always satisfied.

AMBER behaviour blocks (see Figure 7) allow one to structure behaviours. A behaviour block is graphically represented as a rectangle with round corners. Similarly to entities, blocks can be decomposed into 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 block enables an action in the other block.

Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'01)
1071-0485/01 $17.00 © 2001 IEEE

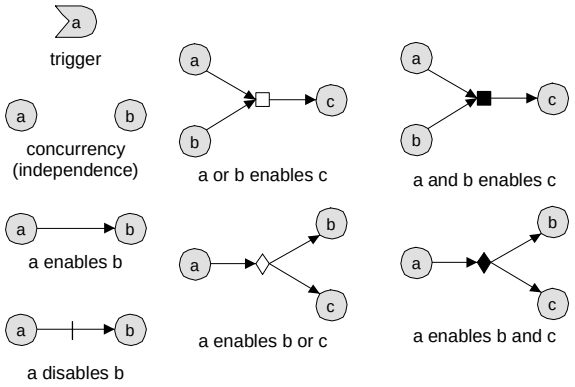


Figure 3: Common action relations

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. Notational details necessary to understand our approach are provided throughout the paper as needed.

3. Process overview

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

The enterprise level aims at providing a unified view of the system and its environment by capturing enterprise-related concepts.

The system level delimits the system being developed, distinguishing it from its environment. The environment of a system consists of information systems 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 a component composition, the design process of this component corresponds to the design process of an isolated system, and the environment of this system contain the other components in the composition.

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 further in this paper.

Figure 4 depicts the levels identified in our approach.

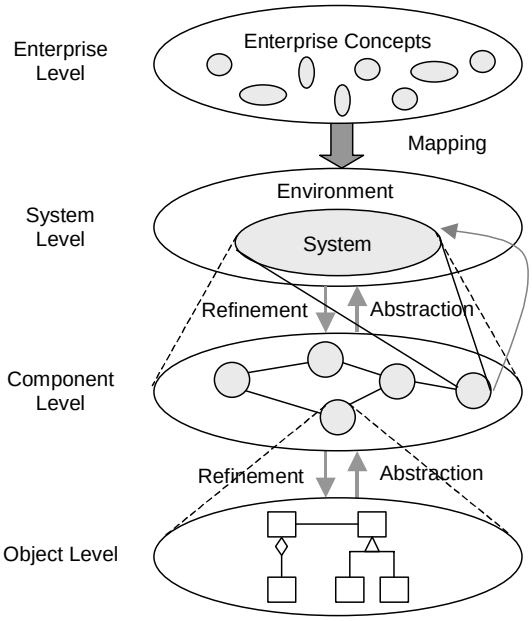


Figure 4: Development using abstraction levels

Besides structuring using 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 structure and static relations between entities. 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 aims at providing a conceptual and integrated description from a system and its environment. The description is conceptual because it models concepts of an application domain and integrated because no formal separation is made between the system and its environment.

Different sets of concepts may be captured at this level according to the target application domain. For example, common concepts usually captured at the enterprise level are actors, activities, goals, processes, informa-

Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'01)
1071-0485/01 $17.00 © 2001 IEEE

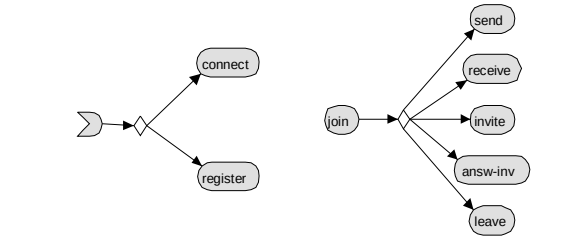
tion (resources), etc. However, specific domains may require specific concepts, such as rules, policies and events.

The structural view at the enterprise level is captured using concept diagrams. A concept diagram consists of a UML class diagram in which a class represents a concept and an association between classes represents a relationship between these concepts. A glossary is developed in parallel to the concept diagrams to document the concepts encountered. 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 convenient to capture behaviour. Therefore, we have decided to use AMBER models to capture the behavioural and structural views at the enterprise level.

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 an example application (chat application) at the enterprise level. Figure 5a shows that the execution of action register disables the execution of action connect and vice-versa (choice), and that a choice can be 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.



(a) relationships between activities

(b) sequences of activities

Figure 5: AMBER model of the enterprise level

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 system 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, allowing entities to interact.

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 corresponds to some behaviour, at the structural level we are concerned with how these pieces of behaviour relate 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 are identified: *Chat Application*, representing the chat application itself, and *Participant*, representing the user (environment) of the chat application. A single interaction point, *ip1*, represents the interaction mechanisms between the two entities.

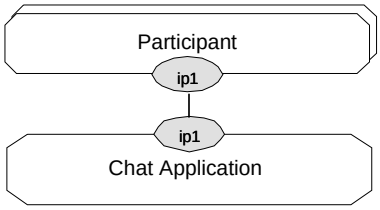


Figure 6: Entity model of the system level

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

To create a behaviour model at the system level we first 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.

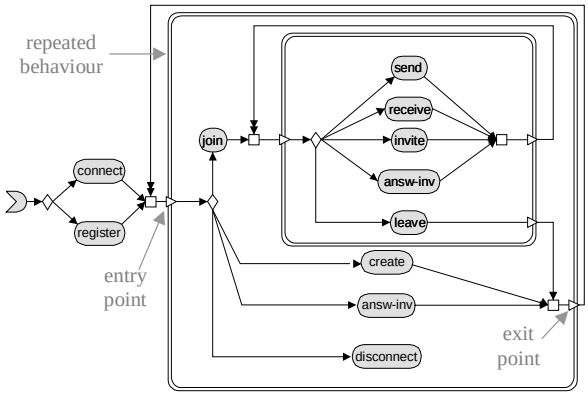


Figure 7: Integrated behaviour model

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

According to 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 concern at this level 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 defined in the UML notation guide nor are they 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 or collaboration diagram can be used to represent the relationship between interactions.

To help capturing the interactional view we use some user-supplied usage scenarios to describe the different situations in which the application can be used.

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.

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 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 this effort will move towards adapting similar components and reusing existing ones [4].

The structural view of an application at the component level is captured using an 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 that represent components, and interaction points are added to connect these entities. The entity model 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 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 models.

A behaviour is assigned to each entity identified at the structural view. 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 have 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: a sub-behaviour representing the connection management of a participant (left side) and a sub-behaviour representing the management of invitations (right side). Figure 8 shows internal details (activities modelled as actions in behaviour blocks) and how the interactions are refined.

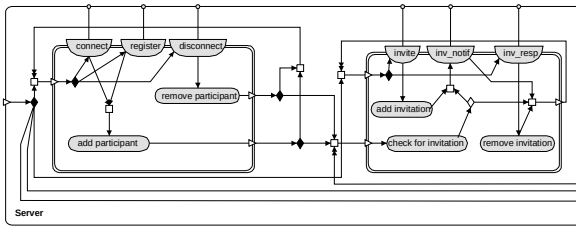


Figure 8: Server component behaviour model

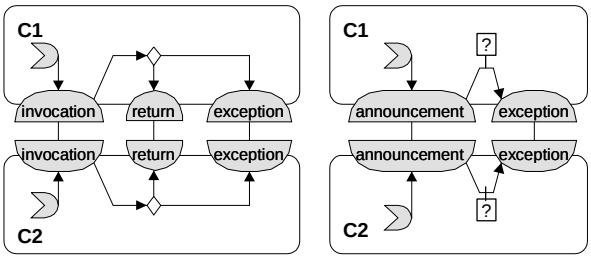
In the behaviour model of the client and server components, the identified sub-behaviours supplied some clues on how the structure of the next component level can be defined. Each sub-behaviour should be assigned to a different component, though similar sub-behaviours may also be assigned to a single component. At each refinement of the component level the internal details are further revealed and the interactions are further refined. A detailed discussion on behaviour refinement and refinement rules is presented in [13, 14].

In order to model the behaviour of a component using AMBER, including its interface, we need to introduce some additional conventions.

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 an interaction. 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 return.

An invocation is modelled as a sequence of two interactions, viz., an invocation request and an invocation return. An announcement is modelled as a single interaction between two components. Optionally, you may have in both cases an additional interaction representing the occurrence of an exception during the invocation or announcement. 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.



(a) invocation (b) announcement

Figure 9: Interface operations

To model an invocation operation, we add the keywords invocation, return and exception 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, we add the keyword announcement is the name of the interaction.

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 define some additional notation. 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 produces an event, while the keyword event notification indicates that a component consumes an event.

The subscription to an event by a consumer component can be modelled as an invocation operation. This

Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'01)
1071-0485/01 $17.00 © 2001 IEEE

component passes in the invocation interaction a reference to the interface 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.

7. Final remarks

This paper presented an approach for the design of component-based systems. According to this approach, the development of a system is structured in four different abstraction levels. At each level, different views are used to capture structural, behavioural and interactional aspects of the application under development; these seem to be the most relevant views for application design. Still, we can have other views if necessary.

Existing commercial UML-based methodologies are not completely component-oriented, which creates some barriers to the use and dissemination of component technology.

The Unified Process [8] is a rather complex UML-based software development process. The Unified Process is not really a development process but rather 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 some prescriptiveness. Nevertheless the Unified Process is flexible and scalable, having being largely used in the software industry. Further, the Unified Process is not really a component-based process. The use of components is an afterthought, since the Unified Process prescribes that the development of a set of objects could be followed by their grouping into components.

Catalysis [2] 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 a particular development project. Catalysis also lacks prescriptiveness, but it is flexible and scalable, being also popular among software developers. A major benefit of Catalysis is its explicit use of components. However, being a broad software development process, Catalysis is not completely component-oriented.

Our methodology is not as generic and complete as the Unified Process or Catalysis; however, it is simpler and component-oriented. Further, it does not rely exclusively on UML and thus is not subjected to its shortcomings.

The use of the formal language AMBER to complement UML diagrams provides two major benefits. 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, AMBER has a formal semantics that allows 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 in the development process.

8. References

1. Almeida, J.P.A.: *Dynamic Reconfiguration of Object-Middleware-based Distributed Systems*. M.Sc. Thesis, University of Twente, 2001.
2. D'Souza, D. F. and Wills, A. C.: *Objects, Components and Frameworks with UML: the Catalysis Approach*. Addison Wesley, USA, 1999.
3. 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., LNCS 1708*, pp. 76-95, 1999.
4. Grasso, M.P.: Distributed component systems: the next new computing model. *Application Development Trends*, 6(11), pp. 43-52, 1999.
5. de Farias, C.R.G., Diakov, N. and Poortinga, R.: Analysis of UML. *Amidst TR, AMIDST/WP1/N006/V04*, 1999.
6. de Farias, C.R.G., Ferreira Pires, L. and van Sinderen, M.: A component-based groupware development methodology. In *Proceedings of the 4th Int. Enterprise Distributed Object Computing Conference (EDOC'00)*, pp. 204-213, 2000.
7. Hruby, P.: Structuring Design Deliverables with UML. *In Proceedings of UML'98 Int. 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. Litiu, R. and Prakash, A.: Developing adaptive groupware applications using a mobile component framework. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW'00)*, pp. 107-116, 2000.
11. Object Management Group: Unified Modeling Language 1.3 specification, 1999.
12. Object Management Group UML Revision Task Force: *OMG UML v. 1.3: Revisions and Recommendations*, 1999.
13. Quartel, D.: *Action relations: basic design concepts for behaviour modelling and refinement*. PhD thesis, University of Twente, Enschede, Netherlands, 1998.
14. 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.