Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models

Tom Verdickt, Bart Dhoedt, Frank Gielen, and Piet Demeester, *Senior Member*, *IEEE*

Abstract—Distributed systems often use a form of communication middleware to cope with different forms of heterogeneity, including geographical spreading of the components, different programming languages and platform architectures, etc. The middleware will, of course, impact the architecture and the performance of the system. This paper presents a model transformation framework to automatically include the architectural impact and the overhead incurred by using a middleware layer between several system components. Using this framework, architects can model the system in a middleware-independent fashion. Accurate, middleware-aware models can then be obtained automatically using a middleware model repository. The actual transformation algorithm will be presented in more detail. The resulting models can be used to obtain performance models of the system. From those performance models, early indications of the system performance can be extracted.

Index Terms—Distributed software engineering tools and techniques, performance of systems: modeling techniques.

1 Introduction

One of the most critical aspects of the quality of a software system is its performance. At the same time, software engineering methodologies strongly focus on the functionality of the system, while applying a "fix-it-later" approach to software performance aspects. The system is designed to meet its functional requirements, postponing considerations about the nonfunctional requirements (such as performance) to the later development stages. As a result, lengthy fine-tunings, expensive extra hardware, or even redesigns are necessary for the system to meet the performance requirements. And, even with fine-tuning, there is no guarantee that the system performance will be appropriate.

1.1 Software Performance Engineering (SPE)

To solve this problem, software engineering techniques have been designed to integrate performance considerations into the design process. Performance modeling methodologies and quantitative solution methods are used throughout the entire development cycle (starting as early as possible) to check whether the system performance is satisfactory [1]. This allows the performance requirements to be "built into" the system, rather than added on later.

Several modeling formalisms have been designed to allow system designers to model the system performance, e.g., queueing networks [2], [3] and Petri Nets [4]. Several automated tools exist for most of these modeling approaches to obtain performance metrics from the models, either by using analytic techniques or by simulation (LQNS [5], SPNP [6], etc.). Using these modeling formalisms and tools, the system designers can obtain performance estimates at an early development stage and detect performance problems when solving them is still fairly inexpensive.

This methodology for performance engineering has an important drawback: It demands extra effort and capabilities from the system designers. New models need to be created in a performance modeling language unfamiliar to the designers. Much recent research is aimed at automating the performance modeling process, facilitating its adoption for system design. Part of the automation effort is the research of algorithms for the transformation of general-purpose system models (such as UML) into performance models (e.g., queueing networks) [7], [8]. This allows designers to model the system using the formalisms they are familiar with (e.g., UML) and obtain the performance models automatically.

Automatic transformation to obtain performance models requires the ability to specify performance parameters in the general-purpose system models. Therefore, some modeling formalisms have been extended with performance modeling features (e.g., the UML profile for schedulability, performance, and time [9]). Ideally, performance models should be automatically extracted from well-established modeling formalisms. However, obtaining accurate models, including bottlenecks, is likely to require intervention of a skilled analyst. The approach presented here aims at reducing this intervention.

1.2 Modeling Formalisms

Probably the best-known and most widely used software modeling language is the *Unified Modeling Language* (UML)

• *The authors are with the Ghent University—IBBT—IMEC, Department of Information Technology, Gaston Crommenlaan 8, 9050 Gent, Belgium. E-mail: {tom.verdickt, bart.dhoedt, frank.gielen, piet.demeester}@intec.ugent.be.*

Manuscript received 29 Oct. 2004; revised 23 May 2005; accepted 6 June 2005; published online 12 Aug. 2005.
Recommended for acceptance by E. Weyuker.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0253-1004.

[10]. Consequently, UML diagrams will be used in this work to model the system architecture and its performance.

The *Model Driven Architecture* (MDA) [11] is a recent effort to improve the use of modeling in system design, by prescribing how a system should be modeled. MDA describes what types of models should be used, how those models should be used, and how the model types relate to each other.

An important aspect of MDA is the definition of different categories of models. The most important model types are the platform independent models (PIM) and the platform specific models (PSM). A PIM addresses the operation of the system, independent of supporting platform details such as the middleware. PSMs give a more detailed, lower-level view of the system, taking (part of) the underlying platform into account. MDA also focuses on transformations between system models (most importantly from platform independent models to platform dependent models).

The transformation presented in this paper and the models it uses all follow the MDA methodology.

1.3 Distributed Systems

System modeling and, specifically, performance modeling, becomes even more complex when considering distributed systems. Distributed systems are a response to the growing demands for processing power and the geographical spreading and heterogeneity of processing power, data sources, and storage. They consist of several collaborating components (both hardware and software) connected by a network.

Often, middleware is used to enhance the interoperability between the various system components. Middleware offers the advantage of location transparency, platform and programming language independence, event handling, etc. Important middleware standards include the *Common Object Request Broker Architecture* (CORBA) [12], Java *Remote Method Invocation* (Java RMI), Web Services, etc.

The growing interest in distributed systems has resulted in a growing interest in performance engineering techniques for those systems. Several efforts to model and predict the performance of middleware-based systems have already been undertaken [13], [14], [15], [16], [17]. Using these models requires a detailed knowledge of the internals of the middleware (and of the modeling language itself) in order to be able to adjust the model to the specific characteristics of the system and to integrate the middleware model into the overall model of the system.

On the other hand, using the MDA philosophy, one should be able to construct a PIM of the system, omitting the platform details (e.g., the middleware), which could then be transformed automatically to a PSM that includes all the details necessary to implement the system (and to obtain performance estimates). That way, the architects do not need to know the full details of the middleware. Those details will be inserted by the PIM-to-PSM transformation tool. This way of modeling would also allow rapid evaluation of the performance of the system with several different middleware technologies, in order to find the one with the best results.

This paper presents an algorithm that performs part of the transformation from a PIM to a PSM of a distributed system by including the middleware details into the model.

1.4 Approach

The goal of the research reported here is the development of a framework for the automatic modeling of the impact of the middleware on the architecture and the performance of distributed software systems. The framework semi-automatically constructs the UML model of a distributed system that uses middleware. This is done by transforming a middleware-independent UML model into a middleware-aware UML model (effectively an MDA PIM-to-PSM transformation). This model allows to obtain more fine-grained performance models, leading to a performance model where the potential bottlenecks situated in these lower layers also become apparent. This allows developers to easily assess the impact of using a certain type of middleware on the system performance, enabling them to detect possible performance problems as early as possible in the development process.

The input to the transformation consists of a high-level, middleware-independent UML model, constructed by the system designers, together with some middleware-specific information (mapping of specific middleware components, like a naming server, to a processor, etc.). This middleware description is supplied to the transformation algorithm as a separate file, containing, for example, execution times for various middleware components, deployment information for additional middleware services, etc. The UML model can be seen as a PIM (where the middleware is considered the "platform"), while the middleware information describes specifics of the platform.

The transformation output is a more detailed UML model (a PSM) of the system, containing all the necessary details of the middleware, both architectural and performance-related.

The framework (see Fig. 1) consists of a transformation algorithm and a library of middleware descriptions, each containing the middleware-specific part of the transformation for that type of middleware. The middleware library gives designers the opportunity to rapidly model the system using different types of middleware, without having to delve into the internals of all those different middleware types. The obtained models can then be used to compare the system performance using the different types of middleware and make a well-founded decision about which middleware to use.

The transformation framework described in this paper follows the modeling approach used in [8] and [18]. There, system models, described using UML activity, collaboration, and deployment diagrams, are transformed into layered queueing network performance models from which performance estimates can be extracted using existing tools. By using the same types of UML diagrams, cooperation between the tools can be ensured, allowing the output of

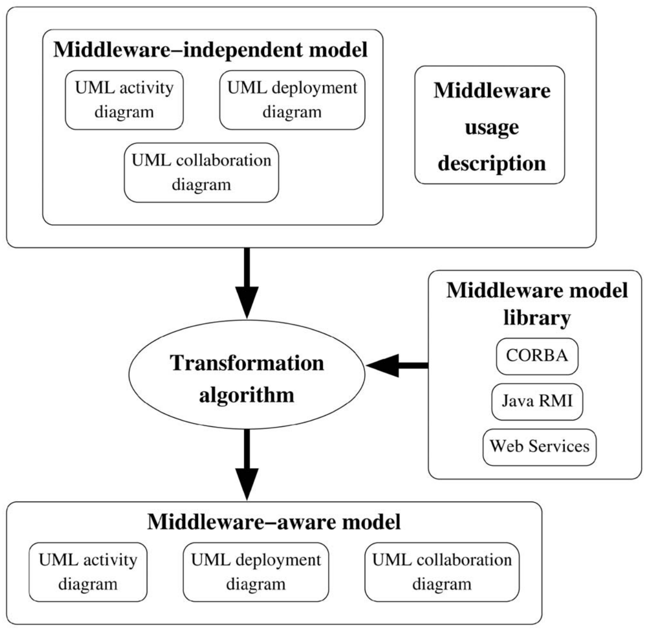


Fig. 1. The transformation framework.

this methodology to be transformed to a performance model by using the algorithms and the tools described in [8] and [18].

In this paper, the current prototype of the transformation framework will be described, using the inclusion of details of the CORBA middleware into a UML model of a sample application as an example.

Section 2 will give a short introduction to MDA. In Section 3, an overview of CORBA will be presented, with the relevant aspects of the "UML profile for schedulability, performance and time" following in Section 4. Sections 5 and 6 will provide a description of the model transformation framework and the actual transformation algorithm for CORBA, which will be used in a case-study in Section 7. Finally, in Section 8, the conclusions of this research will be presented.

2 Model Driven Architecture (MDA)

The goal of MDA is to provide models that are portable, interoperable, and reusable. This is accomplished by letting designers specify the system and the supporting platform separately, providing a separation of concerns at the architectural level. This makes modeling much easier because the system can be modeled without taking the details of the platform into account and vice versa. It also facilitates transforming the specification of the system into one on a different platform, as the system itself and the platform are described separately.

A system description following the MDA guidelines consists of several models, representing the system from several viewpoints, with different levels of abstraction. The system representation from these viewpoints may use any modeling language (ranging from general purpose modeling languages like UML to languages specific to the system application domain).

A high-level type of model used in the MDA is the *Platform Independent Model* (PIM). A PIM can be used across different platforms (as long as they are relatively similar, e.g., with regard to their interface to the system), allowing rapid remodeling for a different platform. To achieve platform independence, a PIM can, for example, model the system for a technology-neutral virtual machine.

Another abstraction level is the *Platform Specific Model* (PSM). A PSM is a combination of a PIM of the system with the supporting platform usage. A PSM might provide all the details necessary to implement the system, or it could be rather high-level, acting as a PIM in a transformation to a more detailed PSM. As such, the modeling can be layered, gradually adding detail and, thereby, allowing several levels of model abstraction. During the development, the model gradually becomes more detailed as more and more design decisions are made.

The supporting platform itself is described using a *Platform Model*, representing the technical details of the platform and the services provided by the platform.

Much of the effort involving the MDA has gone to the automation of the system design. The different viewpoints of the MDA could help such automated design, or at least make it less complex. The PIM, annotated with some extra information, could be transformed automatically (or semi-automatically) to a PSM of the system. If necessary, some additional information (in the form of extra models) can be supplied to the transformation process, as extra input, together with the PIM.

This paper will present a transformation framework to transform a high-level PIM to a lower-level PSM by adding middleware details to the model.

3 CORBA

The core of the CORBA architecture is the *Object Request Broker* (ORB). The ORB provides the communication infrastructure between the client and the server, irrespective of their programming language, application architecture, or supporting platform.

A client attempting to make a request to the server will not send the request directly to the server, but rather to a local *stub* (created during initialization), acting as a local proxy of the server. The stub will pass the request on to the ORB, which will send the request to the server-side ORB using the network. When the request arrives at the server-side ORB, the ORB will deliver it to the *skeleton* (the server-side equivalent of the stub), which, in turn, will forward the request to the server. The server processes the request and sends the response back to the client, using the same path through the different components, in the opposite direction. This communication mechanism provides a form of location transparency to the client. The client only communicates with the local stub, thereby getting the impression that the server also resides on the same computer as the client.

The stub and the skeleton perform additional operations on the request and the response (marshaling and unmarshaling), to transform the data (e.g., parameter values) from the native format to a language independent wire format and back. This allows cooperation between clients and servers, implemented using different programming languages and running on various platforms.

Before a client can send a request to a server object, it needs to obtain a reference to the server, indicating, for example, its location in the network and the port it is listening on. One way to obtain a reference is by using a *Naming Service* (NS). The NS binds canonical server names to remote object references and can be queried by a client to obtain a reference to a server object.

A CORBA implementation may also provide additional CORBA services, as described in the CORBA specification. Examples include a security service, an event service, an interface repository, etc.

This short overview of CORBA indicates that there are some components of CORBA influencing the performance of a distributed system using it. For example, the marshaling and unmarshaling of requests and responses will incur some overhead, as will using any additional CORBA services, like querying the NS. These services might even become a bottleneck, e.g., if many clients try to obtain a server reference from the NS concurrently.

These aspects of CORBA need to be represented in the system model in order to obtain accurate performance estimates for a CORBA-based distributed system. Therefore, these CORBA-specific features (components and interaction logic) will be inserted into the system model when transforming it from a middleware-independent PIM to a middleware-aware PSM. The methodology described in this paper includes the performance influence of the various middleware components during the PIM-to-PSM transformation, by adding components to the models to represent the naming service, the marshaling and unmarshaling of requests, etc.

4 The UML Profile for Schedulability, Performance, and Time

The *UML Profile for Schedulability, Performance, and Time* provides the possibility to: [9]

- enable the construction of models that could be used to make quantitative predictions regarding time, schedulability, and performance-related aspects of real-time systems;
- facilitate communication of design intent between developers in a standard way;
- enable interoperability between various analysis and design tools.

The profile provides abstractions to be used in describing the performance of a system. *Scenarios* define response paths and can have Quality of Service requirements or other kinds of performance information, such as response times or throughputs. Scenarios are executed by *workloads* (sometimes called job classes), which can be open (with a given arrival pattern, such as Poisson arrivals) or closed (with a fixed number of clients or jobs). *Scenario steps* are the elements that compose a scenario. They are joined in a sequence with forks, joins, loops etc., and may have different levels of granularity, from elementary operations to complex subscenarios. Each step has a mean number of executions (the number of times it is repeated each time it is executed), a *host execution demand* (the execution time on its host device), and optionally demands to other resources (not defined by the UML model, but intended for the performance modeling tool). The *resources* themselves are modeled as servers, either active or passive, and having a service time (active resource) or a holding time (passive resource).

In UML, scenarios are most directly modeled as either collaborations or activity graphs. The other performance components described above are modeled by using a system of stereotypes and tagged values in this specific profile. Scenario steps, for example, are identified by stereotyping each action or subactivity state in the activity graph as a `<<PAstep>>`. When using collaborations to represent scenarios, the `<<PAstep>>` stereotype should be applied to messages and stimuli. The execution details of the scenario steps are then provided by associating tagged values with the steps. For example, the execution time is represented by the tagged value `PAdemand`.

The advent of UML 2.0 will probably bring changes to the UML performance profile. At the time of this research and writing, however, the UML 2.0 standard was not yet finalized. Therefore, the transformation framework described here still uses UML 1.4. A modeling formalism change to UML 2.0 will, however, not significantly impact the transformation algorithm presented in this paper, except for some implementation details and possibly a change in the performance description.

5 Input to the Transformation

The input to the transformation contains the following elements, as shown in Fig. 1:

- a UML activity diagram, detailing the operation of the system;
- a UML deployment diagram, showing the allocation of the software components of the system to the processing nodes and the interconnection between the processing nodes;
- one or more UML collaboration diagrams, describing the architectural patterns used in the system (how the software components interact);
- a description of the middleware usage: what type of middleware the system uses, which processing nodes some middleware components are located on, and some performance parameters for middleware-specific components (e.g., stub, skeleton, naming service overhead).

As explained above, this follows the approach taken in [8] and [18] for describing the overall system architecture in order to provide compatibility with the UML-to-LQN transformation tools.

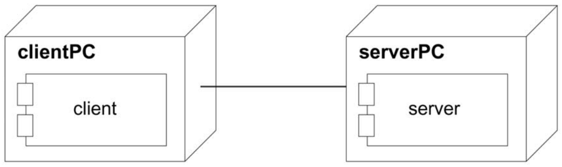


Fig. 2. Example input UML deployment diagram.

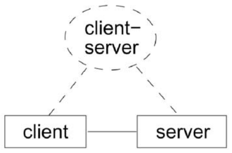


Fig. 3. Example input UML collaboration diagram.

The UML diagrams are all represented as XML files, using the XMI interface [19], generated by state-of-the-art UML tools. However, these UML tools do not yet support the performance profile. Therefore, the performance information (stereotypes and tagged values) for nonmiddleware components was included in the model on an ad hoc basis.

Figs. 2, 3, and 4 show a sample UML model to be used as input to the transformation. The modeled "system" consists of a client and a server, running on different computers, with the client making a single call to the server.

The UML diagrams are linked by the element names. A system component that is represented in different diagrams should have the same name in all the diagrams. For example, the client is represented by a component instance in the deployment diagram, a classifier role in a collaboration, and a partition in the activity diagram, all named "client."

Note the way the call is modeled in the activity diagram and, more specifically, the client part. In this example, the client makes a synchronous call, blocking until it receives a reply from the server. Asynchronous or "deferred synchronous" calls (the client performs some additional work after sending the request to the server and only blocks after that additional work has finished) should be modeled as in Fig. 5. This distinction between call types is important in order to allow the correct parsing of the activity diagram. Without such rigorous modeling, it would be very difficult (or even impossible under certain circumstances) to correctly identify the reply to a certain request, considering the possibility of callbacks.

Special care should be taken when modeling callbacks, since they could cause the activity diagram to become ambiguous (in some cases, it would be impossible to distinguish a reply to a synchronous call from a callback). Therefore, callbacks should be modeled as in Fig. 6: Add a client-side component that will accept the callback and that acts as a "server" to the callback.

The UML model does not mention the use of a middleware, but rather lets the client make a direct call to the server. The specifics of the middleware (its type, some performance information, etc.) are given in a separate input file to the transformation, for which a dedicated XML format was developed (see Fig. 7 for the DTD).

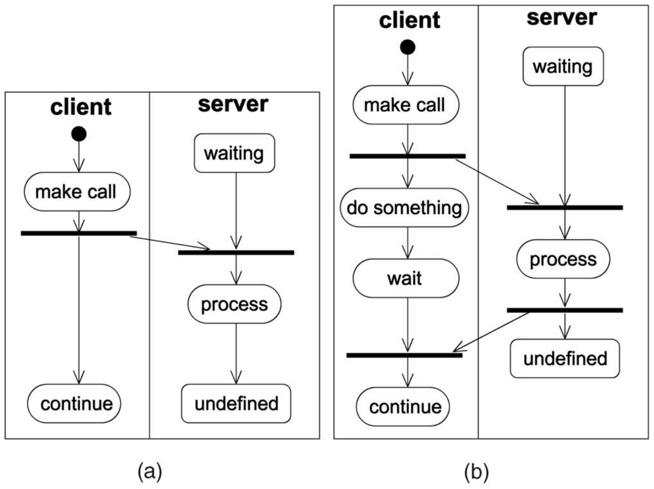


Fig. 5. Activity diagrams for other call types. (a) Asynchronous. (b) Deferred synchronous.

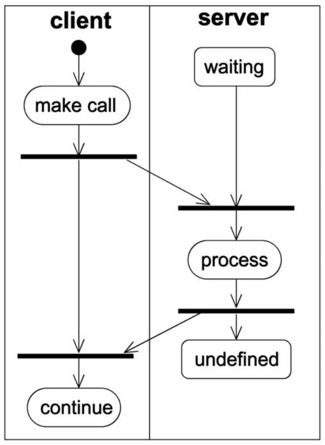


Fig. 4. Example input UML activity diagram.

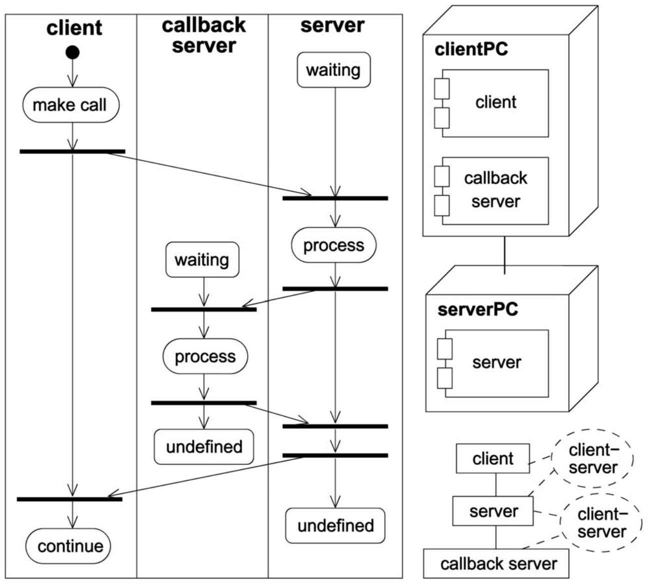


Fig. 6. UML model of a callback.

```
<!ELEMENT middleware (mw_instance|link|service)+>

<!ELEMENT mw_instance EMPTY --A single middleware instance (e.g. a CORBA ORB)-->
<!ATTLIST mw_instance id ID #REQUIRED>
<!ATTLIST mw_instance type CDATA #REQUIRED --CORBA, RMI, etc.-->
<!ATTLIST mw_instance inittime CDATA #IMPLIED --initialization time-->
<!ATTLIST mw_instance destroytime CDATA #IMPLIED --time to clear resources, etc.-->

<!ELEMENT link ((call+|endpoints),use_service*)>
<!ATTLIST link id ID #REQUIRED>
<!ATTLIST link mwref IDREF #REQUIRED --the id of the mw_instance used by this link-->

<!ELEMENT call (use_service*)>
<!ATTLIST call cref CDATA #REQUIRED --The id of the transition representing this call
                                       in the activity diagram -->
<!ATTLIST call stubtime CDATA #IMPLIED --The client-side overhead for using the middleware.-->
<!ATTLIST call skeletontime CDATA #IMPLIED --The server-side overhead for using
                                               the middleware.-->

<!ELEMENT endpoints EMPTY -- indicates the endpoints of a link, if no call-children are
                              specified (indicating that all communication between
                              the components should use the middleware -->
<!ATTLIST endpoints client CDATA #REQUIRED>
<!ATTLIST endpoints server CDATA #REQUIRED>
<!ATTLIST endpoints stubtime CDATA #IMPLIED --The client-side overhead for all calls
                                                in the link.-->
<!ATTLIST endpoints skeletontime CDATA #IMPLIED --The server-side overhead for all calls
                                                    in the link.-->

<!ELEMENT use_service EMPTY>
<!ATTLIST use_service sref IDREF #REQUIRED --References the service element of the service that is
                                               used here.-->

<!ELEMENT service (overhead?) --Description of the service (e.g. the Naming Service)-->
<!ATTLIST service id ID #REQUIRED>
<!ATTLIST service type CDATA #REQUIRED --The type of service (e.g. NS or security)-->
<!ATTLIST service host CDATA #IMPLIED --The host in the deployment diagram (if applicable)-->

<!ELEMENT overhead (initialization?,invocation?)>

<!ELEMENT initialization EMPTY --The time needed to connect to the service, set up its use-->
<!ATTLIST initialization client CDATA #IMPLIED --client-side overhead to set up the service-->
<!ATTLIST initialization server CDATA #IMPLIED --server-side overhead to set up the service-->
<!ATTLIST initialization host CDATA #IMPLIED --The overhead on the service host to set up
                                                 the use of the service in a link-->

<!ELEMENT invocation EMPTY --The time needed to use to the service-->
<!ATTLIST invocation client CDATA #IMPLIED --client-side overhead when using this service
                               (e.g. the client-side execution when calling the naming service)-->
<!ATTLIST invocation server CDATA #IMPLIED --server-side overhead when using this service-->
<!ATTLIST invocation host CDATA #IMPLIED --overhead on the service host to use the service-->
```

Fig. 7. The XML DTD for the middleware description.

`<mw_instance>` elements are used to describe "middleware instances" (e.g., a CORBA ORB). They show the type of middleware and specify the initialization time and the time needed to clean up and destroy the instance and other used resources.

`<link>` elements describe groups of calls (and their responses) between a client and a server using the CORBA platform. They reference a `<mw_instance>` element and show which additional services (if any) are used.

Each `<link>` element may contain one or more `<call>` elements, representing the actual calls that are performed in the link group and are handled by the middleware. A `<call>` element specifies the client-side and server-side overhead of using the middleware (e.g., incurred by marshaling and unmarshaling in the stub and the skeleton) and references a transition in the activity diagram that represents the call in the UML model.

Alternatively, a `<link>` may contain a single `<endpoints>` element, referencing the endpoints (the client and the server component) of the link group. It is then assumed that all communication between the two components uses the middleware. This way, however, all the stubs for the calls between the endpoints will be modeled with the same execution time, and the same goes for all the skeletons. This might introduce large modeling inaccuracies if the calls have different signatures (e.g., a different number of arguments), which would incur different marshaling and unmarshaling overheads. Therefore, the `<endpoints>` element should be used cautiously.

The use of additional middleware services (a naming service, for example) is indicated by `<use_service>` children of `<link>` or `<call>` elements. A `<use_service>` element contains a reference to a `<service>` element specifying more details of the service. A `<use_service>` child of a `<link>` element indicates

```
<middleware>
   <mw_instance id="orb1" type="CORBA"
            inittime="36.549" destroytime="3.3646" />
   <link id="link1" mwref="orb1">
      <call cref="G.11" stubtime="1.8413"
                              skeletontime="0.1021" />
   </link>
</middleware>
```

Fig. 8. A sample middleware description.

that the service is used for all the calls in the link. When <use_service> children are added to <call> elements, only those calls use the service. Note that this distinction is unnecessary for some services (e.g., the naming service), but might be important for others (e.g., an event service, where not every call might use events).

The <service> element specifies the type of service (e.g., NS or security), which will be used to correctly include it in the model (different services might need different ways of modeling). It may also contain a host attribute, which references the processor that will run the service. No host should be present if the service does not require an additional "server" apart from the components involved in the call using the service. A <service> element may also have an <overhead> child, containing an <initialization> and/or an <invocation> element. They specify the overhead in the client, the server, and the service host (if applicable) for the initialization of the service (for a single link) and for a single use of the service, respectively. All overheads are considered to be 0 if not specified.

Fig. 8 shows a possible middleware description for the model shown in Figs. 2, 3, and 4. The system will use CORBA as a middleware without any additional services. Since all the calls between client and server use the middleware, the <call> element could also be replaced by an <endpoints> element:

```
<endpoints client="S1" server="S2"
 stubtime="1.8413" skeletontime="0.1021"/>
```

with S1 being the id of the client swimlane and S2 the id of the server swimlane.

6 THE MODEL TRANSFORMATION

One way of including the middleware-induced overhead is to adjust the performance information of the existing components. This would not make the models more complex and would still allow estimation of the system performance. On the other hand, if an analysis of the model should reveal performance problems, the information would be insufficiently detailed to pinpoint the exact cause of the problem. This requires a more fine-grained model.

Therefore, it is necessary to model the middleware (and the overhead that it incurs) as separate components in the system model, with their own execution times, resource needs, and other performance parameters.

The remainder of this section will describe the concrete transformation to include the CORBA structure into the model. The transformation process consists of finding the involved components, followed by transforming the UML diagrams. The different steps of this process will be further detailed in the remains of this section. The handling of additional middleware services (<service> and <use_service> elements) will be deferred until the end of this section in order to simplify the algorithms and the figures, although, in reality, the services will be included in the model together with the other middleware components.

6.1 Additional CORBA Components

As explained above, transforming a system model to include CORBA (or any other type of middleware) will be performed by adding several new, middleware-specific components to the model. These components reflect the architectural changes incurred by using middleware, as well as the impact on the overall system performance.

Considering the different abstraction layers offered by MDA (PIMs, multiple levels of PSMs), it is clear that the middleware (like the system) can be modeled with different levels of detail. More detailed models (reflecting the exact software architecture of the middleware implementation under study) can generate more accurate performance estimates, but cause the system models to be more complex, compared to more high-level models (showing the middleware from a functional viewpoint, how it interacts with the rest of the system). Because this research is aimed at performance modeling at the architectural level, we have opted for a functional middleware modeling level.

Obviously, the *stub* and the *skeleton* need to be included in the model. These components perform the marshaling and unmarshaling of requests and responses and the transfer of those messages. The ORB part of the communication overhead will not be modeled separately, but will instead be included in the stub and skeleton components to avoid unnecessary complication of the model.

The *ORB* will still be included in the model, but not for the communication part. The initialization and the destruction of the stub (and of the ORB itself) is taken into account in the model. The initialization and destruction of the skeleton will not be modeled because they only happen at server startup and shutdown, whereas the goal of the model is to accurately model the runtime behavior of the system (more specifically, its performance).

Some components are added purely for convenience. *corba_client* is an example of such a component. The corba_client does not represent any real-life behavior of the system. It simply calls the other components of the model, serving as a link between them. It was added to simplify the transformation algorithm by limiting the changes to the original system components to a bare minimum.

If the system uses any additional middleware services, then these need to be modeled as well. The actual components to be added to the model might vary from one service to another (or even from one service implementation to another), but will generally consist of one or

```
function find involved components
begin
 for each <link> element in the
             middleware usage description do
 begin
  if <link> element has <endpoints> child
  begin
   find partitions referenced by "client" and
          "server" attributes of <endpoints>;
   get the names of those partitions;
  else
   take first <call> element;
   find transition referenced by "cref";
   get names of source and target partitions;
  end if;
 end for;
end;
```

Fig. 9. Transformation algorithm, finding the components.

more components at the client, the server, and a possible third computer (the service host).

Obviously, multiple instances of some or all of these components can be added to the model, depending on the concrete application (more accurately, depending on the actual use of the middleware, as described in the "middleware usage description" input file). A separate Naming Service component, for example, will be added for each `<service type="NS">` element in the middleware usage description. For other components (not service-related), an instance (or a separate component with a similar function, e.g., a stub for another server) is added for every connection between a client and a server that uses CORBA (meaning, for every `<link>` element in the middleware usage description). ORB components (specified by an `<mw_instance>` element), on the other hand, are included only once for every client that uses them (meaning, for every component that participates as a client in one or more links that use the ORB instance). Thus, it is perfectly possible for multiple stubs and skeletons to appear in the resulting model, even for a single client or server component in the original model, as long as the stubs and skeletons belong to separate links.

6.2 Locating the Involved Components

Before the transformation can start, it is obviously necessary to locate the components that are involved in the transformation. These are the components that make use of the middleware (the client and the server). This needs to be done for each `<link>` element in the middleware usage description (see Fig. 8) since each `<link>` element has its own client and server.

The pseudocode for this part of the transformation can be found in Fig. 9. If a `<link>` element contains an `<endpoints>` child, its `client` and `server` attributes directly reference the partitions in the activity diagram that represent the components involved in the link. If all the calls in the link are specified separately, then the components are found by looking up the transitions that are referenced in the `<call>` children of the `<link>`. Or rather, this is done for only one `<call>` child because all `<call>` elements of a single `<link>` element should reference transitions between the same components. The names of the partitions in the activity diagram that contain the source and destination of the transition are the names of the components involved in the link. In the example, these are the components `client` and `server`, acting as client and server. They are also the names of the roles in the collaboration diagrams and of the component instances in the deployment diagram that are involved in the transformation.

```
function transform deployment diagram
begin
 for each <link> element do
 begin
  find nodes that contain source and target;
  add corba_client and stub to source node;
  find orb referenced in "mwref" attribute;
  if orb not yet added to source node
  begin
   add orb to source node;
  end if;
  add skeleton to target node;
 end for;
end;
```

Fig. 10. Transformation algorithm, deployment diagram.

6.3 The Deployment Diagram

The actual transformation starts with the transformation of the deployment diagram (Fig. 10). First of all, the processors that will run the new components are located in the deployment diagram. These are the node instances that contain the component instances identified earlier (in this case `clientPC` and `serverPC`, containing `client` and `server`). The new components are added to these processors: `corba_client`, `orb` (if it was not yet added), and `stub` to the `clientPC` and `skeleton` to the `serverPC`.

As an illustration, the final deployment diagram for the client-server system of the example, is shown in Fig. 11.

6.4 The Collaboration Diagram

The main goal of the collaboration diagram is to provide an architectural overview of the system, indicating the architectural software patterns that were applied, in order to allow structured parsing and processing of the activity diagram. Adding CORBA to the system obviously changes the system architecture, demanding that the collaboration diagram be adjusted to reflect the new architecture. Specifically, the new system components (such as the ORB, the stub, and the skeleton) need to be added, together with their relation to the other system components and to each other.

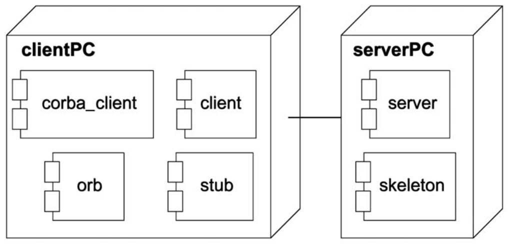


Fig. 11. Transformation result: UML deployment diagram.

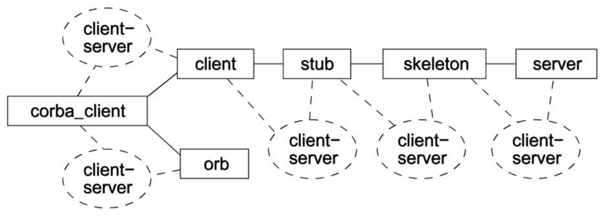


Fig. 12. Transformation result: UML collaboration diagram.

The collaboration diagram resulting from the transformation is shown in Fig. 12. It was obtained by applying the algorithm of Fig. 13. The structure of the collaboration diagram is rather straightforward. It contains a classifier role for every middleware component (as described earlier) and links them all in several client-server collaborations.

```
function transform collaboration diagram
begin
 for each <link> element do
 begin
  find source and target roles;
  find collaboration between source and
                              target role;
  create roles corba_client, stub and skeleton;
  find orb referenced in "mwref" attribute;
  if orb not yet added then
  begin
   create role orb;
  end if;
  create client-server collaboration between
               corba_client and orb,
               corba_client and client,
               client and stub,
               stub and skeleton,
               skeleton and server;
 end for;
 delete collaborations between
           original source and target roles;
end;
```

Fig. 13. Transformation algorithm, collaboration diagram.

6.5 The Activity Diagram

The activity diagram should be adjusted in three places during the transformation in order to include the middleware into the model. Fig. 14 shows the pseudocode for the activity diagram transformation (which will be explained in more detail in the rest of this section) in case the middleware-using calls are specified separately in the middleware usage description (using `<call>` elements). The algorithm can be adapted to work with `<endpoint>` elements, by iterating over the calls (transitions) in the activity diagram (between the client and the server), instead of over the `<call>` elements.

Before the first call[1] is made from the client to the server, an initialization phase should be inserted, e.g., at the start of the client. The calls themselves need to be redirected to use the stub and the skeleton. Finally, the ORB and the stub must be destroyed (along with some other clean-up operations) after the final call.

It is important to note that the transformed collaboration diagram contains only client-server collaborations (at least the part that uses the middleware, since CORBA is designed for client-server systems). To a certain degree, this will be reflected in the activity diagram: calls for a single `<link>` will always have the same source (the "client") and the same target (the "server"), with a possible response in the opposite direction (though not necessary, as calls can be asynchronous).

The resulting activity diagram (for a synchronous call) is shown in Fig. 15. An explanation of the stereotypes and tagged values will be given in Section 6.7. The activity diagrams for asynchronous and deferred synchronous calls are presented in Figs. 16 and 17. The different steps of transforming the activity diagram will be presented next.

The initialization phase (e.g., initializing the ORB) is modeled by an `initialize` action in the `orb` partition of the activity diagram. After the initialization phase, the client can start sending requests to the server. For every `<call>` element in the middleware usage description, the referenced transition is obtained. If the call is synchronous or deferred synchronous, the reply is located as well (how this is done will be explained below). The requesting transition is redirected in order to use the stub and the skeleton to send a request to the server. The reply follows the same route in the opposite direction if there is a reply. Otherwise, only the request part needs to be transformed. If the middleware description only indicates the endpoints of a link, instead of the individual calls, then all calls between those endpoints (and from the client to the server) will be located and the previous algorithm will be executed for each of those calls.

Finding the reply to a given call can be done as follows (and by extension, this algorithm can also be used to check whether the call is synchronous or not, because calls for which no reply can be found are asynchronous). It is important to note that the request transition will start in a fork (see Figs. 4, 5a, and 5b). Start by following the "path" through the server, started by the request transition, until a transition from the server to the client can be found (or, rather, a transition from the server to a join, which has an outgoing transition to an action in the client swimlane). If the target of this transition (or of the outgoing transition of the join) can be reached by following the other path from the "request fork" (following the client swimlane), then this transition (from the server to the client) represents the reply of the earlier request transition.

If this is not the case, then this transition represents the request in a callback from the server to the client (an alternative way of modeling a callback, though the model of Fig. 6 should be preferred). In this case, keep following the path until another transition from the server to the client has been found, which becomes the new reply candidate, and make the same test, etc. If no reply can be found corresponding with a certain request, then the request was an asynchronous call.

Just before the client ends, a request is made to the ORB in order to free the used resources and to destroy the ORB. As long as this clean-up phase is included after the last call

1. Here and in the rest of this section, *call* refers to a call from the client to the server that uses CORBA and is referenced by a `<call>` element in the middleware usage description, or any call from the client to the server in case the middleware usage description directly specifies the `<endpoints>`.

```
function transform activity diagram
begin
 for each <link> element do
 begin
  find the transition referenced in the first <call> child of the element;
  find the source and target partitions;
  create partition corba_client;
  if orb not yet added then
  begin
   create partition orb;
   create action initialize in orb;
   create actions call_init and call_client in corba_client;
   find starting state of source partition (say s);
   find target of transition starting in state s (say s');
   create transitions s -> call_init -> initialize -> call_client -> s';
   fine ending state of source partition (say e);
   find source of transition incoming to e (say e');
   create action dummy in source partition;
   create actions call_destroy and return_client in corba_client;
   create action destroy in orb;
   create transitions e' -> dummy -> call_destroy -> destroy -> return_client -> e;
  end if;

  find the source action of the transition to the source of this call (say, action a);
  create partitions stub and skeleton;
  for each <call> child of the element do
  begin
   find the transition of this <call>;
   find the source and target;
   create action stub_request in stub;
   create action skeleton_request in skeleton;
   create transitions source -> stub_request -> skeleton_request -> target;
   find reply transition for this call;
   if reply found then
   begin
    find source and target of reply transition;
    create action stub_reply in stub;
    create action skeleton_reply in skeleton;
    create transition source -> skeleton_reply -> stub_reply;
   end if;
   if call is (deferred) synchronous then
   begin
    create transition stub_reply -> target of reply;
   end if;
  end for;
 end for;
end;
```

Fig. 14. Transformation algorithm, activity diagram.

using the ORB, the exact location does not really influence the model or the performance estimates obtained from it.

A new action (dummy) is inserted into the client partition, just before the ORB destruction. This action does not have a functional meaning and is inserted only for the convenience of performance estimation. If the ORB destruction would simply be included just before the final action (or state) of the client and, if that state would be reached by a transaction modeling a reply to an earlier call, for example, then the ORB destruction would influence the performance estimates of that call. The extra dummy action allows to separate the CORBA overhead from the rest of the client operation.

6.6 Middleware Services

If the system uses any additional middleware services (naming, security, interface repository, etc.), then that will have an impact on the system model. Not only the services themselves need to be included in the model, but using them might change the interaction between other components.

There are two options for the inclusion of middleware services, requiring different input models. The first option is to let the transformation tool make all the decisions regarding the services (both software and hardware), except perhaps for the decision to use them. This has the advantage that the system designers hardly need to be aware of the fact that the services are even available for that type of middleware. On the other hand, the designers also have no control over how the services are incorporated into the system exactly. All the details would have to be specified by the transformation tool. For example, when modeling the naming service, the transformation tool would have to decide which processor should contain it. This could be a new processor (only supporting the naming service) or an existing one (already used by some other system components). Additionally, for every new processor, the tool would have to decide how the processor should

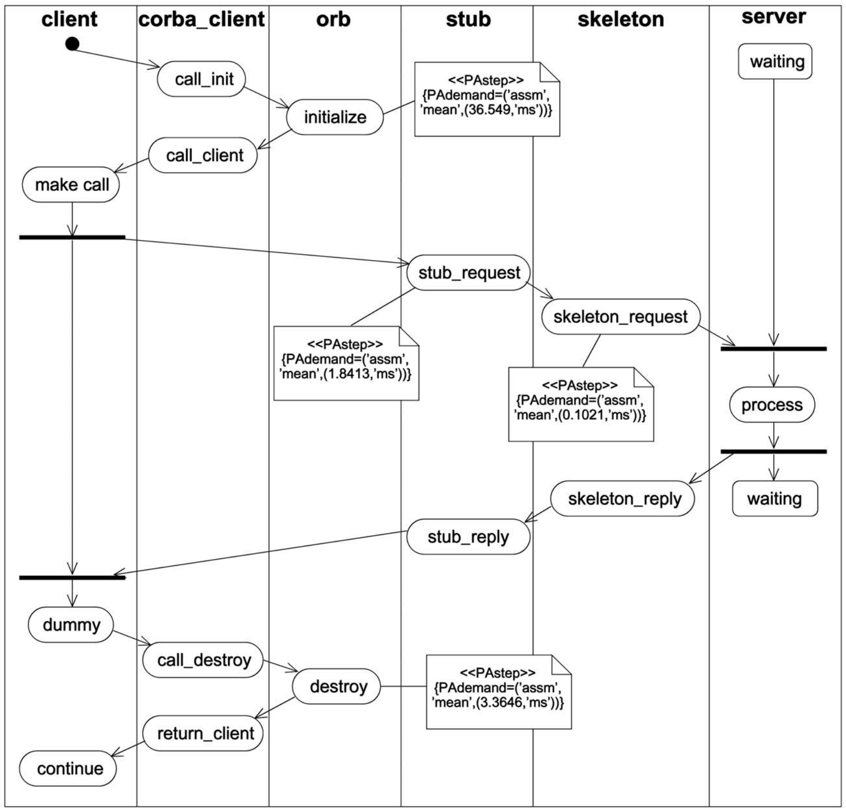

client
corba_client
orb
stub
skeleton
server
call_init
initialize
<<PAstep>> {PAdemand=('assm', 'mean',(36.549,'ms'))}
call_client
make call
waiting
stub_request
skeleton_request
<<PAstep>> {PAdemand=('assm', 'mean',(1.8413,'ms'))}
<<PAstep>> {PAdemand=('assm', 'mean',(0.1021,'ms'))}
process
skeleton_reply
waiting
stub_reply
dummy
call_destroy
destroy
<<PAstep>> {PAdemand=('assm', 'mean',(3.3646,'ms'))}
return_client
continue

Fig. 15. Transformation result: UML activity diagram.

be connected to the rest of the system. These are important design decisions that should be made by the system designers themselves (when making the PIM-to-PSM transformation), rather than by an automated model transformation tool.

Therefore, another approach was adopted in this tool with regard to middleware services. The system designers do decide whether to use a service or not, and they specify certain details of its use, like which processor the service will run on, its connection to the network, and how the service will be used (by which clients and servers). Therefore, this information should be specified in the middleware usage description (which processor should run the NS) and the input UML model (how the processor fits into the network topology of the system). This might mean adding “empty” processors to the deployment diagram (without a component running on them) to be used by one or more services.

How the use of the service needs to be modeled depends on the service at hand. Some services will be invoked only once (e.g., during client or server initialization), while

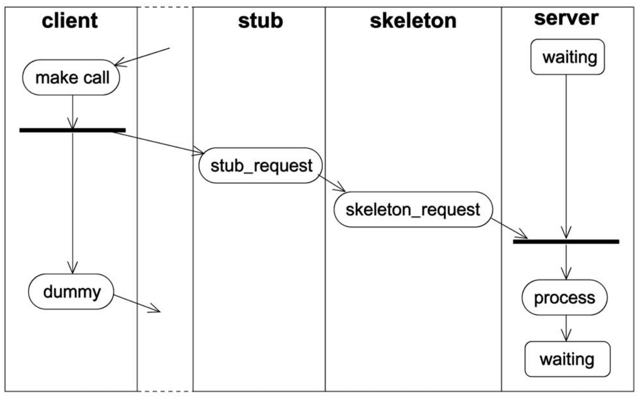


Fig. 16. Transformation result: activity diagram with an asynchronous call.

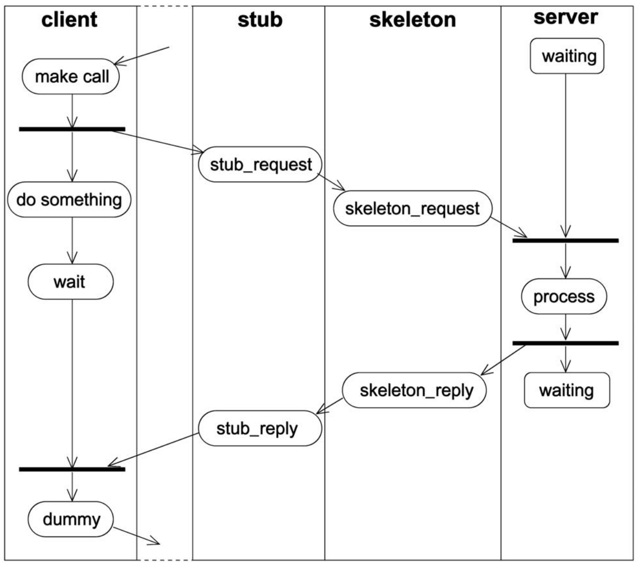


Fig. 17. Transformation result: activity diagram with a deferred synchronous call.

```
<middleware>
  <mw_instance id="orb1" type="CORBA" />

  <link id="link1" mwref="orb1">
    <call cref="G.11">
    <use-service sref="NS1" />
  </link>

  <service id="NS1" type="NS" host="spc">
    <overhead>
      <initialization client="0.3" host="0.1" />
      <invocation client="0.46" host="0.53" />
    </overhead>
  </service>
</middleware>
```

Fig. 18. A middleware description using the Naming Service.

others are used for every or some of the calls between a client and a server. As an example, sample diagrams will be presented for two services, the Naming Service and the Security Service, which have a distinctively different interaction with the other system components.

6.6.1 *Naming Service*

The naming service is only used during the initialization of a client (or rather, of a connection between a client and a server). After the connection to the naming service itself is established, it is queried to obtain a reference to the server the client wishes to contact. For example, performing the transformation with the middleware description of Fig. 18 on the UML model of Figs. 2, 3, and 4, yields an activity diagram which starts as shown in Fig. 19 (irrelevant parts and performance information not shown for clarity). The transformation of the collaboration and deployment diagrams is rather straightforward (adding the `naming_context` and `NS` components) and will be illustrated in the case-study in Section 7.

6.6.2 *Security Service*

The security service is used in a completely different way than the naming service. Instead of calling the service during initialization in order to obtain a server reference or some other information, the security service is used whenever a message (call, reply) needs to be secured, e.g., to provide privacy or authentication. Therefore, using the security service will impact the way calls are modeled, e.g., like in Fig. 20.

6.7 Performance Attributes

Another part of transforming a PIM to a PSM, apart from the structural changes due to including middleware components, is the addition of performance information for those middleware components (performance parameters of nonmiddleware components are outside the scope of the transformation tool). This information is described using the UML Profile for Schedulability, Performance, and Time.

Using the profile, there are two options in representing performance information. The performance information can be included in a collaboration diagram or in an activity diagram. Since the activity diagram generated by

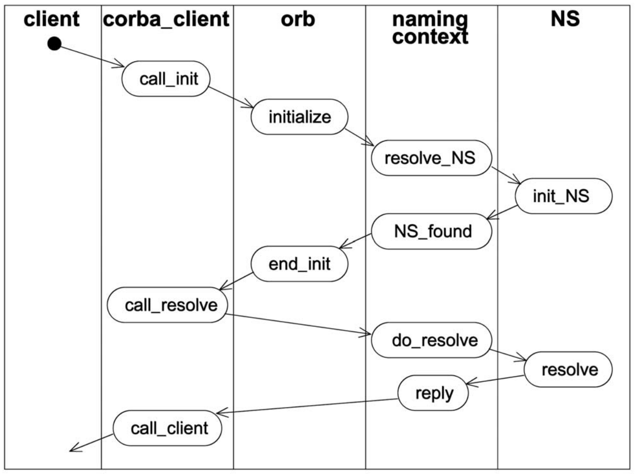


Fig. 19. Activity diagram for a system which uses the NS (initialization part).

the transformation tool gives a more detailed overview of the system than the collaboration diagram, the performance information will be included in the activity diagram. More specifically, execution times will be specified for the relevant middleware-related actions. This will be done by giving the actions the `<<PAstep>>` stereotype and specifying a tagged value `PAdemand` to represent the execution time.

No performance parameters are specified for actions like `skeleton_reply` or `stub_reply`. The reason is that these actions are mainly presented in the model to accept the response to an earlier transition. The possible (small) execution time of these actions can be included in the earlier action in the same partition (e.g., `skeleton_request` and `stub_request`). Similarly, none of the actions in the `corba_client` partition get performance parameters, as those actions only serve to connect the different parts of the model, without having functional real-world counterparts.

The actions for which performance information needs to be specified are `initialize`, `destroy`, `stub_request`, and `skeleton_request`, plus the relevant actions when using additional middleware services.

The execution times for those actions will be obtained from the middleware usage description. The `inittime` attribute of the `<mw_instance>` element will serve as the `PAdemand` tagged value of the `initialize` action. Likewise, the attribute `destroytime` will specify the values for the action `destroy`. `stubtime` and `skeletontime` will provide execution times for the actions `stub_request` and `skeleton_request`. The execution times of the service actions are found in the `client`, `server` and `host` attributes of the `<initialization>` and `<invocation>` elements of the service.

The stereotypes and tagged values that are added to the activity diagram are shown as notes in Fig. 15. The performance data is obtained from the middleware description of Fig. 8.

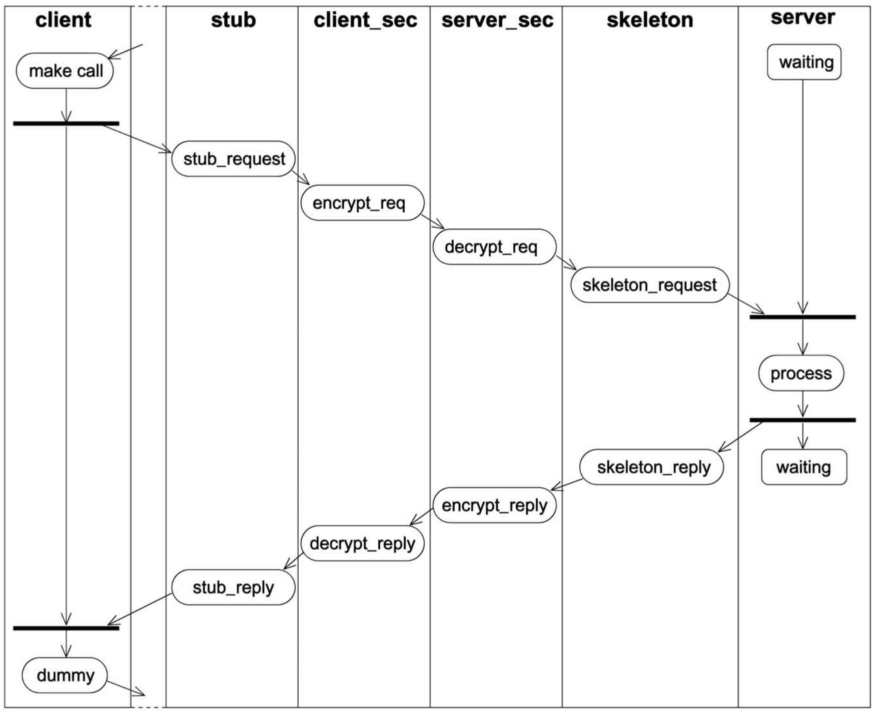


Fig. 20. Activity diagram for a call which uses the security service.

6.8 Transformation Limitations

The transformation framework as described in this paper places some constraints on the models it can handle. Perhaps the most important constraint is the naming convention described in Section 5: Model elements that occur in different diagrams need to have the same name, and that name should be unique in the model. This naming convention, however, is necessary to identify the occurrences of a single component in the different diagrams (deployment, collaboration, and activity).

Another limitation is that not all possible interactions currently can be handled by the transformation framework, e.g., forwarding servers (a client sends a request to server A, who forwards the request to server B, who sends the reply directly back to the client instead of first replying to server A). Since the difference between these interaction patterns is quite significant, however, they should be modeled as a different type of collaboration in the collaboration diagram (e.g., client-forwarding-server, instead of client-server). This allows the framework to detect such interaction, though it cannot yet handle them.

7 Using the Framework: A Case Study

As an illustration of the use of the UML transformation framework, a case study was conducted, modeling an online store using CORBA between the client and the server. The input to the transformation is presented in Figs. 21, 22, and 23.

The middleware usage description can be found in Fig. 24. The performance estimates for the CORBA components (given in the middleware usage description) were obtained from measurements on a prototype implementation of the online store, by instrumention of the ORB and the Naming Service. Performance parameters for the other (nonmiddleware) components are given in Table 1.

Applying the transformation framework to the high-level input model yields the low-level, CORBA-aware UML model presented in Figs. 25, 26 (containing the performance impact of the CORBA middleware), and 27.

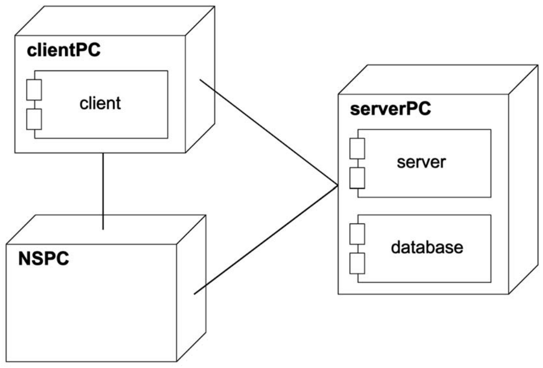


Fig. 21. Input UML deployment diagram of an online store.

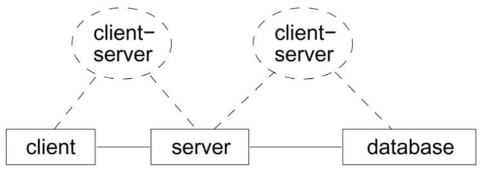


Fig. 22. Input UML collaboration diagram of an online store.

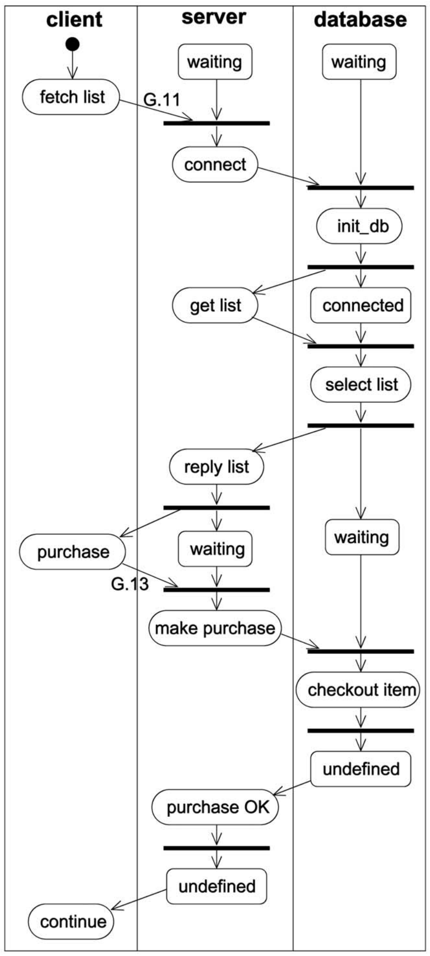


Fig. 23. Input UML activity diagram of an online store.

These models were then transformed (by hand) to a layered queueing network model of the system. The CORBA components were transformed using the CORBA LQN model described in [16]. Using the LQNS solver [5], performance estimates were extracted for varying system parameters.

The LQN model validation performed in [16] indicated that estimation errors for the delay caused by CORBA are expected to be below 5 percent of the actual measured delays. Under most circumstances, this is acceptable, considering the fact that the model (and the automated transformation algorithm) is designed to be used during the architectural design, when only estimates of the performance of the system components are available.

Fig. 28 shows the (estimated) execution time of the client for a rising request arrival rate. The system was modeled without middleware, using CORBA but no naming service, and using CORBA with the naming service. The "execution time" shown in the figure is the time that passes between starting the client and its ending (more accurately, the time to execute the entire scenario of the activity diagram of Fig. 26).

It is clear that a bottleneck occurs in the system using CORBA. Further inspection of the performance analysis output revealed that the server was the bottleneck, due to the added load of the skeleton. Similar information can be used early on during system design to assess the impact of the middleware and to assure that the middleware will not cause the system to break its performance requirements. If the system had only been modeled without the CORBA details (see the "no CORBA" line in Fig. 28), such a bottleneck would only have been detected in the actual implementation of the system, when removing the bottleneck could prove difficult.

The second part of the case study consisted of a series of tests to study whether it would be more beneficial to improve the performance of the naming server or the database, given certain load parameters. Consider $r = \alpha/\beta$,

TABLE 1
Performance Parameters for the Case-Study

Task	Processing time (ms)
$client : fetch\ list$	0.1
$client : purchase$	0.1
$server : get\ list$	0.24
$server : make\ purchase$	0.24
$database : select$	8.5
$database : checkout$	1.8

```
<middleware>
   <mw_instance id="orb1" type="CORBA"
         inittime="36.549" destroytime="3.3646">
    <link id="link1" mwref="orb1">
       <call cref="G.11" stubtime="1.8413"
                         skeletontime="0.1021" />
       <call cref="G.13" stubtime="2.07"
                         skeletontime="0.124" />
       <use_service sref="NS1" />
    </link>

    <service id="NS1" type="NS" host="xmi.17">
       <initialization client="0.3" host="0.01" />
       <invocation client="0.4597" host="0.53" />
    </service>
</middleware>
```

Fig. 24. Middleware description for the online store.

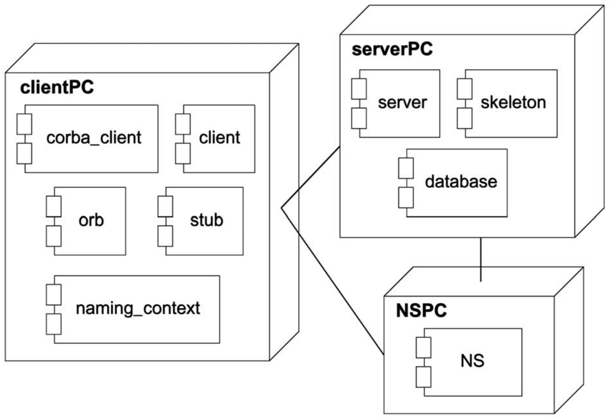


Fig. 25. Transformation result: deployment diagram of an online store.

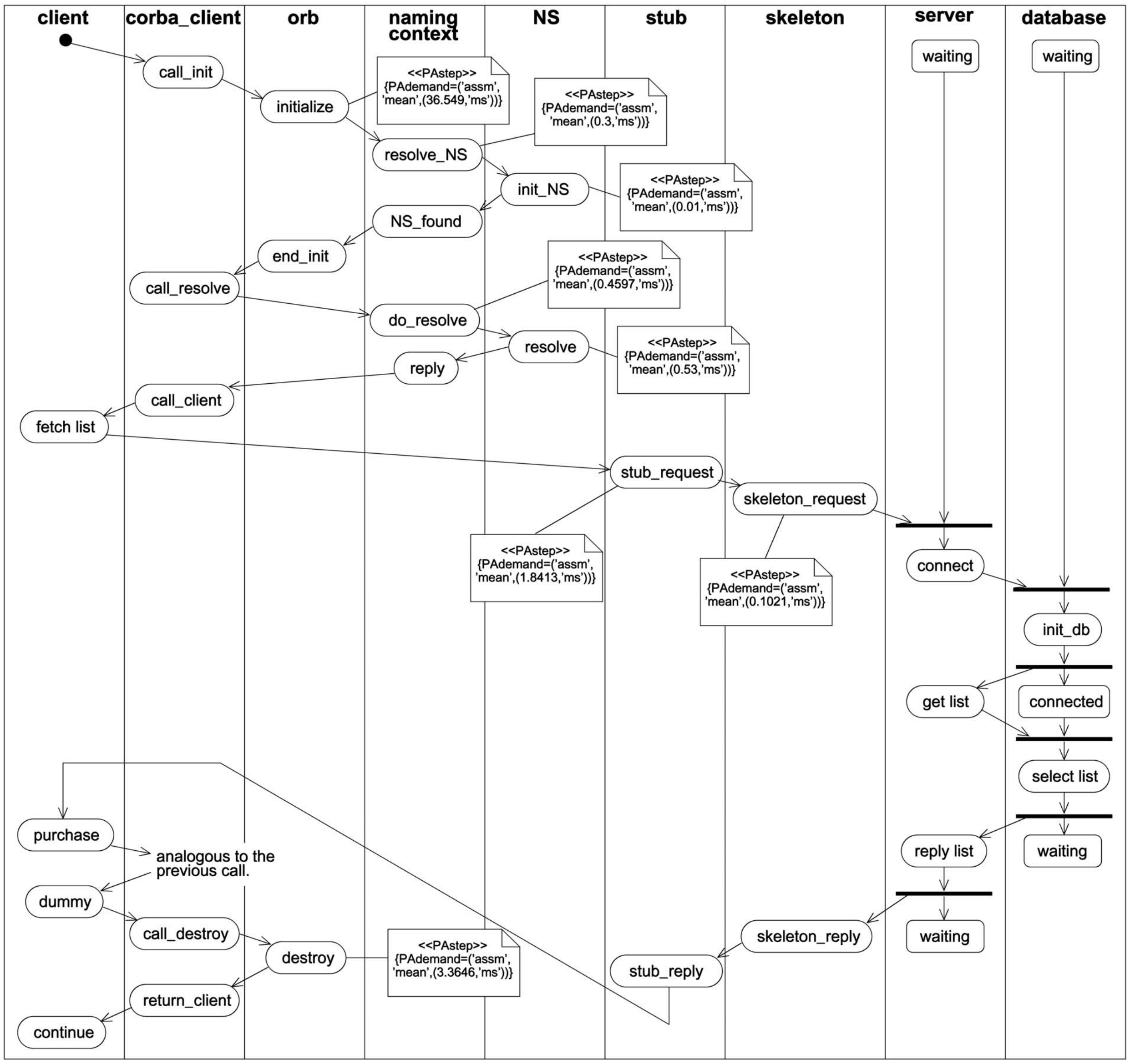

client
corba_client
orb
naming context
NS
stub
skeleton
server
database
call_init
initialize
resolve_NS
init_NS
NS_found
end_init
call_resolve
do_resolve
resolve
reply
call_client
fetch list
stub_request
skeleton_request
waiting
waiting
connect
init_db
get list
connected
select list
reply list
waiting
waiting
purchase
analogous to the previous call.
dummy
call_destroy
destroy
return_client
continue
skeleton_reply
stub_reply
<<PAstep>> {PAdemand=('assm', 'mean',(36.549,'ms'))}
<<PAstep>> {PAdemand=('assm', 'mean',(0.3,'ms'))}
<<PAstep>> {PAdemand=('assm', 'mean',(0.01,'ms'))}
<<PAstep>> {PAdemand=('assm', 'mean',(0.4597,'ms'))}
<<PAstep>> {PAdemand=('assm', 'mean',(0.53,'ms'))}
<<PAstep>> {PAdemand=('assm', 'mean',(1.8413,'ms'))}
<<PAstep>> {PAdemand=('assm', 'mean',(0.1021,'ms'))}
<<PAstep>> {PAdemand=('assm', 'mean',(3.3646,'ms'))}

Fig. 26. Transformation result: activity diagram of an online store.

being the ratio of the processing time of the database (α) and the processing time of the naming service (β). "Processing time" is used here in the sense of the time needed to process a single request, assuming no additional load. A change in the processing time can be achieved, for

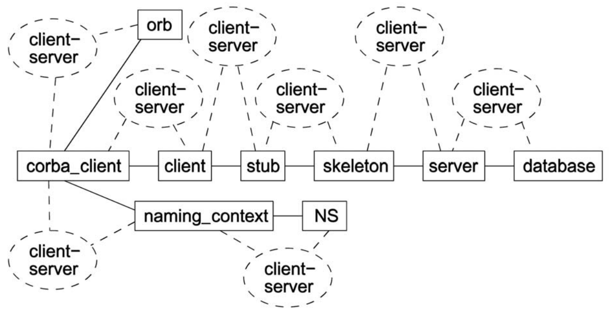


Fig. 27. Transformation result: collaboration diagram of an online store.

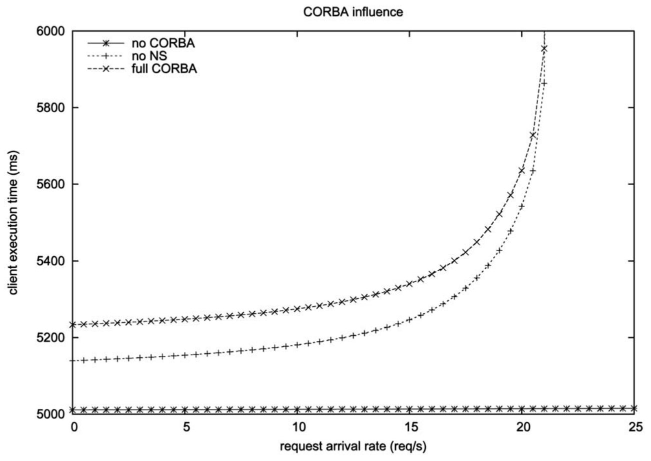


Fig. 28. Performance influence of CORBA.

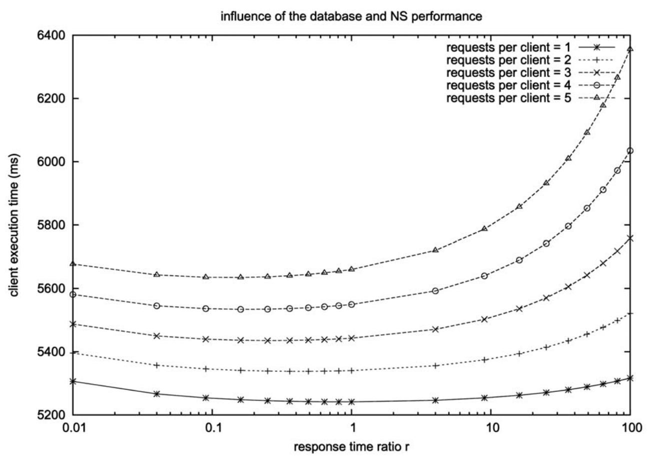


Fig. 29. Influence of the relative performance of the naming service and the database.

example, by changing the NS or the database implementation, or by changing the hardware running them.

Fig. 29 shows the client execution time as a function of r, for a fixed request rate and for different numbers of requests to the server per client (but always a single request to the naming service).

As could be expected, the performance of the database becomes more important with rising numbers of requests per client. However, if the naming service is very slow compared to the database, it might be more favorable to enhance the performance of the naming service (e.g., upgrading the hardware) than the performance of the database. Charts like Fig. 29 indicate under what circumstances that might be the case.

8 Conclusions

This paper presented a UML model transformation framework to automatically incorporate the use of middleware into the system models. The transformation includes both the structural impact of the middleware and the overhead incurred by it into the models. Thus, a middleware-aware model is obtained, starting from a middleware-independent model and a description of the middleware usage (e.g., its deployment) and its performance. A concrete algorithm performing this transformation for the CORBA middleware has been described in further detail.

The resulting UML model (a PSM from a middleware perspective) contains sufficient information to be used in modeling and analyzing the performance of the system and, more importantly, how the performance is influenced by the middleware. The resulting model can, for example, be transformed to a performance model of the system using existing transformation tools. From these performance models, estimates for the performance of the final system can be extracted, again using existing tools.

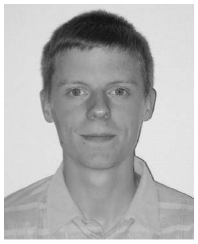
That way, the framework can be used to combine the advantages of MDA and SPE. It allows a standardized modeling of the system with separation of concerns (modeling the middleware semi-automatically) and using modeling formalisms familiar to the system architects. At the same time, it gives the possibility to obtain performance estimates as early as possible, when redesigning the system can still be done without excessive costs by providing system models that can be transformed directly into performance models.

Acknowledgments

This research was financed by a PhD grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). The authors would like to thank the Real-Time and Distributed Systems Group at Carleton University in Ottawa, Canada, for providing the tools to simulate and solve the LQN models and to transform UML models to LQN.

References

[1] C.U. Smith, "Designing High-Performance Distributed Applications Using Software Performance Engineering: A Tutorial," *Proc. Computer Management Group,* Dec. 1996.
[2] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik, *Quantitative System Performance, Computer System Analysis Using Queueing Network Models.* Prentice-Hall, 1984.
[3] M. Woodside and G. Franks, "Tutorial Introduction to Layered Modeling of Software Performance," *Proc. Workshop Software and Performance,* Aug. 2004.
[4] M.A. Marsan, C. Gianni, and B. Gianfranco, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Trans. Computer Systems,* vol. 2, no. 2, pp. 93-122, May 1984.
[5] G. Franks, A. Hubbard, S. Majumdar, J. Neilson, C. Petriu, J. Rolia, and M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems," *Performance Evaluation,* vol. 24, nos. 1-2, pp. 117-135, Feb. 1995.
[6] G. Ciardo, J. Muppala, and K. Trivedi, "SPNP: Stochastic Petri Net Package," *Proc. Third Int'l Workshop Petri Nets and Performance Models,* pp. 142-151, 1990.
[7] V. Cortelessa, A. D'Ambrogio, and G. Iazeolla, "Automatic Derivation of Software Performance Models from Case Documents," *Performance Evaluation,* vol. 45, nos. 2-3, pp. 81-105, July 2001.
[8] P.G. Gu and D.C. Petriu, "XSLT Transformation from UML Models to LQN Performance Models," *Proc. Third Int'l Workshop Software and Performance (WOSP '2002),* pp. 227-234, July 2002.
[9] Object Management Group, "UML Profile for Schedulability, Performance, and Time," Apr. 2003.
[10] Object Management Group, "Unified Modeling Language Specification, Version 1.4," 2001.
[11] J. Miller and J. Mukerji, "MDA Guide, Version 1.0.1," June 2003.
[12] Object Management Group, "The Common Object Request Broker: Architecture and Specification," 2002.
[13] P. Kähkipuro, "Performance Modeling Framework for CORBA Based Distributed Systems," PhD thesis, 2000.
[14] D. Petriu, H. Amer, S. Majumdar, and I. Abdul-Fatah, "Using Analytic Models for Predicting Middleware Performance," *Proc. Second Int'l Workshop Software and Performance*, pp. 189-194, Sept. 2000.
[15] C.U. Smith and L.G. Williams, "Performance Engineering Models of Corba-Based Distributed-Object Systems," *Proc. Computer Measurement Group Conf.,* pp. 886-898, Dec. 1998.
[16] T. Verdickt, B. Dhoedt, F. Gielen, and P. Demeester, "Modelling the Performance of CORBA Using Layered Queueing Networks," *Proc. 29th Euromicro Conf.,* pp. 117-123, 2003.
[17] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance Prediction of Component-Based Applications," *J. Systems and Software,* vol. 74, no. 1, pp. 35-43, Jan. 2005.
[18] D.C. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications," *Proc. 12th Int'l Conf. Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS 2002),* pp. 159-177, Apr. 2002.
[19] Object Management Group, "OMG XML Metadata Interchange (XMI) Specification," OMG Document formal/02-01-01, 2002.



Tom Verdickt received the degree in computer science engineering (option: software engineering) from Ghent University in 2002. In August 2002, he joined the Department of Information Technology of the Faculty of Applied Sciences, Ghent University, and is currently working towards a PhD degree. His research mainly focuses on performance modeling of distributed software at an architectural level.

Bart Dhoedt received the degree in engineering from Ghent University in 1990. In September 1990, he joined the Department of Information Technology of the Faculty of Applied Sciences, University of Ghent. He is responsible for several courses on algorithms, programming, and software development. He is author or coauthor of approximately 100 papers published in international journals or in the proceedings of international conferences. His current research addresses software technologies for communication networks, peer-to-peer networks, mobile networks, and active networks.

Frank Gielen received the Masters degree in telecommunication system engineering from the Royal Military Academy in Brussels (1985) and the PhD degree in computer science from the Free University of Brussels (1993). From 1993 until 2002, he held a number of technical and managerial functions in the software industry. He started as a software architect and technical manager with AT&T Bell Labs in the the US and was also Director of Software Technology at Alcatel. In 1998, he joined Tellium, a US-based startup company in optical network technology, as their Vice President of Software Engineering. He returned to Europe in 2001 as the CEO for Tellium EMEA. In 2002, he was appointed Technology Transfer Officer for the investment fund of the Free University of Brussels and joined the University of Ghent as a professor of software engineering.

Piet Demeester received the Masters degree in electro-technical engineering and the PhD degree from Ghent University in 1984 and 1988, respectively. In 1992, he started a new research activity on broadband communication networks resulting in the IBCN-group (INTEC Broadband communications network research group). Since 1993, he has been a professor at Ghent University, where he is responsible for the research and education on communication networks. The research activities cover various communication networks (IP, ATM, SDH, WDM, access, active, mobile), including network planning, network and service management, telecom software, internetworking, network protocols for QoS support, etc. He is author of more than 400 publications in the area of network design, optimization, and management. He is member of the editorial board of several international journals and has been member of several technical program committees (ECOC, OFC, DRCN, ICCCN, IZS, etc.). He is a senior member of the IEEE.