QoS Analysis

Valérie Issarny, Erwan Demairy, Apostolos Zarras,

Christos Kloukinas, Siegfried Rouvrais

INRIA - Rennes and Rocquencourt

Abstract: The C3DS design and development methodology integrates a number of methods and associated tools for the analysis of complex service design and implementation. In this document, we focus more specifically on methods and tools aimed at QoS analysis. This support originates from the ADL-based ASTER toolset, which was initially aimed at easing the implementation of middleware, customized for applications according to the required QoS. In the context of C3DS, this toolset is being enriched so as to cope with QoS analysis with respect to middleware customization and to temporal properties, while accounting for both structural and behavioral views of complex services.

1. Introduction

The TCCS environment targets complex service provisioning based on the description of the service's structural and behavioral views. The former describes the software architecture underlying service provisioning (i.e. the various, possibly complex, components that are composed for service provisioning) using the TCCS ADL. The latter specifies the workflow schema to be applied upon the software architecture, using the TCCS workflow language. In order to help service designers and developers, the TCCS environment offers a number of CASE tools for the analysis of complex services that are being built. In this document, we focus on the TCCS support aimed at QoS (*Quality of Service*) analysis.

QoS analysis in the TCCS environment builds upon the ASTER toolset, which was initially aimed at the systematic customization of middleware according to the non-functional requirements of applications, as stated in the corresponding ADL specifications (see *Deliverable B1.1*). Such a toolset is being enriched so as to cope with the specifics of the TCCS environment, examining more specifically customization of TCCS-based middleware and QoS analysis with respect to temporal properties. From the standpoint of middleware customization, we have to address customization according to the services' structural and behavioral views. While the former issue is quite direct to handle, given the specification of structural views using the TCCS ADL, the latter requires investigating customization based on the specification of workflow schemas. Considering QoS analysis with respect to temporal properties, this issue has been initially examined due to the increasing popularity of multimedia complex services whose design primarily lies in addressing associated soft-real time constraints. We thus have proposed a method and associated tools for the analysis of multimedia applications based on their architectural description [Demairy *et al.* 1999]. We

are now examining application of our result to the temporal analysis of workflow schemas. QoS analyses with respect to middleware customization and to temporal properties are respectively addressed in Sections 2 and 3. Results presented in these sections are further illustrated using the video-conference system that was introduced in *Deliverable B1.2* (see Figure 1). Finally, we conclude in Section 4, summarizing the TCCS support for QoS analysis, and discussing our work in this area for the next year.

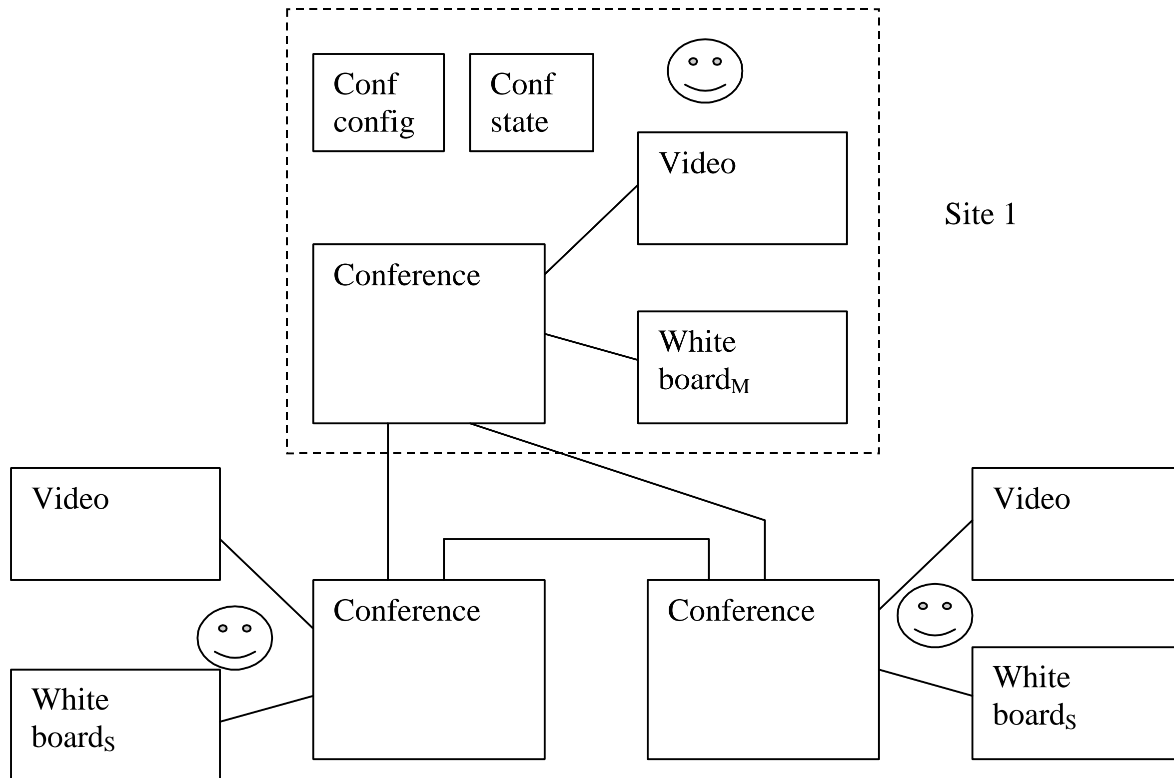


Figure 1: Video-conference system

2. QoS Analysis with respect to Middleware Customization

The ASTER toolset for the systematic customization of middleware according to the non-functional requirements of applications has been extended so as to be independent of any specific middleware platform, hence easing its integration within the TCCS environment while being extendible with respect to future evolution of the environment [Zarras 2000]. The resulting support for middleware customization is described in the following subsection. Subsection 2.2 then concentrates on the integration of the ASTER toolset within the TCCS environment.

2.1. Approach to middleware customization

The systematic middleware customization process comprises: (1) designing a concrete middleware architecture that refines some abstract non-functional requirements of a given application, (2) building an implementation of the resulting concrete middleware architecture that is made out of reusable middleware services. This subsection details the basic concepts towards easing the aforementioned steps. In order to give a clear view of what is the ultimate

goal of middleware customization, this subsection first defines what is considered as middleware architecture, and further discusses the refinement relation defined over middleware architectures, based on previous work done in the software architecture field. Following, the structure and contents of a middleware repository are presented, together with the basic steps that constitute a method for the systematic retrieval of a middleware architecture that refines some abstract application requirements. Finally, this subsection addresses a method for assembling and integrating a middleware implementation within an application, given the architectural description of the application, and the architectural description of a concrete middleware architecture that meets the application requirements.

Middleware architecture

The goal is to design and implement a concrete middleware architecture that enables application components to interact in a way that satisfies their requirements. Before giving details regarding the proposed middleware customization process, it is necessary to give a clearer view on what is a concrete middleware architecture and what does it mean to refine abstract requirements of an application into a concrete middleware architecture. A concrete middleware architecture comprises components, whose implementations correspond to client and server side proxies. Client and server side proxies combine functionalities provided by a broker so as to enable application components to interact in a way that satisfies their requirements. Bindings between client and server side proxies are realized through a more primitive connector like a primitive TCP/IP connector, or a primitive RPC connector. Except for the proxies, a concrete middleware architecture may comprise components that correspond to services provided by the underlying middleware infrastructure (e.g. security, transactions, etc). Middleware services can be used explicitly by the application, meaning that there exist client-side and server-side proxies, which allow application components to interact with the service. Similarly, middleware services can be used explicitly by other middleware services. In this case, one service plays the role of a client, requiring an interface that is provided by the other service, which plays the role of the server. Again, this interaction is realized through client and server side proxies. Functionalities provided by middleware services can be combined within a proxy together with functionalities provided by the broker so as to allow application components to interact in a way that satisfies the application requirements. For example, there exist middleware infrastructures such as ORBIX and MICO, which allow executing operations within the client-side proxies, just before, or right after issuing a request. Moreover, they allow executing operations within the server-side proxies, just before, or right after delivering a request to the server. If however, the previous flexibility is not provided by the underlying middleware infrastructure, the remaining option for combining service functionalities with broker functionalities is with using components that wrap the implementation of a client-side proxy. This kind of components is widely known as *wrappers*. Using wrappers for combining functionalities of a broker with the ones provided by other middleware services, so as to enable application components to interact in a way that satisfies their requirements, keeps the implementation of the client and server side proxies clean. However, it burdens more the developers. From an architectural point of view, a *wrapper* is very similar to a proxy. In our framework, a wrapper consists of a client-side wrapper and a server-side wrapper. In principle, a client-side wrapper is used by a client component, to issue requests towards a server component. More specifically, the client-side

wrapper forwards requests made by the client to the corresponding server-side wrapper. In addition, the client-side wrapper serves for calling operations provided by middleware services just before or right after forwarding client requests to the server-side wrapper. The server-side wrapper delivers client requests to the client-side proxy of the server component. In addition, the server-side wrapper serves for calling operations provided by middleware services just before or right after delivering a request to the client-side proxy of a server component. Figure 2 gives an example of a concrete middleware architecture that mediates the interaction between application components.

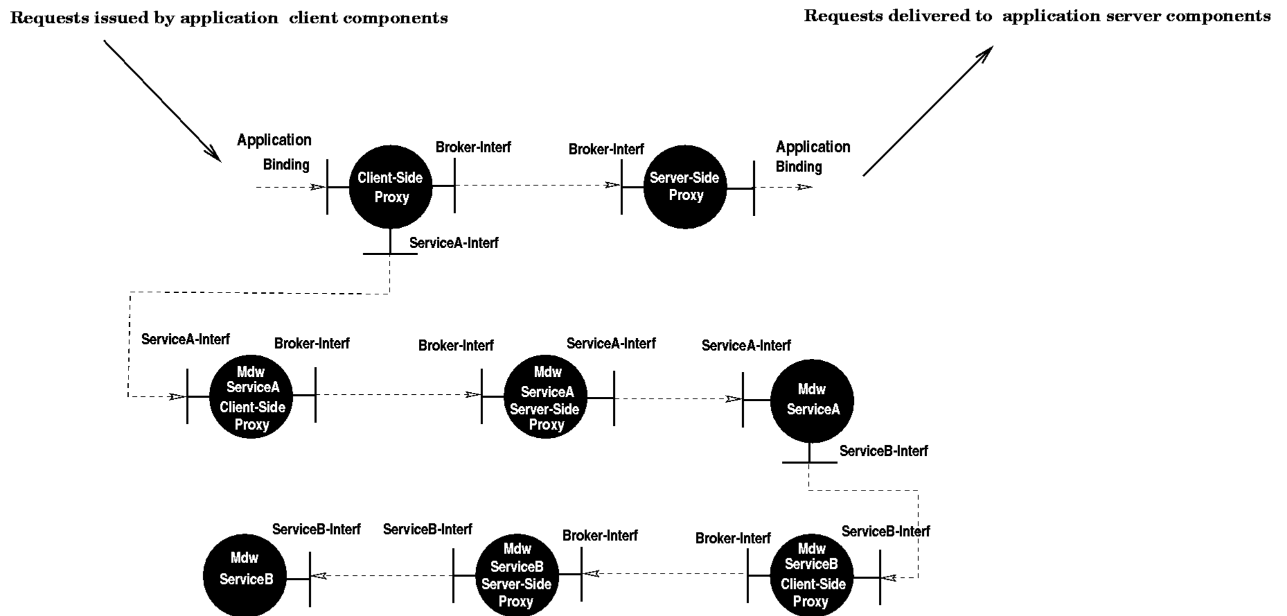


Figure 2: A concrete middleware architecture

Refining an architecture into a more concrete one means to precise on the structure and properties of the elements that constitute the architecture. Hence, during an individual refinement step the architect may decompose an element of the abstract architecture into a configuration of more concrete architectural elements. To verify the correctness of this step, the architect must prove that the configuration of concrete elements provides the properties of the abstract element. The properties of a configuration are a combination of the properties provided by the individual concrete elements. This combination can be derived based on the way the elements are connected. We do not provide here the formal definition associated with correct architecture refinement, which relies on the formal specification of QoS properties using linear temporal logic; the interested reader is referred to [Zarras 2000] for detail. Basically, a middleware architecture *M* refines an architecture *M'* if the properties provided by the former logically imply the ones provided by the latter. The basic inspiration for this approach comes from the work presented in [Mili *et al.* 1997], which was used to define the circumstances under which a concrete fraction of software refines another concrete fraction of software. The ultimate goal of the approach described in [Mili *et al.* 1997] was to structure a repository of available software in a way that enables the efficient retrieval of software. Given the remarks regarding middleware architecture and middleware architecture refinement

it is now possible to proceed with the basic steps that constitute the systematic customization of middleware.

Middleware retrieval

Given the refinement relation over middleware architectures, and in order to simplify the work of both the architect and the designer while trying to refine abstract requirements of an application into a concrete middleware architecture that satisfies them, the systematic customization of middleware relies on an organized repository of middleware architectures. The repository can be either systematically traced using theorem proving technology or browsed, by the architect for retrieving a concrete middleware architecture that refines the requirements of an application. Moreover, the repository can be used by the designer for retrieving implementations of concrete middleware architectures, or implementations of architectural elements that can be used to assemble the implementation of a concrete middleware architecture.

The middleware repository is organized in two parts:

- A *design repository*, that contains the history of all the refinement steps that were performed by the architect, while developing middleware for applications that were built at some time in the past.
- An *implementation repository*, that contains available implementations of :
  - Middleware components, implementing services provided by an existing infrastructure.
  - Concrete middleware architectures that mediate the interaction between application components with respect to certain properties.

Using file system terminology, the design repository is a hierarchy of directories. Every directory in the hierarchy contains information for an architecture that mediates the interaction between application components. More specifically, this information comprises two files. The first file stores a textual description of the middleware architecture, described using the ASTER ADL. The second file contains a formal specification of the properties provided by the middleware architecture. In practice, a formal specification of middleware properties is given as a STeP theory [Bjorner *et al.* 1998]. Except for these two files, a directory may contain links to available implementations of the corresponding middleware architecture. Those links lead to the implementation repository, which is described right after. Finally, sub-directories store information for middleware architectures that refine the one described by the parent directory. Consequently, the whole directory hierarchy reflects a history of refinement steps that were performed by the architect during the design of middleware for applications, developed at some time in the past. The root of the directory hierarchy contains information regarding a middleware architecture that provides unreliable RPC communication. Figure 3, gives an abstract view of a (simplified) middleware repository. In particular, the left side of the figure depicts an abstract view of the design repository.

The implementation repository contains implementations of available middleware architectures and implementations of available middleware components. Its purpose is to help the designer when trying to retrieve an implementation of a concrete middleware architecture handed to him by the architect, or when trying to retrieve implementations of middleware components that can be used to construct a concrete middleware architecture that is not yet implemented. Hence, the designer searches the implementation repository for something really specific, like a CORBA-based architecture, or a CORBA service. For that reason, the implementation repository is divided into a number of sub-directories, each one of which stores software based on a specific middleware infrastructure. For example we can imagine that the implementation repository contains a CORBA directory, a DCOM directory and an EJB directory. The CORBA directory includes a number of sub-directories, each one of which is specific to a particular CORBA compliant infrastructure (e.g. MICO, ORBIX, TCCS, etc.). Every infrastructure-specific directory contains the implementation of the broker, and tools that ease the development on top of the particular infrastructure (e.g. IDL compilers, etc.). Moreover, every infrastructure-specific directory is divided into two sub-directories. One of them stores information for components that implement services provided by the infrastructure. The other contains implementations of middleware architectures built out of those services. In particular, the sub-directory of services itself embeds a number of sub-directories, each one of which is specific to a particular service. Each service-specific directory contains a file with the ADL description of the service, a file that stores the formal specification of the properties provided by the service, and finally the code that implements the service. The directory of architectures stores different implementations of available middleware architectures into different directories, each one of which, has also links to directories in the design repository that store the architectural information related to each implementation. The right side of Figure 3 gives an abstract view of the implementation repository.

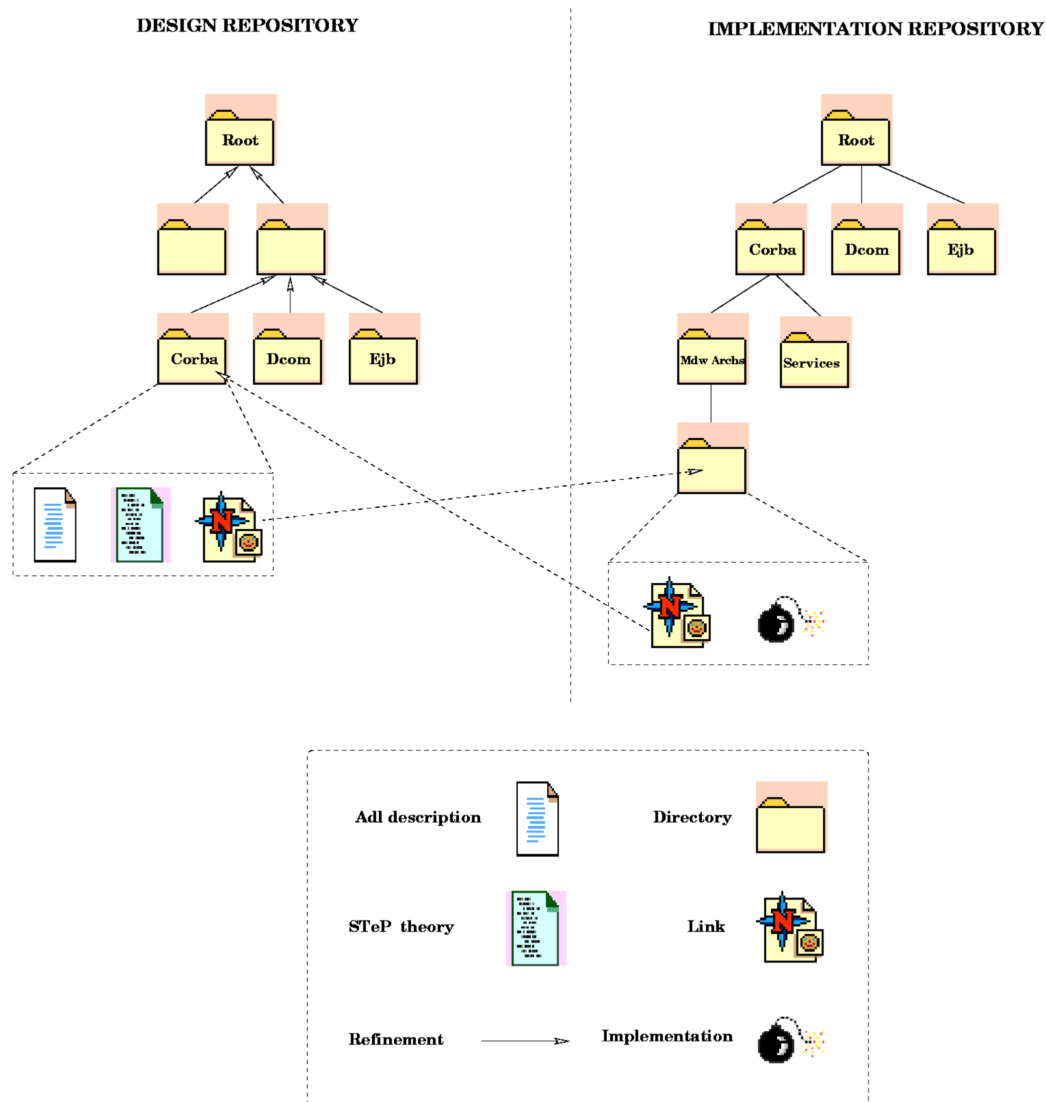


Figure 3: The middleware repository

Given the design repository, the architect is provided with an organized structure that contains all previous attempts to refine abstract requirements of applications into concrete middleware architectures. This knowledge can be a useful assistance towards future attempts for refining abstract application requirements. However, the size of the design repository is expected to be large and hence, a simple and efficient method for navigating through the repository is needed. As mentioned, the goal of the architect is to refine abstract requirements into a concrete middleware architecture. Consequently, the architect should be provided with a method that allows him to easily locate refinement paths leading to concrete middleware architectures that satisfy the abstract requirements of an application. One simple approach for locating such paths is to start searching from the root directory for middleware architectures that refine the abstract requirements. However, this involves proving that the properties of a middleware architecture imply the requirements of the application, which

requires using theorem proving technology. As it is widely known, theorem provers are not the easiest tools to cope with, and certainly they are not user friendly, neither efficient. A method inspired from the one proposed in [Schumman & Fischer 1997] is proposed for reducing the use of a theorem prover while trying to systematically locate paths in the design repository that lead to concrete middleware architectures, which refine the requirements of an application. More specifically, a middleware architecture may refine application requirements if the description of its properties is based on the architectural style used to describe the application requirements, or an extension of that style. In our framework, architectural styles relating to middleware architectures properties and application requirements are defined as STeP theories, i.e. specifications stating base architectural elements and non-functional properties. A style is then based on (or an extension of) another style if it includes (using the STeP `include` statement) a theory defining the latter style. Based on those remarks, locating abstract middleware architectures that are gradually refined into concrete middleware architectures, which satisfy requirements of an application, comprises two steps:

- Locating abstract middleware architectures whose properties are described using, at least, the same style as the one used for describing application requirements (i.e. the theory defining the style is included within the specification of both the application requirements and the abstract middleware architecture).

- Then, starting from those middleware architectures, try to locate one that refines the application requirements based on the definition for correct architecture refinement.

At this point it should be mentioned that if either of the previous steps fails to return a result, then there exists no middleware architecture within the design repository that may be refined into a concrete middleware architecture that satisfies the application requirements. Consequently, the architect is obliged to design the concrete middleware architecture from scratch, possibly re-using middleware architectures enforcing weaker properties than those required. During the design process, the architect typically performs several subsequent refinement steps resulting in middleware architectures that must be stored within the repository in a way that preserves the refinement relation.

Following a typical software development process, the application designer gets as input from the architect, at least, one directory that stores information regarding a concrete middleware architecture. The designer's main responsibility is to realize this architecture, possibly taking into account requirements regarding performance, scalability etc. The first thing checked by the designer is whether the directory, handed to him by the architect contains links leading to available implementations of the concrete middleware architecture, stored in the implementation repository. If this is the case, then, he checks whether the available implementation meets his requirements on performance, scalability etc. Again if this is the case, the designer's work is finished. Then, it is the developer that takes over integrating the available implementation within the application. It must be noted at this point that the systematic customization of middleware does not provide, so far, any help to the designer, when trying to verify properties like scalability and performance, against application requirements. Getting back to the designer's responsibilities, if there exists no available implementation, or if the implementation does not meet the requirements on performance,

scalability etc., the designer is obliged to design the implementation from scratch. This process involves designing the implementation of the individual architectural elements that make up the architecture. However, it is possible that the implementation repository contains implementations of the individual architectural elements that make up the architecture. Hence, before starting to design the individual architectural elements from scratch the designer can look up in the repository for those elements. Since the designer is provided with the description of the concrete middleware architecture, he is aware of the properties that characterize the behavior of the individual elements that constitute it. Moreover, since the middleware architecture is a concrete one, the designer can reason about whether he is looking for architectural elements that implement, e.g., CORBA, DCOM, or EJB services. Hence, searching in the implementation repository for available implementations that can be reused to construct the implementation of a concrete middleware architecture takes place according to the following steps:

- Go to the directory of services corresponding to the middleware infrastructure that the concrete middleware architecture relies on.
- For every middleware component required for constructing the concrete middleware architecture, except the ones describing application proxies, do:
  - Locate middleware components whose properties are described using, at least, the same style with the one used for describing the properties of the required middleware component.
  - Starting from middleware components resulting from the previous step select the ones whose properties imply the ones of the required component. In other words, select middleware components that refine the one required for constructing the concrete middleware architecture.

Given the retrieved implementations of the elements that can be used to construct the implementation of a concrete middleware architecture, the developer takes over for actually building the implementation. Finally, it must be noted that once a new implementation of a concrete middleware architecture is built, the designer inserts it in the implementation repository.

Middleware integration

The integration of a middleware implementation that realizes a concrete middleware architecture comprises assembling the implementations of the individual elements that make up the concrete middleware architecture, and combining the resulting implementation with the application. This work is typically performed by the application developers based on the input they get from designers. In the best case, where an implementation of a concrete middleware architecture was found in the implementation repository, the developer has to combine it with the application. In the worst case, where no implementation was found, the developer has to build the concrete middleware architecture using existing middleware services that implement components that constitute the architecture. This paragraph proposes a systematic method for assembling the implementation of a middleware architecture that can be easily combined with any given application.

Let us take a closer look at the developer's work. Taking the worst case mentioned above, suppose that the implementation repository does not contain the implementation of a concrete middleware architecture. Then, the input from the designer comprises a number of components implementing middleware services provided by a middleware infrastructure. Then, the developer has to perform the following tasks:

- Build client-side and server-side proxies that realize explicit connections between middleware services.
- Build client-side and server-side proxies that combine functionalities of the broker and possibly functionalities of the services, so that application components interact in a way that satisfies their requirements.
- Build wrappers that wrap client-side proxies and combine functionalities provided by middleware services, so as to enable application components to interact in a way that satisfies their requirements.

Let us now examine each one of the aforementioned tasks. If two middleware components, implementing services provided by an infrastructure are explicitly connected, the developer has to build corresponding client and server side proxies that enable the interaction between the two services. Most of the existing middleware infrastructures provide tools which, given a specific input from the developer, generate client and server side proxy implementations automatically. Hence, the developer's work amounts to producing input for the aforementioned tools. In the absence of a tool that automatically generates client and server side proxies the developer has to produce their implementation manually. Nevertheless, building connections among services is performed only once, when assembling for the first time the implementation of a concrete middleware architecture. Then, every time this implementation is reused in the context of a different application, the developer's work is trivial.

Briefly summarizing what was stated in the definition of middleware architectures, a wrapper to a client-side proxy is divided in two parts, the client-side wrapper and the server-side wrapper. The interfaces provided, resp. required, by the previous two elements depend on the application. Moreover, the implementation of the server-side wrapper depends on the application. Hence, when building the implementation of a concrete middleware architecture for the first time, the developer has to build the application-dependent parts. Moreover, whenever reusing the implementation of a concrete middleware architecture within the context of another application, the developer has to rebuild the application-dependent parts. Fortunately, the application-dependent parts of the middleware do not relate to the application implementation; instead they only relate to the architecture of the application (e.g. component types, interfaces, etc). Given the above remarks and in order to ease the work of the developers, we propose a systematic method for building application-dependent parts of a middleware implementation that can be easily reused.

The basic idea behind the proposed method is to provide the developer with a simple language, which enables him to describe abstractly, i.e. independently from the specific application, the process of building the application-dependent parts of the middleware. The description of this process is specified at the time when the implementation of a middleware

architecture is built for the first time. This process description, together with a code generator that additionally takes as input information coming from the ADL description of a particular application, are used to generate the application-dependent parts of the middleware, every-time this middleware is reused for a different application.

To combine a middleware implementation with an application the developer has to either implement client and server side proxies that enable interaction among application components, or he has to provide input to infrastructure-specific tools that generate the client and server side proxy implementations. The proxy interfaces and implementations are application-dependent. Consequently, the input to the tools that generate them automatically is application-dependent. Again, the application-dependent parts of the proxy implementations relate only to the architecture of the application (e.g. component types, interfaces etc.).

Based on the idea introduced above, it is possible to simplify the process of implementing, or generating proxy implementations. In the absence of a tool that automatically generates proxy implementations, the developer uses the proposed language to abstractly describe the process of implementing middleware proxies for application components. In the presence of tools that automatically generate proxy implementations the proposed language can be used for describing the process of producing input to those tools. Then, every-time the middleware architecture is reused for mediating the interaction among components of an application the process description is fed to a code generator that generates the proxies, or input to the tools that generate proxies, based on information coming from the ADL description of the particular application.

Summarizing so far, we propose using a simple language to describe a *process* that must be followed to develop the application-specific parts of a middleware (see [Zarras 2000] for detail). In general, such a language must enable the developer to describe a process that iterates through the basic architectural elements that constitute the application (e.g. components, configurations etc.), or a process that iterates through basic features (e.g. interfaces, operations etc.) that recursively characterize those basic architectural elements and describe source code that must be generated for each one of those. Hence, the *object* of a process that describes the way to develop the application-specific parts of the middleware is either a basic architectural element, or a feature that recursively characterizes a basic architectural element. Moreover, the process language must provide means that enable the developer to describe a process that tests whether or not certain process objects are present in the architectural description of the application, and depending on the case produce a fraction of source code. For example, the developer may want to describe a process, which tests whether or not an operation has a return value. Depending on the case, the previous process generates, or not the declaration of a variable used to store the return value. The developer may also want to describe a process, which checks whether or not a component type is composite, and depending on the case generates the corresponding client and server side proxies.

Every object within the ADL description of the application is named. More specifically, component types have names. Same holds for interface types, operations, arguments, configurations etc. Hence, the process language must provide means that allow to describe a process that generates source code, which contains names characterizing a process object.

For example the name of a particular component type described in the ADL description of the application is needed to generate a corresponding CORBA module. Similarly, the name of a particular interface type described in the ADL description of the application is needed to generate a corresponding CORBA IDL interface. For the purpose of this work a minimal extensible process language was developed. The language grammar accepts as input a text stream that corresponds to a fraction of source code, which occasionally contains development process directives. A directive starts always with the $ character and it is enclosed within parentheses. The process directive can be of four different types: an object, an iteration, a condition or a negation.

Figure 4 gives a complete view of the ADL-based toolset for the systematic customization of middleware. More specifically, the middleware retrieval algorithm takes as input a STeP file that describes application requirements. Furthermore, it uses STeP files stored in the design repository to perform the first step of the retrieval. Moreover, STeP files stored in the design repository and the STeP file describing application requirements are input to the STeP theorem prover, which is used to perform the second step of the retrieval. Similarly, the previous algorithm and the STeP theorem prover are used to ease the designer's work. The ADL tool support comprises a code generator that takes as input application-specific ADL information, as extracted by the ADL compiler, and the implementation of a concrete middleware architecture, parts of which are development process directives. Based on the ADL information, the code generator generates the implementation of the application-dependent parts of the concrete middleware architecture.

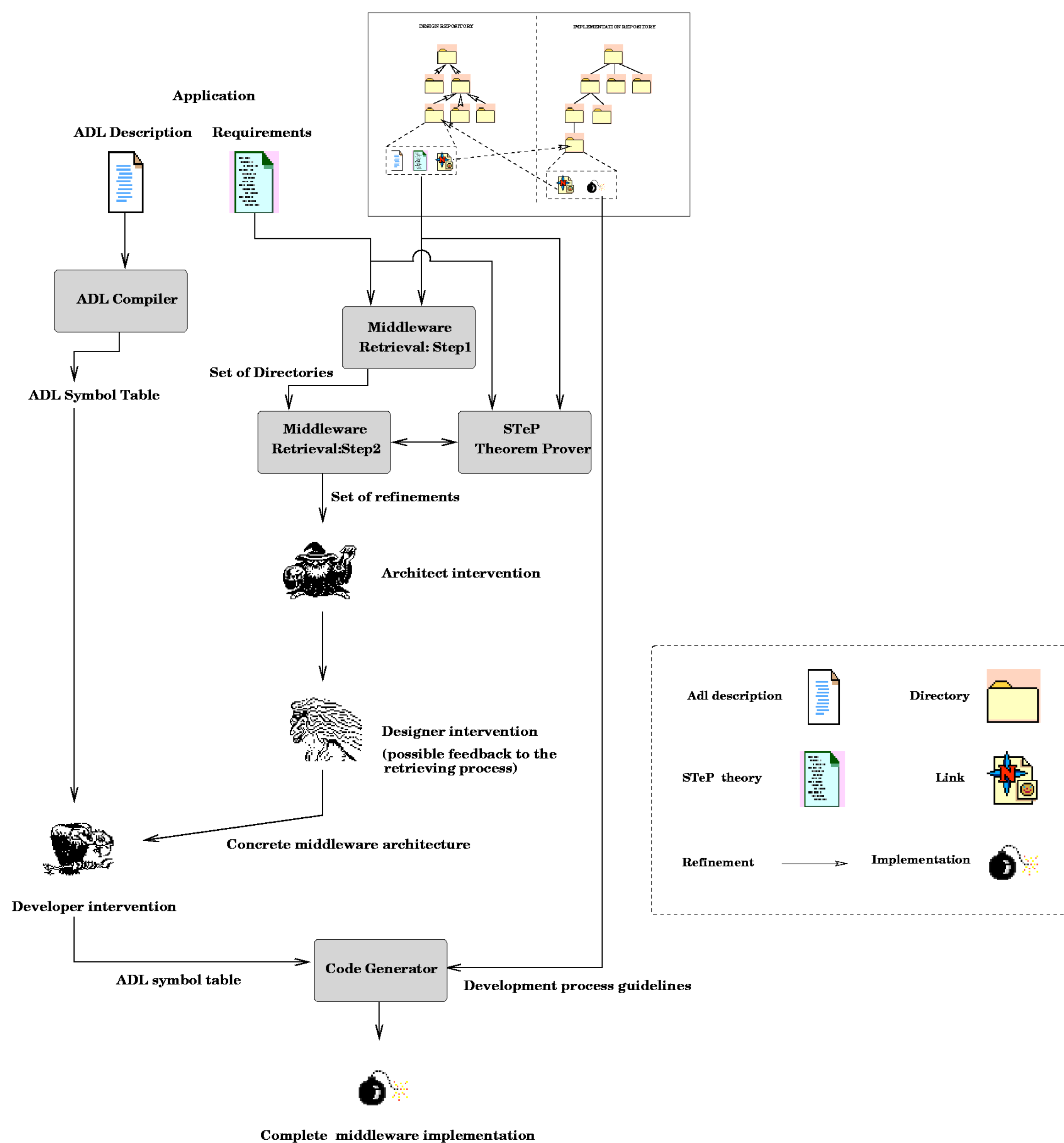


Figure 4: ADL-based toolset for middleware customization

2.2. Middleware customization in the TCCS environment

Let us now concentrate on the integration of the proposed toolset for middleware customization within the TCCS environment. As described, the toolset comprises: the ASTER ADL for architectural descriptions embedding QoS specification, a repository of middleware architectures, and a tool for middleware integration. It further uses an existing

theorem prover (i.e. STeP) for the systematic selection of elements within the repository. Integration of these elements in the TCCS environment amounts to the following:

1. Coupling the ASTER ADL with the TCCS ADL, so as to address the specification of QoS requirements within the structural views of complex services, elaborated using the TCCS environment. This step is trivial to achieve since the TCCS ADL actually embeds the ASTER, DARWIN, and OLAN ADLs. Precisely, the OLAN ADL is used for graphic-based general architecture description, which may be annotated with additional specifications as offered by the three aforementioned ADLs. We further have to address specification of QoS requirements within the description of the behavioral views of complex services; this issue lies in coupling specifics of the ASTER ADL with the workflow language.

2. A repository of middleware architectures based on the capabilities of the TCCS platform, which comprises CORBA, CORBA-based workflow services, and the software agent infrastructure. For middleware customization, we consider only customization with respect to the CORBA infrastructure (including workflow support) since it offers a wide variety of middleware services. Hence, the middleware repository is to be populated with a number of middleware architectures built out of the CORBA-based middleware that may be built using TCCS.

3. Adaptation of the integration tool so as to cope with the specifics of the TCCS infrastructure. The current version of the tool generates C++ code, interfacing with any CORBA-compliant ORB. In the case of the TCCS environment, the tool needs to be extended so as to generate Java code.

The aforementioned adaptations to the ASTER toolset for middleware customization are currently under development. Let us point out here that while the general ASTER approach to middleware customization relies on the formal specification of QoS properties using linear temporal logic, we have taken a more pragmatic approach in the context of TCCS. Properties are specified using names and hence integration and retrieval of middleware architectures within the repository relies on user selection through browsing of the repository, instead of using theorem proving technology. We have undertaken such an approach due to the fact that the first prototype of the TCCS platform is not expected to embed a large number of middleware services.

In the following, we further describe middleware customization in the TCCS environment, addressing customization of middleware with respect to complex services' structural and behavioral views, respectively. We use here the example of the video-conference system for illustration.

Middleware customization with respect to structural views of complex services

In the systematic customization of middleware, we employ the basic concepts introduced in the software architecture paradigm. The input of the systematic customization is an architecture description of the application that includes a specification of the requirements for connectors, i.e. the middleware that mediates the interactions among the components. We also rely on the recursive nature of the architecture description to specify how the connectors

are built from the middleware services. For illustration, we give below the ASTER architecture description of the video-conference system to demonstrate the systematic customization of middleware. Let us notice here that the mapping of the provided description onto the graphical OLAN-based architecture description is straightforward. It amounts to attribute the architectural elements with the corresponding ASTER declarations.

```
// Adl Description in ASTER
// Basic Interface Defs
interface updateState {details on operations' signatures are left out};
interface updateConf {details on operations' signatures are left out};
interface updateMedia {details on operations' signatures are left out};
// Basic Component Defs
component StateView {
      provides localStateUpd : updateState;
      requires localMediaUpd : updateMedia;};
component ConfigView {
      provides localUpd : updateConf;
      requires localStateUpd : updateState, groupUpd[*] : updateConf; };
component MediaView {
      provides localMediaUpd : updateMedia;};
connector WF {
      requires properties ''Workflow'';};
composite component ConfMember {
      instances
            imedia  : MediaView, istate  : StateView, iconfig : ConfigView;
            TCCS    : WF
      requires
            iconfig.groupUpd[*] : updateConf;
      provides
            iconfig.localUpd : updateConf;
      binds // required to provided through connector
            iconfig.localStateUpd istate.localStateUpd through TCCS;
            istate.localMediaUpd imedia.localMediaUpd through TCCS;};
composite component Conference {
      instances
            members[*] : ConfMember, TCCS : WF;
      binds // required to provided through connector
            member[i = 1 to *].groupUpd[j = 1 to *, j != i]
              member[j].localUpd; through TCCS ;};
```

In the above declarations, the `WF` definition requires the connector to provides the ***Workflow*** property, i.e., interaction among the `Conference` components will be achieved through a workflow process.

For illustration, Figure 5 depicts part of the middleware repository associated to the TCCS CORBA-based platform, i.e. CORBA services including support for workflow execution, which relies on transactional services. This repository is then searched when analyzing the video-conference control architecture towards customization of the middleware providing the ***`Workflow`*** property. Once all the middleware services are retrieved, they are assembled according to the middleware architecture associated to the ***`Workflow`*** property, leading to the middleware architecture that is roughly depicted in Figure 6. To assemble the middleware services according to the middleware architecture description, we require the services to export operations defined in the architecture description. The services are then connected together with a binding code generated in a straightforward manner from the architecture description.

We have illustrated middleware customization appertained to the structural views of complex services using requirements for the workflow property, whose support constitutes one of the major features of the TCCS platform. In general, required properties will be of various types since they relate to properties required for the development of base services to be further composed using a workflow schema. Hence, while a workflow-enabled middleware will be required by the base services that are composed within a complex service, internal implementation of those base services may rely on any kind of TCCS-enabled middleware.

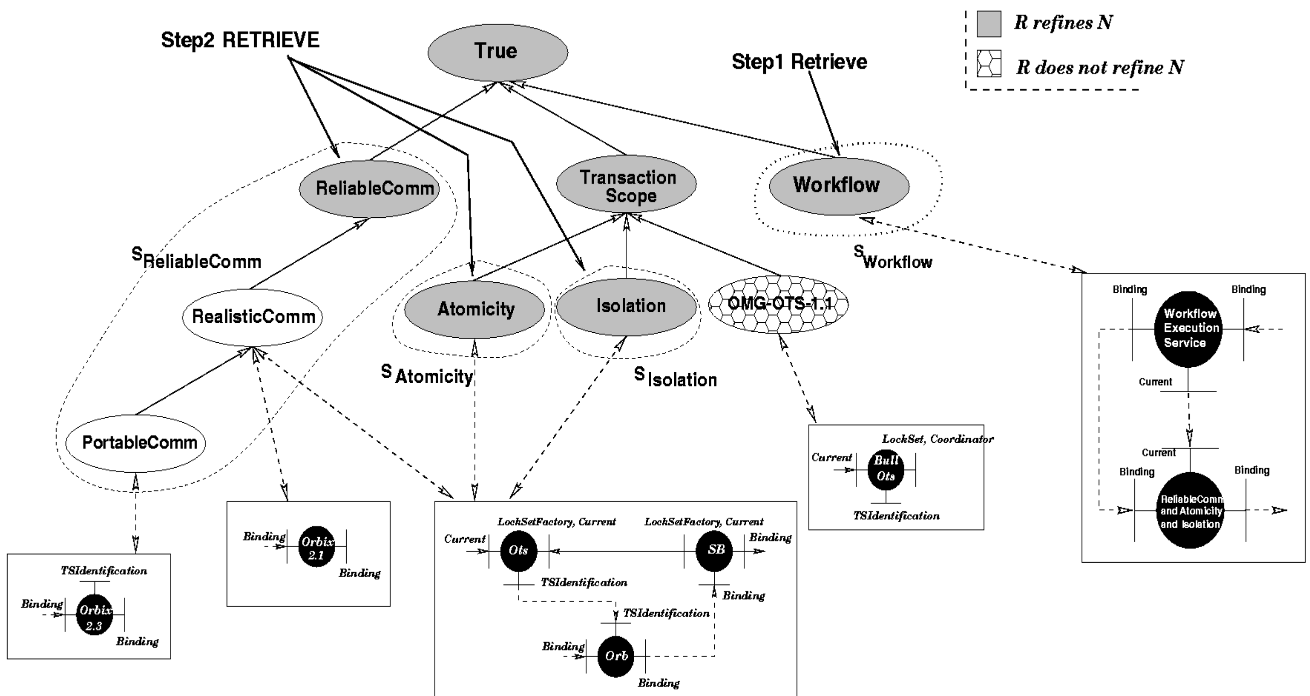


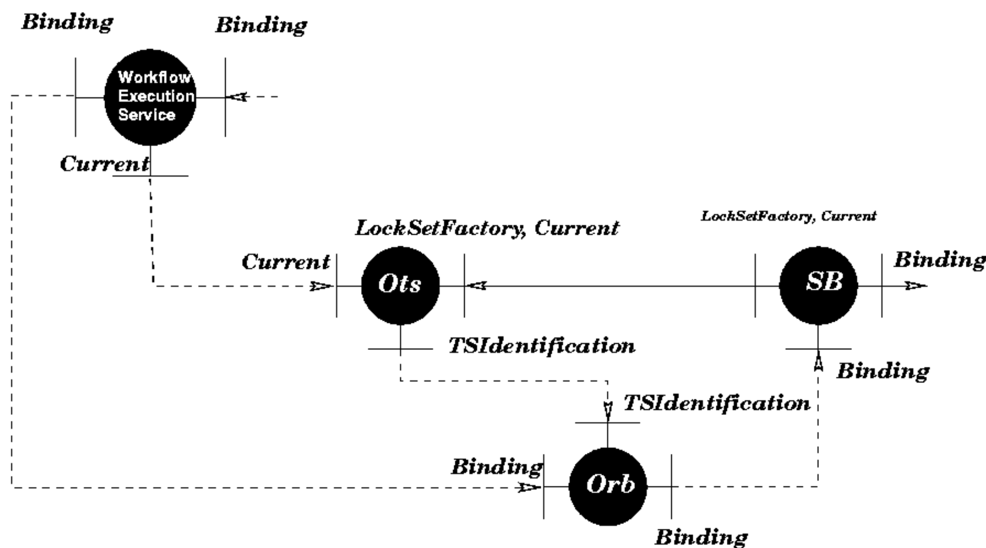
Figure 5: Retrieval process for the `Workflow` property

Figure 6: Middleware architecture providing the `Workflow` property

Middleware customization with respect to behavioral views of complex services

In addition to enabling workflow execution among base services composing a complex service as addressed in the previous paragraph, the workflow execution may itself be dependent upon the provision of a number of QoS properties. Specifically, the workflow management system of TCCS relies on the implementation of *task controllers*, one per workflow task, which serve coordinating the workflow execution and interact using the TCCS platform. By default, task controllers interact using a transactional CORBA-based middleware, which is composed of the TCCS CORBA ORB and the TCCS Object Transaction Service (see Figure 6). However, additional QoS properties may be imposed over this middleware. In particular, considering that workflow tasks may relate to base services provided by distinct organizations, enforcing security properties is among obvious customization for the middleware underlying workflow execution.

Given the proposed approach to middleware customization, the only issue that needs to be addressed lies in specifying QoS requirements for the workflow execution, which will serve for customizing the corresponding middleware. Two approaches may be considered: (i)



specializing the workflow property as specified in the structural view of the complex service, which leads to handle this property in a specific way, (ii) annotating the service's behavioral view (or workflow schema) with the required QoS properties. We prefer the second approach as motivated hereafter. First, although convenient for illustration, there is actually no need for specifying requirements for workflow execution within the service's structural view since complex service provisioning in the TCCS environment lies in the complementary specification of the service's structural and behavioral views where the latter corresponds to a workflow schema. Hence, the customization of a workflow-enabled middleware is implicit, and there is no need for requiring corresponding specification within the service's structural

view. More importantly, the middleware underlying workflow execution specifically relates to the service's behavioral view. Hence, corresponding QoS requirements must be attached to this view.

3. QoS Analysis with respect to Temporal Properties

In addition to the enforcement of QoS criteria through adequate customization of the middleware underlying the execution of complex services, the non-functional properties offered by a complex service further depend upon the properties provided by the elements composing the service. For instance, in the case of a multimedia service, the end-to-end quality of service that is provided depends upon the temporal behavior of the service components (which may be based upon assumptions about the underlying execution environment). In the same way, the performance of a distributed service depends upon how components interact (e.g. minimizing the volume of data transferred over the network). In the framework of C3DS, we are devising a method and associated tools for the temporal analysis of complex services. In a first step, we have been concentrating on the temporal analysis of multimedia services because they are highly demanding in terms of temporal constraints and constitute an increasingly prominent type of services. We are now generalizing our solution to complex services that are primarily targeted by the TCCS environment. An overview of our solution is provided hereafter. Its integration within the TCCS environment is then discussed, addressing ongoing work relating to the temporal analysis of the complex services that are provisioned.

3.1. Approach to temporal analysis

Temporal analysis of a software system may be motivated by different factors. In the case of a software system whose correctness depends upon meeting real-time constraints, temporal analysis is advisory (and even mandatory in the case of a safety critical system) in a way similar to behavior analysis with respect to the system's functional properties. Temporal analysis may further be beneficial for other kinds of systems since it may serve various purposes (e.g., configuring the overall execution environment, detecting the occurrence of failure at run-time). In this subsection, we focus on analysis support for software systems having real-time constraints.

Services having real-time constraints may be subdivided into two categories depending on whether those constraints are *soft* or *hard*. Services from the former category are qualified as *soft real-time services*, and allow missing deadlines. Services from the latter category are qualified as *hard real-time services*, and cannot accommodate missing deadlines. Both kinds of systems impose specific requirements upon the underlying runtime environment with respect to resource management (e.g. real-time scheduling protocol). Such requirements are not accounted for by the TCCS platform. However, this does not imply that the development of soft-real time systems may not be envisioned in the TCCS environment since there is ongoing work on providing CORBA-based platforms for such systems. Hence, in the framework of C3DS, we concentrate on providing support for the analysis of real-time services based on their architectural description, assuming availability of an adequate

underlying platform. In addition, we specifically focus on multimedia systems, which constitute the most prominent set of soft real-time systems.

Analyzing soft real-time systems based on their architectural description

Existing environments providing support for the design of applications based on their architectural description lack expressiveness for two issues raised by multimedia systems: the compatibility of protocols used for transferring data streams (e.g. specialization), and the assessment of the application's temporal behavior. The second issue has been previously addressed in the literature. However, provided solutions rely on mathematical models that are either too formal or too empirical. In one solution, the temporal behavior of the application is computed only by relying on formal distributions. While this method is easily applicable to a limited set of distributions, it turns out to be a tedious, even impossible task when arbitrary formal distributions need to be composed. A convincing example is given by the determination of the distribution of the maximum of any two arbitrary normal distributions. Another solution introduces a mathematical model, which relies on a simulation of the temporal behavior of the application and thus leads to the opposite problem. This model requires that whenever a group of elements is reused, the temporal verification tool has to simulate the global behavior of the application on a per element basis.

Due to the increasing importance of multimedia applications in the distributed system area, and hence to their relevance to the C3DS project goals, we have started investigating both the specification and the verification of the two aforementioned properties of multimedia applications [Demairy *et al.* 1999]. We have proposed a formal framework providing means to the software architect to specify the protocols and temporal behavior of the basic elements of his application. According to these specifications, the verification process determines the set of protocols that are eligible for carrying out interactions among computing elements based on the data streams that are exchanged. This verification is done in three steps:

1. The first step determines whether the protocols of the application are consistent or not. This step results in identifying the protocols that can be used by each component of the application according to the interaction patterns among the application's components;

2. For each combination of the protocols allowable for the application components and connectors, the temporal behavior of the application is assessed against its temporal constraints. The temporal behavior can either be formally computed or simulated, depending on the knowledge the software architect has of the application. This step results in determining whether the application is consistent or not according to its temporal behavior;

3. The two previous steps assess the correctness of the multimedia application both in terms of protocol usage and timeliness properties. If the application is correct, there exists a non-empty set of protocols guaranteeing its correctness. To configure the application, we choose one of the maximal elements of this set according to a user-satisfaction ordering relationship.

Our method alleviates the work of the software architect by allowing him to verify *a priori* (i.e. without building the application) the feasibility of his application. Furthermore, our method allows a hierarchical description of an application, which permits its progressive development. Consequently, adding a new element to a formerly assessed group of elements amounts to compute the algorithms for an architecture composed of the group taken as an element and the new element.

According to the terminology of the software architecture research domain, the *style* of a multimedia application is the set of rules that permit to determine whether an application can be considered as a correct multimedia application or not. These rules encompass the eligibility of the computing and communication elements according to some criteria, e.g. constraints imposed by the target execution environment. In a multimedia application, the interactions among components via connectors are realized by ports, which are used to either send or receive streams of data frames. We further call *server* any component that produces data frames and *client* any component consuming these frames. Without loss of generality, we assume that the time between any two message transmissions within a stream follows a probabilistic distribution. This restriction allows us to model applications in which data exchange follows a continuous temporal behavior over long periods of time, e.g. VoD and tele-medicine applications. The server of a data stream usually performs on it some coding for transmission (i.e. protocols encoding), performance (i.e. compression) or security reasons (i.e. encryption). At the end point of the stream, the software architect has to ensure that the target client is able to decode the data stream by using a decoding process compatible with the coding one. These transformations are carried out according to specific protocols, whether they concern audio (e.g. PCM), video (e.g. CCIR-601, MPEG-2), or arbitrary bit streams (e.g. TCP/IP). Beyond its ability to handle a specific coding or decoding process, a protocol is characterized by its temporal behavior. In general, the elements of a multimedia application handle several protocols, in order to adapt the quality of their output regarding available resources (e.g. CPU, bandwidth). Thus, when building an application out of existing elements, we need to determine which protocol must be used for each element of the application.

An accurate description of the protocols used by each element of a multimedia application is central to the verification of the application's correctness. From this description, given by the software architect, we first determine the protocols eligible for each element of the application, by considering the constraints that each element implies on the others. Then, we verify among the eligible protocols, which ones guarantee that the application meets its temporal constraints. For determining the eligibility of protocols, we rely on the definition of an ordering relationship among protocols similar to the subtyping notion. This enables identifying the protocol that may consistently be used over a connection between two components, based on the protocols handled by the components and the connector between them. Given the architectural specification of a multimedia application, including the specification of the protocols supported at the ports of the architectural elements, an algorithm computes the protocols that may actually be used at each port so as to have consistent data transmission over the architectural elements. Regarding application correctness with respect to timeliness issues, servers express temporal guarantees on time between (*i*) two successive output of frames on a given port and (*ii*) the input of a frame and the output of the resulting data. This information is mandatory when dependencies exist

between input and output ports within a component. Temporal guarantees can be provided either through the provision of a statistical sample of measurements or through the formal distribution describing the service times, which are usually computed after a statistical analysis of the samples obtained by running the component. Dually, clients express temporal constraints on the distribution of the time between arrivals of frames. These constraints express the temporal behavior that must be exhibited by the application. The temporal behavior of the application is determined by applying a timeliness verification algorithm. This algorithm first composes the temporal behavior of each of the elements of the application either by a formal calculus or by simulation. Formal calculus must be preferred whenever it is possible: for example, the sum of two service times following normal distributions is equivalent to a normal distribution. After composition, the algorithm finally checks whether the temporal constraints expressed by the client components can be met.

Example

Let us illustrate our solution using the video-conference system, which is a multimedia application managing quasi-periodic audio and video streams. These streams must meet soft real-time constraints; i.e. users may stand some deadline misses. The video-conference system is made out of the execution of a number of *conference* components. Each of these components represents a user and is in charge of managing the audio, video and white board sub-systems. From the standpoint of the system's temporal analysis, we consider only the audio and video sub-systems since the white board does not produce continuous data. The *conference* components interact through a multicast protocol, which reduces communication cost. In the following, we first examine the configurations that may be taken by the video-conference system, i.e. the description of the protocols that are used for stream delivery, hence setting the quality levels that may be provided by the components and connectors composing the system. In a second step, we describe the system's temporal constraints and the associated theoretical verification process.

Identifying eligible configurations for the video-conference system

In the following analysis, we consider that the system embeds three participants. The system's architectural description relies on two kinds of elements (see Figure 7): the *conference* components handling data streams (denoted by *I*) and the connector for data stream transmission (denoted by *Coms*). The description is hierarchical; e.g., each component I embeds components for managing the video ($video$ component) and audio streams ($audio$ component). Streams are multicasted, meaning that the resources used for communication are independent of the number of participants. Regarding stream delivery, component *I1* receives the video streams from components *I2* and *I3* via components $VI2$ and $VI3$ [1], and the audio streams via components $AI2$ and $AI3$. The same pattern applies for the other *Conference* components. We further assume that the various components of the

[1] Since base components embed a single port in the example, we use the same name for denoting a component and associated port.

video-conference system may offer different quality levels, using corresponding protocols, for the audio and video as given in Table 1.

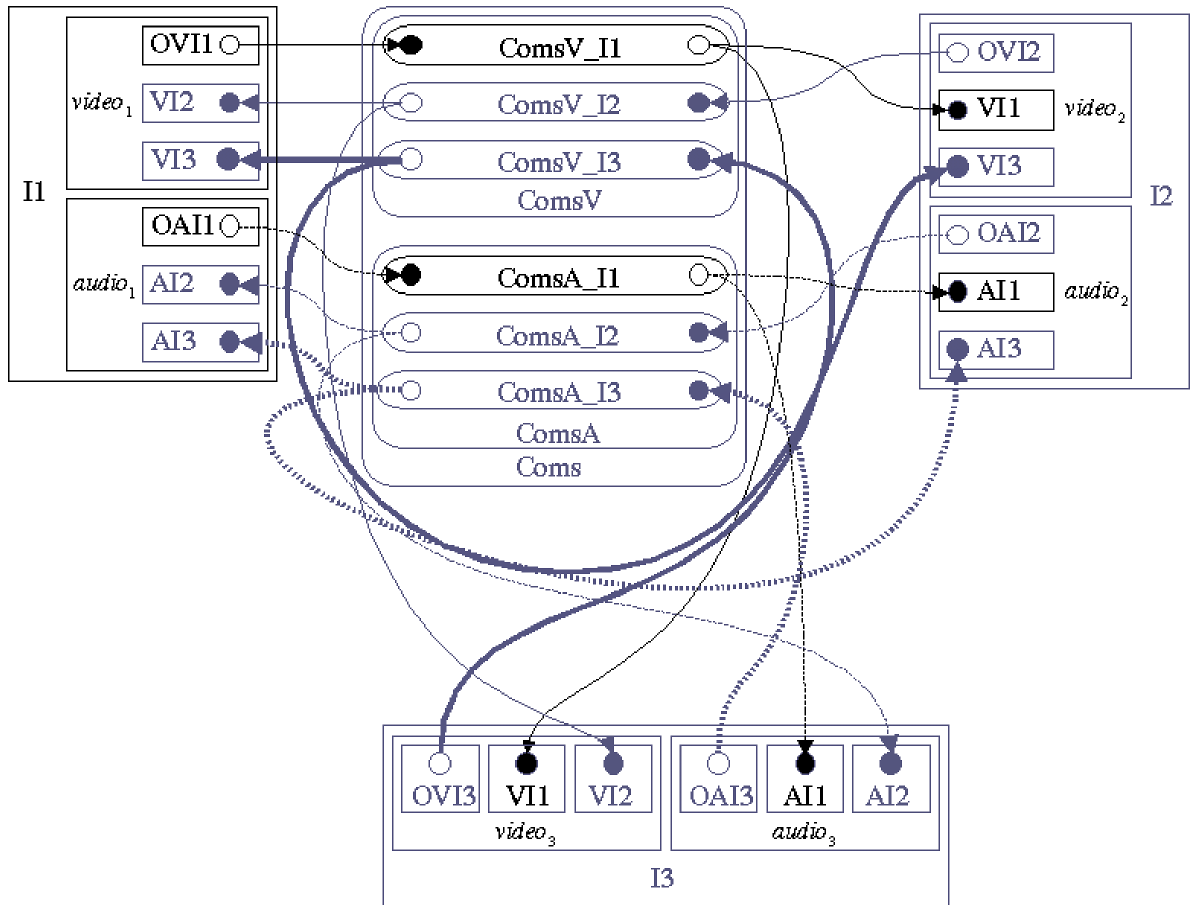


Figure 7: The video-conference system description for temporal analysis

Video	∅	VHS	SVHS	CD
Audio	∅	Telephone	CD	

Table 1: Supported quality levels

Let us now detail the protocol configurations for the overall video-conference system, which set the provided quality level for data streams. We first examine the protocol configurations managed for components. For instance, considering I_1, component OVI_1 may either transmit video streams under formats VHS, $SVHS$, CD, or not transmit any stream ($\varnothing$). Resulting

streams are then to be handled by components VI_2 and VI_3 accordingly. In the same way for the audio stream, component OAI_1 may issue streams according to the quality levels $Telephone$ or CD, or not issue any stream, while components AI_2 and AI_3 should handle the quality level that is being selected. The audio and video sub-systems are closely coupled in that they cannot use any kind of quality levels for streams, independently of each other. Similarly, components that are internal to the video-subsystem are inter-dependent. As an example, let us focus on component I_1, components I_2 and I_3 being handled in a similar way. We assume that the $video_1$ component, which is in charge of managing video streams for I_1, imposes that the output stream and all the input streams should be of the same quality level. We get:

. $Conf_{video_1} = \{(CD \mid CD, CD), (SVHS \mid SVHS, SVHS), (VHS \mid VHS, VHS), (\varnothing \mid \varnothing, \varnothing)\}$,

where $(Q \mid Q, Q)$ represents the states of outport port OVI_1 and input ports VI_2 and VI_3, respectively, for Q belonging to {CD, SVHS, VHS, $\varnothing$}. Since the states of all the ports of $video_1$ are identical, the above definition may be simplified as follows:

$$Conf_{video_1} = \{CD, SVHS, VHS, \varnothing\}$$

In the same way, we get the following specification for $audio_1$:

$$Conf_{audio_1} = \{CD, Telephone, \varnothing\}$$

We further assume that the video-conference system gives higher preference to the audio quality over the video quality. We thus get the following list of protocol configurations for the I_1 component, which are ordered according to the decreasing user preference. Each element of the list gives the states of the *video* and *audi*o sub-systems with respect to the enforced quality levels for streams:

$$Conf_{I_1} = \{(CD, CD), (SVHS, CD), (VHS, CD), (\varnothing, CD), (\varnothing, Téléphone)\}$$

Let us now examine protocol configuration of the *Coms* connector, which handles communication among the system's *Conference* components *I*s. The connector decomposes into two inner connectors for the handling of video and audio streams, respectively. Video communication is carried out through three connectors $ComsV_I1$, $ComsV_I2$ and $ComsV_I3$, which issue video streams from *Conference* components 1, 2, and 3, respectively, to the two other peer components. For communication of audio streams, connectors, $ComsA_I1$, $ComsA_I2$ and $ComsA_I3$ play similar roles. We assume here that all the connectors leave unchanged the quality levels of the stream they transmit, i.e.:

$$Conf_{ComsV_I1} = \left\{(CD \middle| CD), (SVHS \middle| SVHS), (VHS \middle| VHS), (\varnothing \middle| \varnothing)\right\}$$

$$Conf_{ComsV_I2} = Conf_{ComsV_I3} = Conf_{ComsV_I1}$$

$$Conf_{ComsA_I1} = \left\{(CD \middle| CD), (Télépone \middle| Téléphone), (\varnothing \middle| \varnothing)\right\}$$

$Conf_{ComsA_I2} = Conf_{ComsA_I3} = Conf_{ComsA_I1}$

Unlike the system's components, connectors need not be inter-dependent. The possible protocol configurations for the video and audio connectors are then equal to the product of all the configurations that are eligible for each of their composing connectors:

$$
\begin{aligned}
Conf_{ComsV} = \; & \{((CD \mid CD),(CD \mid CD),(CD \mid CD)),\\
& ((SVHS \mid SVHS),(CD \mid CD),(CD \mid CD))\\
& ((VHS \mid VHS),(CD \mid CD),(CD \mid CD))\\
& \ldots\\
& ((\emptyset \mid \emptyset),(\emptyset \mid \emptyset),(VHS \mid VHS))\\
& ((\emptyset \mid \emptyset),(\emptyset \mid \emptyset),(\emptyset \mid \emptyset))\}
\end{aligned}
$$

$$
\begin{aligned}
Conf_{ComsA} = \; & \{((CD \mid CD),(CD \mid CD)),\\
& ((Téléphone \mid Téléphone),(CD \mid CD))\\
& ((\emptyset \mid \emptyset),(CD \mid CD))\\
& \ldots\\
& ((\emptyset \mid \emptyset),(Téléphone \mid Téléphone))\\
& ((\emptyset \mid \emptyset),(\emptyset \mid \emptyset))\}
\end{aligned}
$$

Then, the protocol configurations for the *Coms* connector are equal to the product of the configurations supported by its composing connectors, i.e. $Conf_{Coms} = Conf_{ComsV} \times Conf_{ComsA}$:

$$
\begin{aligned}
Conf_{Coms} = \; & \{(((CD \mid CD),(CD \mid CD),(CD \mid CD)),((CD \mid CD),(CD \mid CD))),\\
& (((SVHS \mid SVHS),(CD \mid CD),(CD \mid CD)),((CD \mid CD),(CD \mid CD))),\\
& (((VHS \mid VHS),(CD \mid CD),(CD \mid CD)),((CD \mid CD),(CD \mid CD))),\\
& \ldots\\
& (((\emptyset \mid \emptyset),(\emptyset \mid \emptyset),(\emptyset \mid \emptyset)),((\emptyset \mid \emptyset),(Téléphone \mid Téléphone))),\\
& (((\emptyset \mid \emptyset),(\emptyset \mid \emptyset),(\emptyset \mid \emptyset)),((\emptyset \mid \emptyset),(\emptyset \mid \emptyset)))\}
\end{aligned}
$$

Using the algorithm of [Demairy *et al*. 1999] and given the possible protocol configurations for the video-conference system's components and connectors, we get that the following configurations may be chosen for the overall system:

$$
\begin{aligned}
Conf_{appli} = \; & \{((CD,CD),(CD,CD),(CD,CD),(CD,CD)),\\
& ((SVHS,CD),(SVHS,CD),(SVHS,CD),(SVHS,CD)),\\
& ((VHS,CD),(VHS,CD),(VHS,CD),(VHS,CD)),\\
& ((\emptyset,CD),(\emptyset,CD),(\emptyset,CD),(\emptyset,CD)),\\
& ((\emptyset,Téléphone),(\emptyset,Téléphone),(\emptyset,Téléphone),(\emptyset,Téléphone)),\\
& ((\emptyset,\emptyset),(\emptyset,\emptyset),(\emptyset,\emptyset),(\emptyset,\emptyset))\}
\end{aligned}
$$

where the elements of $Conf_{appli}$ give the protocol configurations for the elements composing the system according to the following pattern: $(I1, I2, I3, Coms)$.

Each protocol configuration is associated with the corresponding resource usage. Hence we may remove from $Conf_{appli}$, all the configurations that use more resources than those provided by the targeted execution environment. The temporal behavior of remaining configurations must then be checked with respect to the user requirements. Since those configurations are ordered according to the decreasing level of provided quality, the verification algorithm may actually be sequentially applied, starting from the first eligible configuration and iterating over the list of eligible configurations until there is one that meets the temporal constraints that are set for the system.

Checking temporal properties

We identify three kinds of temporal constraints for a multimedia application: end-to-end constraints, constraints relating to streaming, and those appertained to the synchronization of streams. Since, we are considering soft real-time systems, the deadlines as prescribed by the temporal constraints may be missed as long as the probability of missing deadlines does not exceed a given threshold, which is set by the system's developer. For instance, if we assume that the temporal constraints associated with the handling of video streams must be met with a probability of $1-10^{-4}$, this means that timing errors are tolerated at most every 7 minutes, for a refreshing rate of 24Hz. For every *Conference* component of the video-conference system, we set two constraints over the end-to-end delay between the time the image is captured at one component and the time it is displayed at peer components. In general, it is considered that a delay of 42ms (which corresponds to a refreshing rate of 24 Hz) is the maximum delay that may be tolerated by users. We get the following end-to-end temporal constraint for video streams:

$$P[\delta_{VI} - \delta_{OVI} \leq 42] \geq 1 - 10^{-4}$$

where δ_{VI} denotes the time at which a frame is delivered at component VI of one of the two destination components and δ_{OVI} denotes the time at which the frame was issued by the sending component, given that the delay for the frame transmission must not exceed 42ms. We further impose that this constraint should be met with a probability of $1-10^{-4}$. In addition to the above constraint, images should be delivered on a periodic basis to destination components, as stated below:

$$P[\Delta_{VI} \in [0,40]] = P[T_{OVI} + \Delta(T_{ComsV_I} + T_{VI})] \geq 1 - 10^{-4}$$

As for the handling of video, we identify two end-to-end constraints for audio. Since there are several audio frames for a single image that is displayed, audio frames are grouped so that an image corresponds to a group of audio frames. We may further set stronger constraints over audio delivery since this is considered as more important than the one of video. We get:

$$P[\delta_{AI} - \delta_{OAI} \leq 42] \geq 1 - 10^{-5}$$

$$P[\Delta_{AI} \in [0,40]] = P[T_{OAI} + \Delta(T_{ComsA_I} + T_{AI})] \geq 1 - 10^{-5}$$

We finally have two temporal constraints relating to the synchronization of the audio and video streams issued to *Conference* components. The delay between the time an image is

displayed and the one at which the corresponding audio is displayed must not exceed 100ms. We get the following two constraints for component *I3*, similar constraints being set for peer components:

$$P[\delta_{AI1} - \delta_{VI1} \leq 100] \geq 1 - 10^{-4}$$

$$P\left[\delta_{AI2} - \delta_{VI2} \leq 100\right] \geq 1 - 10^{-4}$$

where δ_{VI} denotes the time at which a video frame issued by one participant is delivered at component VI of another participant, δ_{AI} denotes the time at which an audio frame is received by component AI, and the delay between these two events must not exceed 100ms. We further impose that this constraint must be met with a probability of $1 - 10^{-4}$.

Given the temporal constraints set for a system, verifying its temporal correctness lies in checking those constraints against the system's temporal behavior. The temporal behavior of a multimedia system is given by composing the temporal behavior of each element composing the system [Demairy *et al.* 1999]. The behavior of a given individual element may be specified under the form of either statistical samples or formal distributions although the former case is the most common. Regarding composition in the context of our video-conference example, temporal constraints apply to the individual connectors for audio and video delivery, which are primitive elements. Hence, the temporal behavior of the system that is of interest here lies in the temporal behavior of each primitive connector, as obtained through sampling, and does not require any composition of temporal behaviors.

Given the proposed expression of temporal constraints and the specification of temporal behaviors through statistical samples, checking that the temporal constraints of a system are met may be rephrased as determining whether the ratio of samples that meet a temporal constraint *P* is greater than the given threshold *K*, which we call the *H* assumption in the following. Since the verification relies on statistical samples, there is a risk of error in assessing *H*. So as to assess *H* while accounting for possible errors in the samples that are used, we are able to set the probability with which H is verified based on results from the statistics domain. We do not give here the corresponding computation; the interested reader is referred to [Demairy et al. 1999].

3.2. Temporal analysis in the TCCS environment

We have introduced an approach aimed at statically verifying the correctness of a multimedia system with respect to its temporal behavior, based on the system's architectural description. This result contributes to the field of multimedia system design and analysis in general and to the field of software architecture in particular. In the context of the TCCS environment, the proposed analysis method and associated tools may be used as is, provided that there is adequate support for handling data streams offered by the underlying platform. For instance, considering the video-conference example that was initially introduced in *Deliverable B1.2*, our solution complements the current design and analysis capability of the TCCS environment, which enables analyzing the system with respect to its control functions (i.e. creation, monitoring and termination of a video-conference session).

We are further interested in applying our result to the construction of complex services that are targeted by the TCCS environment, i.e. those implemented by workflow schemas. In general, we want the analysis tools offered by the TCCS environment to be exploited for the analysis of both structural and behavioral views of complex services. Such a feature is already supported by the LTSA toolset and is being integrated within the ASTER toolset relating to middleware customization. We are currently investigating use of the proposed framework for temporal analysis, for analyzing the temporal behavior of complex services that are primarily provisioned by the TCCS environment. In that context, the primary difference of targeted services compared to multimedia systems lies in the exchange of discrete data as opposed to continuous data. However, the issue of protocol selection and possible timing constraints among interacting parties remains. A direct application of our work is thus to annotate the description of the services' structural and behavioral views with the specifications supported by temporal analysis, given that the specification of protocols must be complemented with the definition of the ordering relationship over the protocols that are used. Integration of the proposed temporal analysis framework within the TCCS environment is being studied. A first step towards that goal is to examine the kinds of temporal analyses that are needed for complex service provisioning based on the approach we have undertaken for temporal analysis. The next step will then be to possibly enrich the analysis framework accordingly. Finally, we will make the resulting toolset available in the TCCS environment.

4. Conclusion

This document has presented the ongoing work in the C3DS project towards providing methods and associated tools for the QoS analysis of complex services. The proposed support subdivides into:

- A framework for the systematic customization of middleware according to the service's requirements in terms of non-functional properties. The corresponding toolset will be integrated within the TCCS environment during the third year of the project and will enable middleware customization regarding both the structural and behavioral views defining a given complex service.

- A framework for the temporal analysis of complex services. The proposed framework is currently aimed at multimedia services, i.e., services handling data streams and having soft real-time requirements. These services have been examined in the first place because the proposed temporal analysis framework conveniently complements the TCCS environment given the increasing development of multimedia applications. We are now examining use of the framework for the temporal analysis of the complex services that are primarily targeted by the TCCS environment, i.e. services implementing a workflow schema and handling discrete data. This study includes identifying the kinds of temporal analyses that may be beneficial for complex service provisioning based on the proposed approach to temporal analysis, and possibly extends the toolset for temporal analysis accordingly. The resulting toolset will finally be made available within the TCCS environment.

The above work contributes to the software architecture domain from two standpoints: (i) to our knowledge, ADL-based development environments do not provide similar analysis support, (ii) the provided analysis supports are aimed at both the structural and behavioral views of complex services, hence addressing analysis support for workflow schemas. The latter aspect constitutes one of the contributions of the overall TCCS environment regarding the design and analysis of software systems since it also stands for the LTSA analysis toolset aimed at behavioral analysis with respect to the system's functional properties.

References

[Bjorner *et al.* 1998] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, M. Pichora, H. B. Sipma, T. E. Uribe. STeP – The Stanford Temporal Prover Educational Release. Stanford University, 1.4-a edition. July 1998.

[Demairy *et al.* 1999] E. Demairy, E. Anceaume, V. Issarny. On the Correctness of Multimedia Applications. Proceedings of the 11th Euromicro Conference on Real Time Systems. June 1999.

[Mili *et al.* 1997] R. Mili, R. Mittermeir, A. Mili. Storing and Retrieving Software Components: A Refinement Based System. IEEE Transactions on Software Engineering, 23(7). July 1997.

[Schumman & Fischer 1997] J. Schumman, B. Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. Proceedings of the 12th International Conference on Automated Software Engineering. November 1997.

[Zarras 2000] A. Zarras. Systematic Customization of Middleware. PhD Thesis. University of Rennes I, Rennes, France. To appear.