Charles University, Prague, Czech Republic
Faculty of Mathematics and Physics

MASTER THESIS

Tomáš Kalibera
SOFA Support in C++ Environments
Department of Software Engineering
Advisor: Ing. Petr Túma, Dr.
Program in Computer Science
I would like to thank my advisor, Petr Tůma, for his expert advice he gave me with extraordinary patience and for providing me with study materials and hardware in a remarkably fast and effective way. Petr Hnětynka helped me to keep focused on my work by providing me with an instant technical support for his Type Information Repository and Component Definition Language Compiler. I would also like to thank Vladimír Mencel, Stano Višňovský and others for their valuable comments and ideas they provided me with during our seminars.

I hereby certify that I wrote the thesis myself, using only the referenced sources. I give consent with lending the thesis.

Prague, December 13, 2001

Tomáš Kalibera
Contents

1 Introduction 4

2 Goals 7
   2.1 Prerequisites ................................. 7
   2.2 Objectives .................................. 7

3 Structure of the Paper 8

4 Analysis 8
   4.1 CDL to C++ Mapping ......................... 8
   4.2 Connectors .................................. 10
       4.2.1 Generic Run-time and Pre-compiled Connectors 11
       4.2.2 Middleware Independence .................. 12
       4.2.3 Distribution Transparency ................. 14
   4.3 Deployment Mechanism ....................... 15
       4.3.1 Lazy and Immediate Binding ............... 15
       4.3.2 Deployment Control ....................... 17

5 Solution 18
   5.1 CDL to C++ Mapping ......................... 19
   5.2 Deployment Mechanism ....................... 20
   5.3 Connectors .................................. 21

6 CDL to C++ Mapping 22
   6.1 Mapping IDL Types ............................ 22
       6.1.1 Interfaces ............................... 23
       6.1.2 Constructed Types ....................... 23
       6.1.3 Basic Data Types ....................... 24
       6.1.4 Form of Generated Code .................. 24
   6.2 Mapping Original CDL Types .................. 24
       6.2.1 Properties ............................... 25
       6.2.2 Frames ................................. 25
       6.2.3 Architectures ......................... 26

7 Connectors 27
   7.1 Connector Manager ............................ 28
   7.2 CORBA Connectors ............................ 29
8 Deployment Mechanism 30
  8.1 Deployment Descriptor .......................... 30
  8.2 Deployment Control .............................. 31

9 Component Model Proof of Concept 33
  9.1 Address Space Dependency ....................... 33
  9.2 Hidden Reference Passing ....................... 34
  9.3 Building Static Architectures .................. 36
    9.3.1 Parameters of Instances .................... 36
    9.3.2 Parameters of Bindings ..................... 37
    9.3.3 Custom Code in Deployment .................. 38
    9.3.4 Custom Deployment Configuration ............. 39
  9.4 Conclusion ....................................... 39

10 Prototype Implementation 40
  10.1 Overview ....................................... 41
  10.2 Technical Issues ............................... 42
    10.2.1 TIR Access ................................ 42
    10.2.2 Mapping Generator ........................ 43
    10.2.3 IDL Generator ............................. 45
    10.2.4 SOFA Runtime ............................. 45
    10.2.5 CORBA Connector Manager .................. 46
    10.2.6 CORBA Connector Generator ................. 48
  10.3 Demo Applications ............................. 49
    10.3.1 Hello Space ................................ 50
    10.3.2 Banking Demo ............................. 50
    10.3.3 Gnome Applet ............................. 50

11 Conclusion ....................................... 51

12 Summary ......................................... 52

A Deployment Descriptor 53
1 Introduction

When creating huge software projects in big teams of programmers, it is a general approach to split the application into smaller parts (components) which have as simple interactions as possible. Structure of components the application is built of (architecture) as well as interactions between these components should be described in an unambiguous way.

Once this is done, it shall be possible to develop the components separately, knowing only their required interactions with other components. It shall also be possible to reuse some of the components in newer versions of the application or in other applications.

Component models try to provide means of describing an architecture and interactions in a technical way, so that these descriptions can be exploited by software.

It is impossible for a description of interactions to be solely used for development of components, since only the component’s source code is a full description of its behavior. It seems that only a “human readable” documentation can serve this purpose.

Still, it can be useful to have descriptions allowing generic software to deploy components transparently into possibly distinct address spaces, create ties between them and run the application. Components with the “same description” can be interchanged in that process. It makes separate development and reuse of components technically easier, helps with remote communication and allows components to be written in different languages. It remains the programmer’s responsibility that components with “same descriptions” do really interact in the same way.

Each description should be as simple as possible, it should hide details of implementation of components. It must not become similar to the source code of the components. This means that the descriptions are inherently restrictive to the source code of components and to the whole application itself. Stronger restrictions allow for more features of the component model, such as for transparent remote communication, components in different languages and dynamic update of components.

At the same time, it makes applications less effective and possibly harder to implement. It is the problem of component models to find out an optimal
level of restrictions for a reasonable class of applications. It seems that such level can only be found empirically.

Indeed, there can be pretty strong restrictions disallowing many constructs that are already considered inappropriate for a class of applications. Therefore, the problem is also to provide a sense of suitability of the restrictions to be imposed on a class of applications.

There are some component models which are really used in applications, but these use extremely weak restrictions. They were not designed to serve as a framework for modular development of applications, but rather as means to support very special features in a small class of applications.

One of such component models is Microsoft DCOM\textsuperscript{1}. It was designed to allow implementation of some features of desktop and office applications, namely compound documents, in-place activation and drag/drop. It may be used to support other similar features in the future, but it is more a middleware than a component model.

DCOM lacks any kind of architecture description. It means that components cannot be deployed automatically to form applications and also that it is left to the designers to keep the structure of components consistent. Moreover, descriptions of interactions are significantly limited, there are no descriptions of dependencies between components. All ties between components are created at run time and can be changed at any time in any way. It is neither possible to update applications safely, even when they are not running, nor to remove applications from the system.

There are other component models similar to DCOM: Bonobo and UNO. Bonobo \cite{14} is a component model for Gnome, a desktop environment for X. Bonobo is built on top of CORBA \cite{9}, so it is not a middleware itself, but has otherwise all listed disadvantages of DCOM.

UNO (Universal Network Objects) \cite{7} is a component model of Open Office \cite{17} \cite{16}, an office suite being developed by Sun Microsystems. At the beginning, it was used only to similar communication purposes as DCOM, but it has been later extended to support applications consisting only of components.

It handles dependencies on components allowing upgrade when the application is not running or deinstallation. Nevertheless, it does not describe

\textsuperscript{1}Distributed Component Object Model
dependencies on objects of components, and thus does not allow dynamic update of a structure of components by a different structure of components. It does not have any kind of architecture description.

Moreover, UNO is middleware dependent. It uses its own middleware with specific restrictions making it hard to design bridges to CORBA.

SOFA (SOftware Appliances) is a component model with architecture description and with description of dependencies of components on objects. It is middleware independent and its level of restrictions on components may allow even dynamic update of components. It has not been developed to serve any too specific purpose.

Rather than starting with a specific purpose model and then trying to extend it to be more generic component model, like UNO, SOFA starts as a very generic component model with many possible features. It has to be discovered if its level of restrictions is not too high, and thus if real applications can be written.

SOFA has so far been developed for Java platform only. Currently only a very trivial demo application is available and all code generation, intended to be once automatic, is done by hand. Although SOFA was designed to be implementable in C++, too, it has to be proven that this implementation is really possible.

As stated before, it is even more important to find out weaknesses of SOFA component model when used in real applications and possibly propose changes to the model solving such problems. It is likely that the proposals would ease restrictions imposed on components by the model, going to the opposite direction than UNO does.

The resource cost for developing a “real” application in SOFA from scratch would be too high. Instead, it seems that similar feedback could be obtained by trying to integrate some “real” application into SOFA. It is believed that desktop applications are good for component models. Thus, the intention is to gain feedback on SOFA by trying to integrate Gnome, a framework for desktop applications, into SOFA.
2 Goals

2.1 Prerequisites

SOFa component model [13] defines a black box description of a component, called frame. A frame lists all objects (including their types – interfaces) the component requires and all objects the component provides.

Each component also has a “grey box” description, called architecture. An architecture can be either compound or primitive. Primitive component (component of a primitive architecture) does not have any further descriptions, it comprises only implementation provided by an application programmer.

A compound component consists of other subcomponents. All provisions and requirements of a compound component are bound to provisions or requirements of its subcomponents, bindings are also defined by the architecture [13].

Component Definition Language (CDL) [6] [2] is a special language, independent on programming languages, for defining frames and architectures. Interfaces of objects employed in frame definitions are also defined, including all types used for arguments in their methods.

2.2 Objectives

The first objective of the thesis is to design a mapping of CDL into C++. This mapping must allow for assembling components of an application both into a single address space and into multiple address spaces. It should be transparent to components if the components they communicate with are located in the same or in another address space. The mapping should not restrict connectors, used for communication between different address spaces, to be implemented by any specific middleware.

Next objective is to design a mechanism to deploy components into address spaces and create bindings as defined in architectures of the components, described in CDL.

The most important objective of the thesis is to identify weaknesses of SOFA component model when creating real applications. This includes possible weaknesses of component models in general, component models with
architecture descriptions and transparent deployment of components into different address spaces. Solutions to the discovered problems, including proposals to change or enhance SOFA component model, should be provided.

3 Structure of the Paper

Analysis of SOFA component model in real applications is provided in section 9. The analysis is based on experience gotten from prototype implementation of C++ SOFA, described in section 10.

In order to find out how to create the implementation, general problems of language mappings, transparent deployment, middleware independence and generic binding mechanisms are discussed in section 4. Because good solutions to different problems tend to be conflicting, decisions on what features are more important are made in section 5.

Based on these key decisions, language mapping of CDL into C++ is proposed in section 6. Architecture of connectors, mechanism of their generation and utilization and structure of CORBA connectors specifically are proposed in section 7. Deployment mechanism, based on the CDL to C++ mapping design, is proposed in section 8.

Short discussion of contribution of this work to SOFA and component modeling together with issues that should be addressed in the future are provided in section 11.

4 Analysis

This section analyses main problems of the objectives of the thesis. The objectives discussed here are a design of CDL to C++ language mapping, a design of connectors and a deployment mechanism. Weaknesses of SOFA component model are analyzed in section 9.

4.1 CDL to C++ Mapping

Mapping a definition language into a programming language means generation of source code. The generated code is then compiled together with user-supplied code, some parts can sometimes be compiled separately and
user code linked against it. Component Definition Language contains
description of interfaces suitable for remote invocations of methods of objects
implementing those interfaces. Types used for arguments of methods must
allow marshaling of the values.

Many C++ types do not allow for serialization, such as pointers to void
and pointers utilized to pass arrays. Instead of these types, special types
defined by the mapping must be employed. At the same time, these new
types should be easy to use for programmers, their behavior should be similar
to behavior of the traditional C++ types. This is a problem solved in any
middleware design and it leads to generation of many helper types for each
type defined in the programming language independent definition.

It would not be a good idea to reduce amount of generated code
described above since it is used too often and even when running in a single
address space. Creating more run–time types would simply make C++ appli-
cations interpreted with all performance losses it means. The experience
with existing middleware with language independent definitions (CORBA,
DCOM,..) shows that amount of source code of generated types themselves
(not stubs or skeletons) is reasonable. And this applies to SOFA more than
to any middleware since SOFA should take into account applications mostly
or solely running in a single address space. Splitting current applications
into components must not make them too much slower.

However, there is no such experience regarding representation of frames
or architectures. Current component models have neither architecture de-
scriptions nor dependencies on objects, provided objects are usually given
by runtime. Therefore, it has to be decided whether information given in
CDL should be compiled into components (and therefore included into gen-
erated code) or the information should be made available by runtime, possibly
through SOFA Type Information Repository (TIR) [2].

It is clear that using run–time solution saves not only amount of generated
code, and therefore amount of binary code of components, but it also reduces
compile time. Making compilation faster makes development of components
cheaper, so it is not true that compiling in all information would always be
the better solution.

In SOFA component model, architectures (and more precisely compo-
nents) can be viewed as implementations of frames. Although currently
SOFA is a purely static component model, some dynamics is expected to
be necessary in the future. It would be natural if a component was represented by its frame to the user code. It would then make sense to allow only type safe access to variables referencing frames. This kind of type safety requires generation of a distinct C++ type for each frame.

CDL uses the same syntax for describing interfaces and types of method arguments as OMG IDL [9]. This is actually a wise solution since IDL is language independent, allows for generation of connectors (CORBA ones) and is proven to be usable. OMG specification of IDL to C++ mapping is pretty easy to use and at the same time allows effective implementations. Under these circumstances it would be nice to allow for simple porting of CORBA applications to SOFA. This portability can be achieved by implementing as much of IDL to C++ mapping as possible.

CDL also defines types that are not part of OMG IDL, those are arrays of size given by a property. These are not ordinary OMG IDL arrays because their size is not known at compile time. More non-IDL types are types for frames and architectures, should those be generated distinct types. In order to be able to generate connectors for any ORB, non-IDL types must be mapped to IDL types. In case of the arrays of unknown length, the natural mapping is unbounded sequence.

4.2 Connectors

Remote invocation should be transparent to caller, it should behave similarly to simple local method call. In order to achieve this transparency, source code for special proxy objects is generated. These proxy objects implement the same interface as the target object, they take care of marshaling input arguments, actually making the network (or other) remote connection, and unmarshaling output arguments. They are usually called stubs and are generated by the language mapping implementation of a specific middleware.

Stubs need not to be used for calls within a single address space unless additional functionality is wanted, like intercepting calls in POA\(^2\) or some kind of synchronization, for example dynamic update. Stubs are commonly used for remote communication. Marshalling is an inherently slow process

\(^2\)Portable Object Adaptor in CORBA
itself, requiring deep copying of arguments. Network communication makes the process even slower. Therefore, it could make sense to have stubs implemented by runtime, if an overhead of a run-time implementation of stubs was negligible compared to the overhead of network communication.

### 4.2.1 Generic Run−time and Pre−compiled Connectors

In component systems it does not seem a good idea to generate all stubs at component design time. This would require generating stubs for each object of each component and for a reasonable set of middleware. Indeed, faster stubs (less run−time oriented) take longer to be generated and their code is bigger. Such approach would significantly slow down the design time and, in the end, a fraction of generated stubs will be used. Still this solution can be used in situations where run−time performance is of highest priority.

UNO is used as a component model for a whole office suite. Huge number of components reside in a single address space, stubs need to be used only for cooperation between C++ and Java components. UNO therefore generates the stubs fully at run time. The run−time stubs require a type repository for marshaling arbitrary method parameters. Type repository is already a common part of middleware implementations for other reason, handling types not known at compile time.

It is necessary to keep remote calls transparent, and therefore to create at least minimal part of a stub for the given type in the given address space. UNO creates these stubs by altering pointers to virtual method tables of objects representing interfaces of target objects of the call. This solution is very fast, but heavily C++ compiler and platform dependent. Still the stubs implemented by runtime are slower than pre−compiled stubs; penalty for each call also depends on implementation of type repository. There are no measurements comparing pre−compiled and run−time stubs.

In remote invocations, skeletons unmarshal method arguments at server side and invoke methods locally. Same as stubs, skeletons can be pre−compiled and run−time. UNO generates the skeletons at run time, too. The actual local method invocation on server object with unmarshaled arguments is again heavily C++ compiler and platform dependent.

Connectors [1] are primitives in component systems which take care of all communication between components. Stubs and skeletons are used by connectors. The simplest type of connector handles procedure call (possibly
remote) between two objects, client and server, it needs just a stub and a skeleton for the server object. Another connector type is event delivery, it allows events to be distributed from a single emitter to multiple listeners.

Event delivery can be implemented using procedure calls, but object references for listeners must be passed (by a procedure call) to the emitter. Passing object references is in strict terms considered a change of architecture and thus not allowed by strictly static component models. This is an example where even building a static architecture (fixed emitter with fixed listeners) is in traditional applications done dynamically, in this case by creating emitter component and then creating listeners and binding them to the emitter. It does not seem likely that forcing programmers to use connectors to build all ties between components can be successful. It would have to be proven that real applications could be written using strictly static component models.

In strictly static component models, all connectors needed for run time are known at deployment time. One choice would be to generate “pre–compiled” connectors at deployment time, even for example by generating their source code, compiling them and dynamically linking them to the application. This process will be very slow, but would only slow down the deployment.

However, even in case when components have in fact static architecture (converge to the static architecture very fast at run time), pre–compiled connectors could be generated at run time, as references to objects are passed through procedure calls, realized by already existing connectors. Indeed, this solution could be used in applications with very dynamic architectures, but it would not be possible to make general assumptions on responsiveness of such applications.

4.2.2 Middleware Independence

The basic idea of middleware independence of connectors is to be able to connect every two components by any middleware. Moreover, the choice of middleware for connectors should be fully transparent to the components.

There is no generally approved definition of middleware. Nevertheless, most of the systems called “middleware” use stream based communication layers, connecting different address spaces to implement transparent invocations of methods of programming language objects.

Middleware independence can be achieved by having bridges translating
the marshaled calls between two protocols of two middleware implementations. Efficiency of this solution depends very much on the similarity of the two protocols being translated. A bridge between two very similar binary remote protocols with middleware offering similar features could be very effective.

Once the features differ, protocol bridges may be able to emulate some features of one middleware in another, but this would often require access to type repository or keeping information on all objects, which is not realistic. With protocol bridges, components’ code actually needs not to be middleware independent. Components written for specific middleware would run much faster.

UNO uses its own middleware communicating by binary remote protocol URP (UNO Remote Protocol), similar to IIOP (Internet Inter-ORB Protocol) used by CORBA. UNO components are dependent on URP and there will be protocol bridges to protocols used by other middleware. Although URP and IIOP are similar, UNO has many features different from CORBA.

One of them is that UNO takes care of object life cycle by reference counting. CORBA uses reference counting only within address spaces to control life cycle of stubs. It means that reference counting used in CORBA is hidden from the IIOP to URP bridge. This and similar problems would be solved by having higher-level bridges, while there can still be incompatibilities between middleware implementations that would have to be solved in components’ code or would have to be ignored.

Another way to maintain middleware independence is to translate calls within the address space of the components, before arguments are marshaled. There would be an “independent” mapping of an “independent” interface definition language. Bridges will be stubs and skeletons conforming to this mapping while being skeletons and stubs for the given middleware.

Although it may seem straightforward to use existing middleware implementations to implement these high-level connectors, such solution may lead to deep copying method parameters since third party implementations may use memory management not compatible with the “independent” mapping. There could also be problems with implementation of run-time high-level connectors should parts of existing middleware implementations be reused. Nevertheless, from the architectural point of view, high-level connectors can
always be implemented effectively, although one possible, though impractical, solution is to re-implement the middleware to the level of its remote protocol.

4.2.3 Distribution Transparency

Distribution is often not really transparent. Different middleware implementations offer different guarantees of transparency to the components. One such guarantee can be that if two object references are different according to equality operator (\(==\)), they refer to different objects. CORBA does not guarantee this but provides means to find out that two references probably refer to different objects. It is not possible to find out for sure.

This behavior in CORBA applies even for references at protocol level to a certain extent and helps for effective management of proxies. Once a proxy (for example stub) is created, it needs not to be remembered forever as a proxy of the given reference. Thus, there can be an effective cache of proxies implemented by hash table of a fixed size.

It can be assumed that in component systems, many components exist in a single address space and remote communication is quite rare. In such situation, it would make sense for the middleware used for remote communication to emulate locality as much as possible. In other words, it would make sense to move possibly smaller distribution overhead from components, when running within a single address space, to the bigger overhead of middleware (connectors), when running in multiple address spaces. This approach would also reduce code complexity of components.

UNO guarantees object identity to be unambiguously exploited by equality operator applied on references. It also guarantees that a callback from server to client would be run in the same thread as it was initiated. This behavior is called “thread identity”. There is an additional guaranty for one-way calls in order to make them suitable for reference counting. One-way calls invoked by the same client thread on the same server object may be delivered in a different order from the order in which they were invoked.

Although additional transparency increases efficiency of a component system within a single address space, it makes it impossible to use current middleware for remote communication. It is a very restrictive solution for a component system not to support CORBA. Decision of UNO designers to provide additional transparency could have been strongly motivated by the
fact that a great amount of code had to be rewritten to components rather
than by higher efficiency in a single address space.

There are no known measurements comparing overhead of these two ap-
proaches, distribution aware code running within a single address space and
distribution unaware code running within multiple address spaces.

4.3 Deployment Mechanism

SOFA is currently a static component model, but there are some aspects
making it not strictly static. CDL allows architectures to be specified lazily,
there can be arrays of subcomponents and provided and required objects of
sizes to be specified at deployment time (by a property value). Still, bindings
of provisions and requirements in these arrays are described in architecture
definitions in CDL.

Since only primitive components contain real implementation code, the
objective of the deployment is to bind requirements of primitive components
to provisions of primitive components. The definition of this binding is split
into definitions of architectures of compound components.

Architecture of a whole application cannot be obtained only from CDL,
because compound architectures formally view their subcomponents as
frames, and a frame can be implemented by multiple architectures. It is
specified at deployment time which component implementing the specified
frame will be used, and implicitly what is the architecture of the subcom-
ponent.

The binding algorithm is simple, a tree of components, where relation
parent–child reflects relation component–subcomponent, is built, and binding
commands of the compound components (internal nodes of the tree) are
interpreted to create bindings between primitive components (leaves of the
tree).

4.3.1 Lazy and Immediate Binding

There are two fundamental choices when to create the bindings, the first
one is to create all bindings immediately (before real implementation code of
components runs), the second one is to postpone binding of each requirement
until it is needed by implementation of a primitive component.
It is obvious consequence that immediate binding ensures more deterministic responsiveness of the application and also discovers most or all binding problems at application startup. On the other hand, lazy binding provides faster start up and does not waste time binding requirements that will not be used by the specific run of the application.

There are some additional differences between these two approaches inherent to SOFA. Binding commands with value generators in CDL cannot be effectively split into bindings of single requirements to single provisions, when value generators contain properties. Value generators would have to be interpreted repeatedly for each binding or, more preferably, pre-interpreted at application startup. The latter would probably be chosen for big arrays of provisions or requirements, but there would be additional memory requirements for storing results of the interpretation.

Lazy binding also requires access to some sort of representation of compound components during whole application run time (or at least until all requirements are bound, should it be suitable to detect this), while immediate binding allows such information to be freed before actually running real code of the application. This information must include at least a containment tree of components with architectures of the components.

Immediate binding allows for various kinds of optimization not available to lazy binding. Although there can be complex bindings between arrays of provisions and requirements, there will also be bindings of a single array of requirements to a single array of provisions, mapping corresponding indexes. Such array assignment can be implemented by a single call, instead of by one call for each array index. It would save a big amount of communication overhead if components resided in different address spaces. There could possibly be more ways to group bindings into calls even in some more complex situations.

Lazy binding can be more effective than immediate binding, should dynamic components be supported and should changes to bindings be made often. In this case, only requirements needed between changes are rebound by lazy binding, while at least all requirements affected by the update are rebound by immediate binding. Still, for dynamic update of components, immediate binding can be a better solution, because the dynamic update would be a rare dynamic change, and because a consequent fall of the application due to a binding error would shortly follow the update.

Both binding approaches can be optimized for dynamic bindings by being
able to identify requirements affected by a change of bindings. Such detection in extreme case, when only the requirements really affected will be rebound, would require binding commands to be split into bindings of a single required object to a single provided object. This is already required by lazy binding in static case, but not by static immediate binding.

Although algorithms for dynamic binding would be still trivial tree traversals, choosing the optimal accuracy of identification of requirements affected by each architecture change would be the hard part. It seems that when dynamic changes are made only during dynamic updates, which upgrade the application or just fix minor bugs, minimal accuracy may be optimal. Regarding all requirements as affected may be the right solution.

Applications that would implement parts of their basic functionality by dynamic changes to their architecture, should there be any such applications, can require almost exact determination of affected requirements. Adaptive methods may help to create a solution usable for a wide range of applications.

The previous discussion of dynamic bindings assumes a design where there is at most one proxy object for each implementation object, provided by a primitive component. Another option would be to have a separate proxy for each compound component. This solution would be easy to implement, but would introduce a big overhead on each method invocation, scaling with depth of component hierarchy.

Implementation complexity may be an important factor here. Although it seems that it would be possible to synchronize an update only by proxies for implementation objects of primitive components, the actual algorithm has not been designed, yet.

4.3.2 Deployment Control

There is a general idea that deployment of a SOFA application should be configured in one place (file), this configuration is called deployment descriptor. Deployment descriptor consists of isolated parts describing deployment of individual components. There are two options how to manage deployment descriptor and how to control the whole deployment process.

One option is to allow custom configurations of individual components. Generic deployment runtime would then simply pass portions of deployment descriptor to individual components. Each component would have full control over its deployment. This approach allows for flexible interpretation of
bindings between components. Such flexibility can be crucial for a strictly static component model, where the architecture is static, but not easily built. On the other hand, this approach makes components depend on a concrete form of deployment descriptor.

Moreover, this approach dissolves structure of an application since there is no way to see the structure from the deployment descriptor without understanding deployment configuration of individual components. It is also left to the components to make sure that the configuration is correct.

Another question is how the deployment process should be distributed. Should the components control their deployment, there is the only option that distribution of deployment process copies distribution of components. However, if there was a general deployment descriptor and the components provided only basic binding methods, the deployment process could also run in a single address space. This solution would separate deployment control and deployment descriptor not only from components themselves, but also from SOFA runtime. Thus, there could be many general forms of controlling the deployment, such as multiple formats of deployment descriptors, interactive deployment, etc.

General deployment descriptor allows components to have their own configuration in separate files, which is the good thing since deployment configuration should be separated from application configuration. It remains to be seen if general deployment configuration can meet requirements of components in real applications.

This question cannot be answered formally, it is always possible to encode source code of the component into properties. Although a proof that a general deployment configuration can always be used, it does not say anything about realistic usability of a concept of general deployment configuration.

5 Solution

This section provides important design decisions on CDL to C++ language mapping, deployment mechanism and connectors. These decisions are based on analysis in section 4.
5 SOLUTION

5.1 CDL to C++ Mapping

Key decisions on the mapping are maximal run–time performance within a single address space and portability of existing CORBA applications. The mapping uses OMG IDL to C++ mapping [8] for all IDL types supported by CDL in order to maintain portability of CORBA applications and ensure high performance of operations on the mapped types.

There are no proxies mediating access to objects within a single address space, implementations of interfaces simply extend classes representing the interfaces. This decision makes local invocations faster than in CORBA by throwing away any support for intercepting calls, etc. There is no marshaling support within generated and built-in types, to get better performance within a single address space. For these reasons, SOFA cannot be built on top of any ORB.

An alternative decision could be to build SOFA on top of CORBA. This solution would provide better portability of CORBA applications and better performance of very distributed applications using CORBA as middleware. Applications running mostly within a single address space would be slower. Bad performance of CORBA in single address space was the main reason why CORBA was not used as a middleware for UNO.

The alternative solution would make it hard to build CORBA connectors for different ORBs since source code of the connector cannot be linked with two different ORBs. Even if SOFA was implemented on top of CORBA using only specified (“standard”) CORBA features, and therefore could have been compiled with any ORB, it would not be possible to have components communicating by different ORBs run within a single address space.

The main reason to allow this is to support other protocols than IIOP, possibly more effective ones in special cases, implemented by ORBs. There could also be portability, performance and licensing reasons to use specific ORBs for specific connections between components. Only protocol bridges could have been effectively used to “support multiple ORBs” if SOFA was built on top of CORBA.

In order to retain high performance within a single address space at run time, representations of individual frames and architectures will be generated. This solution also allows to access frames and architectures in a type safe manner and allows for type safe substitutions of architectures for their frame. It makes it possible to compile all type information needed for bindings,
including binding commands, into architectures of compound components. This is additional performance gain since a generic access to a type repository would always be slower.

An alternative solution could be to have a single class for all frames, instances of this class would describe different frames. These instances could contain even information specific to individual architectures. This solution would reduce amount of generated code tremendously, but would also decrease performance both in single and in multiple address spaces. It would also reduce type safety.

5.2 Deployment Mechanism

Key decisions on deployment mechanism are maximal performance of deployment within a single address space and separation of deployment configuration from code of individual components. In order to allow various optimizations, immediate binding was chosen. Specific binding code for components is generated into code representations of architectures. This code uses direct accessor to properties, provisions, requirements and instances (subcomponents) whenever possible.

The deployment is controlled from a single address space, neither components nor SOFA runtime of each address space have access to the deployment descriptor. It means there can be multiple ways to control the deployment. Although components cannot have their own form of a deployment configuration within deployment descriptor, the proposed solution does not prevent them from having it elsewhere. Since the bindings themselves are created by components, these can be adjusted by programmers of the component. A component may also be configured as primitive in a deployment descriptor, while being in fact compound component with its own ways to describe bindings.

Although altering the general architecture description seems to be a bad decision for philosophical and conceptual reasons, the technical possibility to do it is a good feature of a proposal of implementation of an evolving project.

An alternative solution would be to obtain binding information at deployment time from a type repository. It would tremendously reduce amount of generated code. It would include reduction of the number of interfaces
needed for deployment, and therefore number of connectors. In case all connectors were to be generated on demand, distributed deployment might be faster. Still, connections to a type repository will significantly slow down the deployment even in a single address space.

Giving components access to deployment descriptor and passing control over deployment to them could possibly reduce remote communication between components at deployment time, and it would allow components to have custom deployment configuration integrated into a single deployment descriptor. Although very flexible, this solution would degrade SOFA to a component model without architecture description.

5.3 Connectors

Key decisions on connectors and their architecture are middleware independence and support for run–time generation of connectors. In order to support widest range of middleware, connectors translate unmarshaled calls within address space of each SOFA runtime. The “independent” language mapping is from the connectors’ viewpoint IDL to C++. No additional distribution transparency guarantees (additional to CORBA’s ones) are provided, because those could not have been exported by CORBA connectors into different address spaces.

Support for run–time generation means that connectors may be loaded on demand one by one, and that connectors of individual interfaces can be compiled separately.

An alternative is to generate all required connectors (for the given deployment descriptor, for example), compile them and link them to SOFA runtime before starting the application. This solution will make applications run slightly faster, if connectors were only loaded at run time, or significantly faster, if connectors were also generated at run time. Compiling connectors in groups may be much faster than compiling them one by one, but would disallow a system wide cache of connectors, effective packaging, etc.

Loading connectors on demand also allows for generating them by altering some code already present (for example virtual method tables) and possibly using run–time based connectors. It is definitely a more robust solution and opens a way to variety of extensions.

There will be only connectors for remote procedure calls. These connec-
tors are essential for any application. Connectors that were more complicated could be built on top of these connectors. With an option to pass object references as method arguments, it will be possible to emulate higher-level connectors by components.

An alternative solution would be to create at least some event-based connectors. Such connectors can be added at any time should they be required.

For capacity reasons only CORBA connectors are considered as a proof of concept of the connectors’ architecture. CORBA connectors will only use CORBA specification, in order to be ORB independent.

6 CDL to C++ Mapping

6.1 Mapping IDL Types

IDL types are mapped as prescribed by CORBA 2.4 C++ Language Mapping Specification [8]. The specification does not define run-time structures, it only ensures source code compatibility. The following descriptions focus on how source code compatibility is realized and on some minor extensions. Only high-level solutions are shown, low-level implementation details conforming to CORBA specification are not included.

Although there should be some exact CDL to C++ specification that would define also run-time representation of data types, needed for effective construction of connectors, it is beyond the scope of the thesis to provide one.

The general idea is to use new features of C++ compilers widely to make the mapping easy to read and to reduce amount of generated code. General implementations of IDL to C++ mapping also have some overhead for effective marshaling, this is not needed in CDL to C++ mapping because marshaling is done in connectors.

Module ::CORBA is named ::SOFA to allow compilation of code that uses both real CORBA and SOFA, such code is needed for example by connectors.
6.1.1 Interfaces

Interfaces are mapped to classes, interface inheritance is reflected by class inheritance. Operations are mapped to pure virtual methods. Each class representing an interface inherits directly or indirectly from BaseInterface by virtual inheritance. BaseInterface defines a single string method getTypeName returning absolute type name of the class that is the same as absolute type name of the original interface.

This primitive type system is more efficient than accessing Type Information Repository, which corresponds to CORBA’s Interface Repository. It can also be easily implemented by different types of middleware.

Middleware connectors create stubs always for the narrowest type, therefore narrowing is implemented by dynamic cast. This approach is fast, but does not allow handling narrower types than those known at compile time. T.ptr types are mapped to pointers. There are helper types T.var, T.member and T.element for interface T. T.element types are used for elements of sequences.

Reference counting on interfaces is not used for life cycle control of implementation objects within a single address space, it exists only for CORBA compatibility reasons and to enable future extensions towards dynamic components, where objects and their connectors would be dynamically destroyed at run time.

6.1.2 Constructed Types

Constructed types are mapped to C++ class types with self-managed member types, as proposed by IDL to C++ mapping specification. Anonymous definitions of arrays and sequences are named _member_arr and _member_seq for member member.

Default discriminant of unions is fairly searched for through the whole range of the discriminant type, from the lowest to highest values. It would be more robust to detect free discriminant values in a random manner for large ranges, but it is unlikely there would be unions even with thousands of cases.

Helper T.member types, used also for mapping IDL union members, cannot be parts of a C++ anonymous union since they have non-trivial constructor [3]. All other members of IDL unions are put inside one anonymous C++ union in order to save as much space as possible.
6.1.3 Basic Data Types

<table>
<thead>
<tr>
<th>CDL</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>const</td>
</tr>
<tr>
<td>enum</td>
<td>enum</td>
</tr>
<tr>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>octet</td>
<td>unsigned char</td>
</tr>
<tr>
<td>wchar</td>
<td>wchar_t</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>long</td>
<td>int</td>
</tr>
<tr>
<td>long long</td>
<td>long</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
</tr>
<tr>
<td>string</td>
<td>char*</td>
</tr>
<tr>
<td>wstring</td>
<td>wchar_t*</td>
</tr>
</tbody>
</table>

Table 1: Mappings of Basic Data Types

Mappings of basic data types are shown in table 1. Mappings of numeric types are platform dependent for 32-bit Intel CPUs and serve only as an example, for other platforms different mappings would comply with IDL to C++ mapping specification.

6.1.4 Form of Generated Code

Code is generated into a directory structure that copies hierarchy of modules defined in IDL. All objects defined as direct members of the modules are mapped into distinct files. Include directives are generated relative to directory representing the root module. This structure makes it easier for selective packaging of interfaces and introduces finer granularity of dependency resolution.

6.2 Mapping Original CDL Types

Original CDL types are mapped to C++ in a way that they can be viewed as mappings of IDL types. This approach helps to integrate these new types
with IDL types and also allows for easier creation of CORBA connectors. Therefore, frames and architectures are mapped as if they were interfaces and provisions, requirements and properties are mapped as attributes. Architecture modules are mapped to namespaces.

6.2.1 Properties

Properties can be defined only within frames and architectures and can only be referenced from the frame or architecture in which they are defined. Properties defined in a frame can be referenced also from architectures implementing that frame.

Within a frame definition, properties can be used to set size of an array of requirements or provisions. Within an architecture definition, properties can be used to set size of an array of instances (subcomponents) and as indexes to arrays of provisions and requirements in binding commands and value generators.

6.2.2 Frames

A frame is mapped into a C++ class with accessors for frame properties, provisions and requirements. Helper types are generated as if the class represented IDL interface. Accessors are generated as if properties, provisions and requirements were writable attributes. The attributes are also implemented, members of the class are defined and accessors access those members in a CORBA compliant way.

The class inherits from FrameInterface described in IDL in figure 1. Modules have been omitted from the definition for better readability. The interface defines methods for generic access to a frame. This generic access is needed to be able to deploy components by a generic loader. Generic accessors for provisions, requirements and properties are generated into the class representing the frame, other methods are to be implemented by individual architectures.

Generic accessors use string identifiers of provisions, requirements and properties taken from the CDL definition of a frame. Provisions and requirements may be recursive structures of arrays of object references. In string identifiers, brackets with single index are used to access array members, dots are used to access structure members.
interface FrameInterface {

    BaseInterface getProvisionOrRequirement(in string name);
    boolean setProvisionOrRequirement(in string name,
                                           in BaseInterface value);

    boolean setProperty(in string name, in string value);

    FrameInterface getInstance(in string name);
    boolean setInstance(in string name, in FrameInterface value);

    void initialize();
    void createBindingsAndDelegates();
    void createSubsumes();

};

Figure 1: Interface of Frame

Initialization of data structures, members of a class representing a frame, is generated into method initialize(). This initialization cannot be part of a constructor because it requires properties to be set. There must be a generic way to set the properties and instantiate the component, thus properties cannot be parameters of a constructor of the component.

6.2.3 Architectures

An architecture is mapped into a C++ class with accessors for its properties and instances, as if it was an interface with attributes instead of properties and instances. The generated class inherits from a class generated as a mapping of the implemented frame. Accessors for the properties must include upcalls to possible accessors for frame properties.

Accessors for instances (subcomponents) are generated in a similar way as accessors for provisions or requirements in frames. The generated accessors are type safe, they refer to classes representing frames. Initialization for data structures is generated into method initialize(), an upcall for initialization of frame data structures must be included.
Interpretation of all commands \texttt{bind} and \texttt{delegate} from CDL is generated into method \texttt{createBindingsAndDelegates}. This method assumes that instances of this architecture (subcomponents) can already provide valid object references to its provisions and can already accept object references for its requirements. Code of the method can use direct accessors to simple (single object reference) provisions and requirements of instances (subcomponents).

Interpretation of all \texttt{subsume} commands from CDL is generated into method \texttt{createSubsumes}. This method is called only when valid object references have been assigned to all requirements of this architecture. It calls the same method recursively on all instances (subcomponents) that are part of a chain of subsume commands. Code of the method can take advantage of direct accessors to requirements and subcomponents, it can be decided at code generation time what subcomponents the recursion has to go through.

For compound architectures, a component factory is also generated, it is a simple method \texttt{createComponent} returning reference to \texttt{FrameInterface}. The generated C++ source file needs only to be compiled in order to get a representation of the compound component/architecture.

For primitive architecture, the generated source is a header file. Programmer of the component creates an implementation class that inherits from the generated architecture class.

7 Connectors

Connectors transparently handle remote invocations of methods of objects. Each connector is generated for objects of a specific interface, it consists of a server's role and client's role. Server's role of a connector is a middleware skeleton that acts as a SOFA client. Client's role of a connector is a middleware stub that acts as a SOFA object.

Connectors translate method arguments, return values and re-throw exceptions between SOFA and specific middleware. Components never know whether they are using local or remote SOFA objects. To achieve this level of transparency, client's roles of connectors must extend classes generated as a mapping of a specific CDL interface type.

Connectors are based on objects and are fully orthogonal to components. Distribution can be done with a granularity of single objects. Deployment
class ConnectorManager {
public:
    virtual BaseInterface_ptr loadCRole(const char *ref) = 0;
    virtual char *loadSRole(BaseInterface_ptr ser) = 0;
    virtual void startListening() = 0;
    virtual void stopListening() = 0;
};

Figure 2: Interface of Connector Manager

mechanism does not support loading implementation objects of a primitive component into different address spaces. Still, connectors could be used for distributed applications based only on objects, without any component model.

7.1 Connector Manager

Connector manager is a lightweight component that controls connectors of specific middleware. It is not a real SOFA component, but it is developed independently on SOFA runtime. Each SOFA runtime can be configured at deployment time to load a different set of connector managers.

The task of a connector manager is to provide control over listening loop of the given middleware and to enable creation of client and server’s roles of connectors in a middleware independent manner. Interface for connector managers is shown in figure 2, namespaces are omitted in the figure.

In order to be middleware independent, ConnectorManager interface uses stringified forms of remote references. Each middleware should be able to serialize a remote reference in one address space and then deserialize it and use in another address space. The serialized form needs not to contain all information on the remote object itself, it can very well be a name of the object that can be resolved by a middleware specific naming service. Should not be a middleware able to provide a string version of a remote reference, it is questionable how any remote communication could be started.

Method loadCRole creates a client’s role of a connector, stub for the given middleware skeleton; the stringified reference refers to BaseInterface
or narrower type. Method loadSRole creates a server’s role of a connector, middleware skeleton for the given SOFA type. It activates the skeleton and provides stringified identification of it. This identification is opaque to the caller, it can only be understood by method loadCRole of a connector manager for the same or compatible middleware, possibly run in different address space.

In addition to methods of ConnectorManager interface, connector managers for specific middleware may, and typically would, provide more methods to the specific connectors they are loading. These would include methods for loading server’s roles and client’s roles by run-time middleware representation of object references.

Such methods are needed when references are passed by connectors. A reference can be received from another address space by out, inout or return parameter of a method call mediated by client’s role of a connector. It can also be received by in or inout parameter of a method call mediated by server’s role of a connector. Client’s role of a connector for received interface must then be loaded. When a reference is to be sent out, server’s role of a connector for the respective interface must be loaded.

7.2 CORBA Connectors

CORBA connectors are created similarly to ordinary CORBA applications. IDL definitions for interfaces are generated and they are compiled by an IDL compiler. C++ sources for each connector are generated, they use files created by the IDL compiler. These sources can be compiled by any CORBA compliant ORB, only simple ORB dependent configuration is needed. Connector manager for CORBA can also be compiled by any ORB, using the same configuration. It is necessary that both connector manager and connectors to be loaded by it were compiled by the same ORB.

In order to work with any ORB, connectors must only use OMG IDL to C++ mapping specification, they cannot use any knowledge of run-time representation of data types used by specific ORB implementations. As a consequence, all parameters must be deep copied.

There are some subtle exceptions, according to the specification it should be possible to omit copying for input string parameters. It should also be possible, with some additional type casts, to omit copying for input arrays of base data types.
CORBA connectors are fully pre-compiled, including code for copying the arguments. Generic run-time CORBA connectors would have to use Dynamic Invocation Interface and Dynamic Skeleton Interface and would be much slower.

Inheritance tree of client’s roles and server’s roles copy inheritance tree of the original interfaces in order to reuse code of connectors of wider types. Client’s roles then have to break inheritance ambiguity between classes representing the interfaces and client’s roles of wider interfaces, by creating upcalls for reused methods.

Reuse of code by inheritance significantly reduces amount of generated code. It does not slow down invocations since upcalls can be inline methods. On the other hand, it introduces more dependencies to connectors and thus connectors must be loaded with coarser granularity.

8 Deployment Mechanism

There is a generic SOFA runtime configured by deployment descriptor. This runtime is generic in a way that it can load components not known at compile time of the runtime. The runtime can be used to start a server that would only allow remote loading of components, or to deploy and run an application in its address space as well as in different address spaces.

The exported interface used for loading components from different address spaces allows only to instantiate a component, all initialization and bindings are done through FrameInterface of the component remotely.

8.1 Deployment Descriptor

SOFA runtime is configured by a deployment descriptor. Formal specification in DTD is shown in figure 4 on page 54. Tree of subelements of element application copies (or defines) architecture of the application. Connector manager can be specified by element connectors. Units only help to group components into locations easily.

Deployment descriptor is currently designed to use at most one connector manager at a time. It can easily be extended to support multiple connector managers, but more research would have to be done on how and when to translate references between different middleware in SOFA runtime.
Locations are specified by serialized remote references to component loaders to be used to load the component. The default location is “local”, component loader of the local address space. Text files with the references may be specified with prefix “reffile://”.

Figure 5 on page 55 shows a deployment descriptor to start a SOFA server that would accept remote requests for loading components by omniORB. It stores a stringified reference to its component loader into file server.ref on its local file system.

Figure 6 on page 55 shows a deployment descriptor to start an application on a server described by previous example. The application has a single compound component with three primitive subcomponents. All components of the application are run on the same server.

The application is deployed by Orbacus connectors. This is possible because both omniORB and Orbacus are CORBA compliant ORBs and use a compatible stringified version of remote references (IOR).

### 8.2 Deployment Control

The deployment is controlled by a tree of components copying architecture of the application being deployed. Each node of the tree is represented by an implementation of FrameInterface. It means that each node has attributes to hold object references to provisions and requirements. At the beginning, only primitive components have valid object references stored in provision attributes.

In the first tree traversal, these references are recursively propagated from components to their parents or siblings as described by compiled-in binding commands, delegate and bind.

In the second tree traversal, references stored in requirement attributes are recursively propagated from the root component to its subcomponents as described by compiled-in subsume commands.

SOFA runtime uses methods of architectures defined by CDL to C++ mapping to bind the application. It builds the architecture tree according to Document Object Model tree of elements application and instance from the deployment descriptor.

Starting with top-level component the runtime “creates the component”: 
interface ApplicationInterface {
    short run( in sequence&lt;string&gt; args);
};

system frame ApplicationFrame {
    provides:
        ApplicationInterface ApplicationProv;
};

Figure 3: Frame and Interface of Application

• instantiates the component by component loader specified in location attribute

• sets component’s properties to values defined in deployment descriptor in elements property; uses generic accessor setProperty

• calls initialize method of the component to initialize its data structures with previously set properties

• for each instance (subcomponent) defined by instance element

    – “creates the component” by this algorithm

    – passes the subcomponent’s reference to this component by generic accessor setInstance

• invokes createBindingsAndDelegates method on the component

Handling of unit elements is not described here for clarity purposes. The algorithm is trivial and is only shown to demonstrate how the pre-compiled binding commands are used in the deployment process.

After finishing processing of the previous algorithm, SOFA runtime invokes createSubsumes method on the root component (component describing the application). This method recursively interprets subsume commands and finishes the binding.

An application is a component that implements ApplicationFrame shown in figure 3. After an application is deployed, SOFA runtime invokes method run on implementation of ApplicationInterface provided by the
deployed root component. Arguments are taken from uninterpreted arguments of command line of the SOFA runtime, which controls the deployment.

9 Component Model Proof of Concept

It has never been proven that component model of SOFA can be used for developing some real applications, not only trivial demos. SOFA requires an application to be built solely of components, it requires the components to be transparently deployable into different address spaces and it describes a static architecture of components.

It is not possible to learn from verification of another similar component model because there is not any such model. Existing component models impose very weak restrictions on applications, they do not attempt to describe or restrict architecture or its changes at all. These models are used in desktop applications.

Current weak models have proven to some extent that component models are suitable to implement at least some functionality of desktop applications. Given this experience, it seems meaningful to start with desktop applications, to check whether desktop applications can be built of SOFA components.

It would be too costly to attempt to create a brand new desktop environment, it was decided that trying to incorporate existing desktop system into SOFA would be more effective. Gnome [11] is a good desktop environment for this purpose, mostly because its sources are publicly available and it is widely used. Although probably many weaknesses of the component model could have been found simply by studying SOFA component model and Gnome architecture, the intention was that some integration principles would be tested on prototype C++ SOFA implementation. Thus, Gnome was chosen also to match platform of the prototype implementation.

To summarize, the general aim was to find out weaknesses of SOFA component model and propose solutions. To achieve this, implementation effort was made to find out how to develop Gnome applications in SOFA using components that would wrap Gnome libraries.

9.1 Address Space Dependency

Some programming methods require its custom runtime. Once an application that needs a runtime is to be built of components, it must be made sure
that the runtime is present exactly once in each address space where any of application’s components are deployed.

This scenario includes event driven applications. Exactly one instance of the event loop should be running inside each address space, where any of application’s components, which emit or handle events, are deployed.

This problem can be solved by isolating the runtime into a shared library and ensuring that it is always run at most once by proper synchronization. All components using the runtime should be linked with the shared library. This solution is not very general, because there are libraries that are not components, but components depend on them. This method is based on shared libraries that have data per address spaces.

More systematic solution would be to enhance SOFA runtime by general methods for registering running instances of a custom runtime. Each component would then contain its own copy of the runtime, it will be achieved by synchronization against SOFA runtime that the custom runtime is run at most once within each address space.

This solution will not be effective for a complex custom runtime, because its code would be copied into possibly many components running within a single address space. Instead of introducing a new abstraction for a runtime into SOFA component model, it would be more feasible to keep using custom runtime isolated in shared libraries, as long as components would be linked against shared libraries anyway.

In Gnome, address space dependency exists in Gtk library. Gtk is used by Gnome to implement user interfaces. It is an event driven library, it indirectly uses event loop functions implemented in another library, Glib. In the case of Gnome, SOFA components wrapping Gnome functionality would already need to be dynamically linked against many libraries. These components then can be also linked against a special library that would ensure that Gtk event loop is run at most once within each address space.

9.2 Hidden Reference Passing

There are situations in software when from conceptual point of view references to objects are passed, although they are neither programming language pointers nor references per se. Those pseudo-references are various handles to files, sockets, windows, etc. Passing such handles between components
could be considered as a change of architecture and be prohibited in strictly static component models.

For philosophical reasons these pseudo-references should be somehow translated to object references. For handles that are not universally unique or cannot be used to lookup the referenced item from any place, object wrappers are necessary.

These handles are for example file handles. File handles can be wrapped by objects without great performance loss, it is already done by many object oriented system libraries. Remote access to such handle can always go directly into the address space the handle is accessible from, as far as middleware used for connectors allows this.

Some handles can be used even when transferred to different address spaces. One example is X resource identification, for example X window handle. To be able to use such handle, one must know on which display the resource is. Analogous solution to the file handle problem would be to wrap window handle near where the window resides, in this case inside the specific X server.

This solution does not seem to be realistic, given the complexity of X protocol that would have to be modified to follow some CDL definitions. There are also X servers running on dedicated hardware, like X terminals.

Transition between X protocol and SOFA could also be done in a special SOFA server running separately from X server. This will slowly emulate the previous solution without having to alter X servers themselves.

A very novel solution would be to regard X protocol as a middleware and create X connectors. The connectors will provide an objective view on X resources. Big advantage of this solution is that original remote protocol would be used. Dedicated remote protocols should be, by their very nature, more effective than general middleware. More research would have to be done to make this solution work, including designing a mechanism for cooperation between multiple types of middleware.

In prototype implementation of component wrappers of Gnome, original X protocol is used through Gtk (and then GDK) library. References to windows are passed as X resource identifiers through Gtk Socket/Plug mechanism. Implementation of any of the discussed solutions would be terribly time consuming, applications will run slower in the end, and reference passing would not be avoided, anyway.
It seems that access to a file system could not be done without creating new objects and passing references at run time at all. It would be possible to have components reflecting individual files on file system (installation directory of the application), but creating new files at run time would not be possible, anyway. Generic file system access is a common example of a basic task that cannot be accomplished by a strictly static component model.

9.3 Building Static Architectures

Although it was shown there are dynamic structures of objects even in basic tasks of applications, those were still “at the edge” of the component system, connected to operating system, network or display. It is to be seen if there could still exist a pretty static structure of small enough components for real applications.

The idea of static component models is that it seems that from higher perspective many applications have static structure, and that having this structure described in a generic way would ease separation of responsibilities in software development, and it would make it possible (or easier) to design and implement a usable mechanism for dynamic update. Dynamic update is not considered here contradictory to a static component model. New architecture of an application would again be statically described.

The generic description of architecture should allow a generic code to deploy and run applications. Once application is run, no new components or bindings should be added or modified. Creating a file is an example showing that this is not always possible.

Creation of a file is an inherent dynamics and cannot be fully overcome in general cases. Nevertheless, it showed up that even creating inherently static architectures in a generic way is problematic.

9.3.1 Parameters of Instances

Some objects require parameters for their creation, in programming languages these are passed to constructors of objects. It is always possible to postpone actual initialization of an object, let caller set up parameters by special methods and then call some custom constructor of the object.

This solution is not suitable for component systems, because the parameters of objects provided by one component would have to be set up by
another component. This would happen whenever there was a general component used in a specific way in an application, because in such case the general component cannot specialize itself.

Once the postponed initialization of component’s provided object would be done by different component, re-instantiation of a component could not be transparent to (all) other components. This transparency is helpful for dynamic update.

Fortunately, initialization parameters can be passed by properties either directly, or indirectly through an external configuration. There should not be any object references among these parameters, object references must be described by binding commands.

### 9.3.2 Parameters of Bindings

Static structures of objects are built dynamically in current applications. Objects are being created and their references passed as method arguments to other objects. This transfer of references is called a binding and described within architecture descriptions in component models. The problem is that these methods often take also other arguments.

It was not discovered there would often be more than one reference argument per method in a manner not trivially replaceable by multiple calls with single reference arguments.

A binding with arity greater than two can always be expressed by multiple binary bindings and an additional object. In this situation, it means that a call with multiple references can always be replaced by multiple calls with a single reference argument and one extra non-reference argument.

Still, such situations would serve as an argument for more complex connectors, had they been discovered to exist in significant numbers, to describe connections among more than two components conceptually. Indeed, these connectors could possibly be implemented just in a way described above by binary connectors.

The problem is that it turned out there are often calls with a single reference argument and additional non-reference arguments. The non-reference arguments are parameters of a binary binding between an object on which the method is invoked and an object to which the passed reference points.

Similarly as in the problem of instance parameters, initialization of bindings could be postponed. Parameters for bindings could be provided by third
components at run time. However, it would then not be possible to change or re-create bindings transparently to components. Such transparency would be helpful for dynamic update.

Also for conceptual reasons, once bindings are described statically in CDL they should be described there fully, including their parameters. Values for the parameters then could be provided in deployment descriptor in the same manner as values of properties are provided currently. Parameters of bindings could be regarded as special types of properties.

Current properties can also be used to add configuration to bindings, but it is not defined how to identify which properties refer to which bindings, if to any. Similar identification to that currently accepted by generic instance accessors of architectures could be used. In C++ SOFA, components could alter their `setProperty` methods to handle such properties, but the conceptual solution would be to define properties of bindings in a generic way at CDL level.

Parameters of bindings are for example required by user interface containers. These containers need to know how to place graphical widgets into its window. Once containers and other widgets are mapped to distinct components, containment relation between a container and a widget is a binding and information how to place the widget inside the container is parameterization (property) of the binding.

### 9.3.3 Custom Code in Deployment

It may be necessary to have some custom code run within deployment process. It could be opening a file for reading by a system library call before a reference to it is to be bound to another component. In a process of building a user interface, it could be creating a window on X server before a reference to it is to be bound to a child widget component. There could also be some necessary preprocessing of properties before creation of an implementation object that is to be bound to another component.

There is already some custom glue code in components, this code at least creates implementation objects and sets their references to attributes for provisions. This code is run at deployment time.

Custom code in deployment means that deployment is not fully generic, but as long as it does not break transparency regarding to other components, it can hardly be considered as a conceptual problem. Attempts to describe
operations like opening a file or reading a custom configuration would tend to create a new definition language with expression power similar to a programming language.

9.3.4 Custom Deployment Configuration

It seems that it would be possible to create a static user interface built of widgets represented by individual components using current deployment descriptor with added support for parameterization of bindings. This very generic configuration would be very hard to write, read and manage. At the same time, there are already existing ways to describe a static user interface.

For Gnome there exists an interactive user interface editor, GLADE, this editor defines its own XML based interchange format of Gtk/Gnome user interface. This format is already being used to generate source code to build the interface in several programming languages. If real Gnome development under SOFA is considered, GLADE’s interchange format can be used to:

- generate frame definition and source code for a single primitive component that will build the user interface and provide objects to access signals emitted by the user interface
- configure a single primitive component that will set up the environment (same as previous one, but the component is generic)
- generate CDL definitions and deployment descriptor for a structure of components representing individual widgets, user interaction will be needed to define compound components and a containment hierarchy of components
- configure components representing individual widgets instead of configuring them by generic methods

9.4 Conclusion

Distribution transparency can be achieved with a reasonable overhead. Migration of components, possibly during a dynamic update, was not considered. Static structures of code can be mapped by a static structure of components if parameterization of bindings is added to SOFA component model. Generic description and configuration of such structures may not be suitable
to be created manually by programmers, specialized conversion and design tools may help.

There are tasks of real applications which need to pass object references (or hidden object references) between components. Some custom code must be run by all components even at deployment time. Additional attempts to describe this code should be made only when needed to introduce an important feature, like dynamic update; the descriptions may well become too complex and similar to source code.

10 Prototype Implementation

The purpose of the implementation is to show that the proposed major design features are consistent and can be implemented. Some of the presented drawbacks, such as a need for a simple middleware independent type system or necessity to deep copy arguments in ORB independent CORBA connectors, have been discovered at implementation time, and thus the implementation also helped to harmonize the design itself.

At the same time, there has been effort to make the implementation extensible to become usable for development of real applications. Such extensional efforts would have to be preceded by huge testing of the prototype implementation, the testing so far was limited to make it possible to verify the design and to make provided demos work. In order to reduce amount of labor programming work, the implementation is platform dependent, although it is a traditional requirement for component models that components can be implemented not only in different programming languages, but also on different platforms.

This text gives high-level overview of the implementation and focuses on specific technical issues. Brief installation instructions, including software and hardware requirements, instructions to run demo applications and generated technical documentation of the source code, are parts of implementation’s source tree.

CDL compiler and Type Information Repository(TIR) are SOFA tools created by Petr Hnětynka [2]. They are used by the implementation, incorporated into its source tree and discussed in this text, but were NOT created by author of this thesis.
10.1 Overview

From user’s perspective, parts of the implementation are Mapping Generator, SOFA Runtime, IDL Generator and CORBA Connector Generator.

Component’s life cycle is usually understood to be a sequence of design time, deployment time and run time [1]. Sometimes assembly time is identified as a distinct stage between design time and deployment time [5].

Life cycle of components of an application in C++ SOFA environment provided by the implementation comprises:

- *design time*

  - design of architecture and interfaces, description in CDL is created
  - CDL compiler is used to import new CDL definitions into TIR (Type Information Repository)
  - Mapping Generator is used to generate C++ representations of CDL types stored in TIR
  - C++ compiler is used to compile representation of compound components generated by Mapping Generator
  - implementation of each primitive component in C++ is created by programmer and built

Created components and CDL descriptions are exported.

- *assembly time*

  - an expert selects which components should be the application built of
  - creates deployment descriptor to deploy the selected application’s components into address spaces
  - loads CDL descriptions into TIR by CDL compiler
  - uses IDL Generator to create IDL definitions of types stored in TIR
  - uses IDL compiler of a chosen ORB to compile the IDL definitions
  - uses CORBA Connector Generator to generate source of connectors for interfaces and frames stored in TIR
— uses C++ compiler and ORB libraries of the chosen ORB to build the connectors

*Created package of application’s deployment descriptor, components and connectors for each deployment dock is exported.*

- **deployment time**

  - SOFA Runtime deploys application’s components into address spaces as described in deployment descriptor
  
  - SOFA Runtime creates bindings between deployed components as described by CDL definition (this description is compiled into components through mappings of architectures generated by Mapping Generator)

*Application’s components are deployed in address spaces and interconnected.*

- **run time**

  - SOFA Runtime runs the application by passing control to object implementing `ApplicationInterface` provided by a primitive component that implements `ApplicationFrame`

*Application is running.*

### 10.2 Technical Issues

#### 10.2.1 TIR Access

Mapping Generator, IDL Generator and CORBA Connector Generator retrieve type information from SOFA TIR. Currently TIR can be accessed only via RMI (Remote Method Invocation). Mainly for this reason, these generators are implemented in Java. It is important that TIR access is required neither at deployment time nor at run time, and therefore an end user of applications built of SOFA components does not suffer from performance drawbacks of the choice.

Part of Mapping Generator is implementation of IDL to C++ compiler. It turned out that it is much slower then C++ or even Python implementations of the compiler, although no sophisticated measurements taking into account
virtual machine start up were made. As expected, profiling analysis revealed there is a great performance loss in object serialization and deserialization in RMI communication. It also seemed that caching some TIR calls would help.

For this reason and for general conceptual reasons, a subset of TIR API, needed by the generators, was identified and extended to “TIR independent API” (package comp.tir). This API fully hides use of RMI, does not support write access to TIR, and was designed to allow for choosing repository implementation at link time.

The new repository API has been implemented by RMI wrappers to SOFA TIR. These wrappers use Java’s soft references [15] to cache objects retrieved from TIR. The fact that TIR models each item of the repository as a distinct object made it necessary to wrap all TIR interfaces describing CDL types, even when API distinction was not wanted.

In order to create a mirror structure of objects representing repository types in a way independent on TIR implementation objects, it was necessary to make additional TIR calls by wrapper objects. As a consequence, performance gain introduced by the cache was minor in Mapping Generator. It would be more significant in IDL Generator and CORBA Connector Generator, but exact performance measurements have not been made.

The implementation proves that SOFA TIR can be used to create programming language mappings of CDL definitions stored in the repository. There are some remaining minor issues with TIR dependence on IDL to Java language mapping.

TIR stores unsigned long long IDL constants in Java long variable and long double IDL constants in Java double variable as described by IDL to Java Language Mapping Specification [10]. As a consequence, although C++ has types to hold these values precisely, it uses values rounded to C++ long and C++ double.

10.2.2 Mapping Generator

IDL to C++ compiler is a significant part of Mapping Generator. There are many existing implementations of IDL to C++ compilers, but neither can be used as a black box to implement the proposed design, nor can easily be

---

3 long double is actually not supported by the specification at all
incorporated into the code, because type information is to be retrieved from TIR. It has been stated that SOFA itself does not marshal types and that designing the mapping not to be optimized for marshaling results in some performance gains.

Still, most ideas of existing IDL compilers could have been reused. The mapping is mostly based on omniORB 3.0.4. Implementation of IDL `fixed` type has been based on MICO 2.3.5. This is actually a very restricted implementation, it is effective, but does not fully comply with the specification. MICO has been chosen for its license, the reused code is licensed under GNU LGPL (GNU Library General Public License).

Implementation of type `fixed` is pretty isolated in implementations of the compilers, because it is implemented by a distinct class as defined by the mapping specification. It would therefore be simple just to use a more elaborate implementation, for example from Orbacus code, should license conditions be met.

Some IDL types, which are part of current CDL (more IDL types would likely be added to CDL in the future), are not fully supported by the implementation. IDL `any` type can only hold basic data types (numeric types and strings). As a consequence, `UnknownUserException` is not supported. Wide strings are not fully supported, either.

Generator of mappings for original CDL types is heavily integrated into IDL to C++ compiler since many IDL structures are reused by these types. All generated code is as much CORBA compliant as possible, for example, it uses correct IDL to C++ management of object references although not required by current SOFA implementation.

Structures and arrays with recursive closure are supported by all accessors and binding commands, with the following exceptions:

- when properties are used to mark array dimensions, the arrays must be single dimensional arrays of object references or frames (for provisions, requirements and instances)

- properties can only be of type `string` or of an integer type

Binding commands are fully supported and pre-compiled as designed, including value generators. Use of properties is restricted as described by the design.
Generated generic accessor do not use hash tables, but simply sequentially compare pre-parsed strings. Some third party generator of perfect hashing functions should be used in future versions, but more research would have to be done to find out how to hash accessor of fields of arrays and structures effectively. One possible perfect hash functions generator could be GGPerf [4], it can generate C++ source for perfect hash functions from Java code.

There are built-in SOFA types. BaseInterface and FrameInterface are built into all parts of the implementation. Additional types built into SOFA Runtime are ApplicationInterface, ApplicationFrame and ComponentLoaderInterface.

At the same time, these types are described in CDL (file system.cd1). CDL definitions of these types must be loaded into TIR before or at the same time as C++ SOFA tools are used.

10.2.3 IDL Generator

IDL Generator can be used to generate IDL definitions for all CDL types not including architectures. Architectures are considered to be “implementation details” of components and should be hidden within address spaces. The generated IDL files can be used by different implementations of SOFA environment tools, including a future Java implementation of SOFA environment, to generate connectors to connect to components running within an address space of (C++) SOFA Runtime. It can also be used by pure CORBA clients to connect to SOFA servers and vice versa.

The tool is independent on target language, but it is indeed affected by general CDL language mapping decisions, such as a new type system, distinct types for frames, FrameInterface, mapping arrays of property length to sequences, etc.

10.2.4 SOFA Runtime

SOFA Runtime is written in C++. The binary is independent on user types, components, connectors and middleware. It only depends on built-in types listed in section 10.2.2.

At deployment time, it builds Document Object Model (DOM) tree for a deployment descriptor by Xerces-C library. First part of the deployment is
then controlled by this tree, as described by the design. The implementation uses a wrapper object `GenericComponentLoader` to load components into address spaces. This loader connects to remote component loaders or loads components locally.

Loading components into address spaces is technically done by system’s dynamic loader, components are from the operating system’s view shared libraries. Each component has a “factory”, a function that creates object, which represents the component and returns reference to it, the function is accessible by C call conventions. This principle is used for all dynamically loaded items: connector managers, client and server’s roles of connectors and real components.

There is always an interface, a C++ class with (pure) virtual methods, known both to the loading body and to the component. Main object of the component implements this interface, extends the C++ class. In case of ordinary SOFA components, this interface is `FrameInterface`.

Components can be linked with additional shared libraries, these libraries are automatically loaded by the dynamic loader. It is important that components be not loaded with `RTLD_GLOBAL` flag. This flag would cause additionally loaded libraries to be linked also with symbols of previously loaded libraries. This would introduce namespace clashes and would not work.

It is important that this mechanism makes it possible to have two middleware implementations running in the same address space. Although it was not implemented in SOFA Runtime in the end, it was tested that two selected ORBs (omniORB and Orbacus) can coexist in a single address space. There are of course other conditions that had to be met to make this possible, such as compatible thread control. This one is not a big problem in Linux because Pthread library is widely used.

### 10.2.5 CORBA Connector Manager

CORBA Connector Manager is a connector manager that can be loaded to and used by SOFA Runtime to control connectors. It is loaded by SOFA Runtime in a way described in section 10.2.4.

The manager can be compiled by any CORBA compliant ORB, only small ORB configuration in a header file is required. This configuration includes

---

4This flag is specific for interface of dynamic loader introduced by Solaris and used by many Unix systems, including Linux.
name of the connector technology (like “omniorb”). This identifier is used
to look up individual connectors on the file system, it is necessary for future
extensions when multiple connector managers could cooperate within a single
address space.

CORBA Connector Manager was tested with omniORB 3.0.4 and Orba-
cus 4.0.5. It turned out there are some more things to be configured, such
as set up code to make the ORB choose a thread control that would allow
callbacks, and such as printing error messages. It may easily turn out that a
little more configuration will be needed when another ORB is added to the
supported ones.

Both server and client’s roles of connectors are loaded one by one. Depen-
dencies on other connectors (connectors for wider types) are to be handled
by the dynamic linker. A strange but specified [18] behavior of the dynamic
linker is that libraries are not looked up in directories formed by appending
the provided “subdir/file.so” parameter to LD_LIBRARY_PATH. Directories can
only be specified in LD_LIBRARY_PATH or only in the parameter to dlopen.
This rule also applies for lookup of libraries dependent on loaded libraries.

At the same time, connectors are stored in a directory tree corresponding
to a tree of namespaces of its interfaces and this tree is also prepended by
identifier of the connector technology. It would not help to use full paths to
load connectors, because the dependent connectors will not be found by the
dynamic linker. The dependencies cannot contain full paths to depending
connectors, because this would require the connectors to be installed in the
same directory on every computer.

The solution used in the implementation is a nasty workaround, it just
changes current directory to the specific tree of connectors (specified by envi-
ronment variable) before loading each connector. This solutions means that
current SOFA applications cannot rely on current directory, since it is being
changed time to time.

The current directory applies to a whole process, there will possibly be
components of different applications running there, therefore components
should not use current directory, anyway. Still it does not seem a good
solution to force components to synchronize on current directory in some
way.

This is indeed a more general issue, since there are other operating sys-
tem’s data managed per process. There probably should be some support
for control of what components are put into a same address space in future versions.

Regarding the dynamic linker problem, it seems that rather dynamic linker should be partially incorporated into SOFA Runtime and adapted for its requirements. It may turn out that a custom linker may also be necessary for versioning support.

One fast solution, but still a pretty nasty workaround, would be to modify link paths in binaries of the connectors after installation to full paths.

10.2.6 CORBA Connector Generator

There are general tasks that each generator of SOFA connectors would have to accomplish. These include converting frames into interfaces, provisions, requirements and properties into attributes, detecting implicit inheritance on BaseInterface and FrameInterface and computing transitive closure of inherited interfaces. These tasks are separated into ConnectorGenerator class with abstract listeners for interfaces and methods to be processed, this class is to be extended by connector generators for specific middleware implementations.

Source for CORBA connectors is ORB independent, it uses the same configuration header file for specific ORB as Connector Manager. It is unfortunately not possible to modify names of files to be included by C preprocessor macros in a portable way, and so it is left to configuration capabilities of the IDL compiler of specific ORB to name generated files in a common way.

Connectors are generated into a directory tree corresponding to the directory tree of modules of the interfaces. CORBA stubs and skeletons are expected to be generated in the same structure of directories. It is not specified that IDL compilers have configuration options to do this, so it seems that eventually post-processing would have to take place.

Two ORBs have been configured for connectors, omniORB 3.0.4 and Orbacus 4.0.5. Orbacus had no problems generating files into the described structure, but omniORB was unable to generate include directives for depending types correctly. Therefore, simple scripts had to be created to alter these directives in files generated by omniORB, using corresponding IDL files.

Having files in a directory structure corresponding to the tree of namespaces is conceptually believed to be the right way to ensure real separation
of namespaces. This principle is popular among Java programmers. It is also used for source files in UNO. Therefore, it seems that a good ORB would tend to support such structure in the future. Moreover, the omniORB problem preventing generation of correct include directives seems more like a bug than intention.

Because of dependencies on generated connectors, it was necessary to name component factories of the connectors differently, so that connectors can be linked together. Fully qualified name of the connector’s interface is therefore mangled into name of the factory.

This type of mangling could indeed be employed by CORBA Connector Manager to distinguish binaries of connectors stored in a single directory to avoid dynamic linker workarounds discussed earlier. Nevertheless, the directory structure is still much easier to be managed by installation scripts and possibly by hand.

Current implementation does not provide generation of dependencies. Dependencies should be handled together with versioning, both problems have been omitted because of capacity reasons. Therefore, compiled code for all connectors and ORB mappings of parameter types is linked together into a single library by build scripts. This big library file is then hard linked to all server and client’s roles of connectors.

If connectors are linked manually into smaller libraries with correct dependencies, they can be used by SOFA Runtime and CORBA Connector Manager, included in the current implementation.

10.3 Demo Applications

Demo applications are provided to demonstrate functionality of C++ SOFA. None of them is useful for anything else. Gnome Applet demo served as a base for analysis of usability of SOFA component model in real applications, but it is not itself a real application at all.

Although such details are not described in this text, demo applications also show how SOFA components can be compiled and how can be used to build compound and primitive components, having dependencies on generated code in mind.
10.3.1 Hello Space

This trivial demo application consists of a single compound component with three direct primitive subcomponents. The subcomponents are Emitter, Output and Application. Implementation of application’s main is included in component Application, it displays its “command line” arguments using Output, then asks Emitter to emit a message and displays the message using Output.

The objective of the demo is to show how C++ SOFA components can be implemented, built, deployed into an application and run.

There are additional versions of the same demo to show how properties work, demonstrate that components implementing the same frame can really be substituted and show how arrays and structures of provisions and requirements can be used, including binding commands with value generators.

Multiple deployment descriptors are provided to run the application within a single address space and within multiple address spaces, using omniORB and Orbacus connectors.

10.3.2 Banking Demo

Banking demo is a traditional SOFA demo application [6] [13] [12] [1]. C++ implementation tries rather to look like a simulation of a bank than a base for a system that would really control financial transactions.

Customer component is here the application’s main component, it simulates repeatedly creation of customer’s account with a “random” initial balance, attempt to withdraw “random” sum and deletion of the account.

The objective of the demo is to show that the prototype implementation is good enough to run at least common SOFA demo. Semantics of individual components can be looked up in the cited articles.

10.3.3 Gnome Applet

Gnome Applet is an implementation of a trivial applet running on Gnome panel. The demo itself uses two general SOFA/Gnome components: Applet and Button. Application’s main is provided by Applet, a general component to be used for creation of Gnome applets. The demo defines a single compound component, mandatory for each application to be built
of a tree of components, with three subcomponents: Applet, Button and ButtonListener.

It is ensured by bindings of objects implementing WidgetListener interface that the button would be displayed inside the applet. ButtonListener then implements a trivial listener for press and release events of the button, it just changes a label of the button.

The two general components, Button and Applet, are wrappers around respective Gnome and Gtk objects. Widgets are passed by Gtk Socket/Plug and Gtk event loop is implemented in a distinct shared library to ensure transparency of deployment into address spaces.

The objective of the demo is to show how at least trivial Gnome applications can be written in SOFA. The demo can be extended to support less trivial applications. Ways to write real application are discussed in section 9.

To demonstrate that Button is really a general component, another variant of the demo is provided. This variant has the same “functionality”, but runs as an application in a separate top-level window, instead of running as a Gnome applet.

11 Conclusion

CDL to C++ mapping has been designed. The mapping allows for transparent deployment of components both into a single address space and into multiple address spaces. It supports a wide range of middleware connectors. Features of the mapping include easy adoption of CORBA code, fast pre-compiled bindings and type safe access to provisions, requirements and frames.

Prototype implementation proves that the proposed design of the mapping is realistic, can be implemented and does not prohibit implementation of other important parts of the SOFA environment.

The deployment mechanism, the SOFA runtime architecture and the connectors architecture have been designed and tested on a prototype implementation. Deployment control and configuration are separated from components, access to Type Information Repository is not needed at deployment time. SOFA runtime is independent of components it loads and of the middleware used to implement the connectors. Connectors are fully object based
and can be loaded on-demand at run time. The connectors' architecture allows for run-time generation of connectors.

An ORB-independent connector manager and a generator of CORBA connectors were designed and tested on a prototype implementation. It was discovered that ORB independent connectors must deep copy method arguments, because language mapping specification of IDL does not define run-time representation of instances of data types.

The prototype implementation also helped to verify suitability of SOFA Type Information Repository [2] for CDL language mapping generators and usability of its current implementation.

Based on experience gained from the implementation of several user interface components, several problems related to deployment distribution transparency, incorporation of existing distribution and building of a static architecture were addressed. It has been proposed that parameterization of bindings be added to the SOFA component model.

Further evolution of the SOFA component model should be based on design of dynamic updating. Versioning and dependency resolution mechanism, in general and in the C++ environment in particular, were not addressed and should also be a part of an additional research.

12 Summary

An environment for development of SOFA components in C++ was designed and a prototype implementation of its basic features was created. This shows that the SOFA component model is not restricted to the Java platform, but can also be used in less interpreted environments.

Static architecture, the main characteristics of SOFA, has been discussed. While trying to identify situations where it does not work, ways have been proposed to make static architecture description easier to use where possible.

With main objectives accomplished, it should now be possible to design a dynamic update mechanism for the C++ SOFA and concurrently define ways to handle the necessary dynamics in architecture.
A Deployment Descriptor

This appendix section shows a formal definition of deployment descriptor and two examples of a deployment descriptor.
Figure 4: Format of Deployment Descriptor
<deployment>

<connectors
    source="corba/omniORB/libcm.so"
    remoteLoading="yes"
    referenceFile="server.ref"
/>

</deployment>

Figure 5: SOFA server accessible by omniORB

<deployment>

<connectors
    source="corba/orbacus/libcm.so"
    remoteLoading="no"
/>

<application source="helloapp.sc"
    location="reffile:/server.ref">

    <instance name="EmitterInst" source="cemitter.sc">
        <property name="MessageProp" value="Hello !"/>
    </instance>

    <instance name="OutputInst" source="output.sc"/>
    <instance name="HelloInst" source="hello.sc"/>

</application>

</deployment>

Figure 6: SOFA application run remotely
References


REFERENCES


