

UNIVERSITAS CAROLINA PRAGENSIS
FACULTAS MATHEMATICA PHYSICAQUE

Master Thesis

Component Definition Language

by
Vladimír Mencl

Master Thesis

Faculty of Mathematics and Physics
Charles University
Prague
The Czech Republic

Advisor: Mgr. Nguyen Duy Hoa

Declaration

This is to certify, that I wrote this thesis on my own and that the references include all the sources of information I have exploited.

Prague, April 28, 1998

Acknowledgements

This document has been produced by drawing upon the efforts of a number of people who have contributed ideas. Thanks go to all contributors, but especially to **Radovan Janeček**, **Nguyen Duy Hoa** and, last but not least, **František Plášil**, for spending his precious time in consultations with me. Thanks also to the staff of the Department of Software Engineering who made their material available to us.

Vladimír Mencil

April 1998

Contents

1	Introduction	1
1.1	Background, motivations	1
1.2	The goal of the thesis	2
1.3	Structure of the thesis	3
2	Background	4
2.1	Component oriented software development	4
2.2	Architecture description languages	5
2.3	SOFA and DCUP	8
3	Solution	12
3.1	About the language	13
3.1.1	CDL based on IDL	13
3.1.2	Language elements	14
3.2	Components	14
3.2.1	Attributes of a component	14
3.2.2	Component template	14
3.2.3	Component architecture	14
3.2.4	Properties	15
3.2.5	Requirements and provisions	16
3.2.6	Subcomponents	17
3.2.7	Implementation objects	18
3.2.8	Bindings	21
3.3	Bindings	21
3.3.1	Introduction	21
3.3.2	Entities to appear in bindings	21
3.3.3	Possible variants of a bind statement	22
3.3.4	Multiple bindings	26
3.3.5	Bindings of complex structures	27
3.3.6	Bindings of arrays	28
3.4	CDL syntax	31
3.4.1	Syntax of language constructs specific to CDL	32
3.4.2	Example 1 — sample banking application	35
3.4.3	Example 2 — multiple bindings demonstration	39
3.5	Architecture verifications performed	41
3.5.1	Preface to verification	41

3.5.2	Type-checking	42
3.5.3	Completeness check	43
3.5.4	Redundant bindings check	43
3.5.5	Cycle detection	44
3.6	Generated code (java implementation)	45
3.6.1	Built-in IDL-to-java compiler	45
3.6.2	DCUP code for interfaces and objects	47
3.6.3	DCUP code for components	48
3.6.4	Modifications of generated code required from the user	49
3.7	Summary	50
4	Current status and future work	51
5	Conclusion	53
A	CDL-grammar	57

Chapter 1

Introduction

1.1 Background, motivations

In the area of software engineering, the need to reuse existing software forces the developers to seek new approaches to the application development process. One of these approaches is component oriented programming, which provides better future for reuse. Existing techniques usually require the developer to make changes to the part being reused, which makes two disadvantages in one (covered for example in [Bosch97b]). First, it takes extra time and effort of the developer to *adopt* that part, and, second, when the original part gets modified, updating the adopted version can be difficult and may require human effort and interaction.

Component oriented programming provides an easier way for reuse. A *component* is an autonomous unit of code, having an exactly defined interface, and being a black-box to the user. The user can ignore the component's implementation, and focus on the component's interface only, which usually consists of some *provisions* and *requirements*.

Provisions are service provided by the component. The form, in which they are specified, can vary, they can be either function calls, object interfaces, or message slots, there are many *ADLs* (architecture description languages), but all of them have a way to *provide* and *require* services.

Requirements are services required by the component to be able to operate and serve the services it provides. They again might take quite any form, usually, in an ADL, the form of requirements matches the form of provisions. A discussion of existing ADLs will be given in chapter 2.

When a component is defined by its interface, the user can use it with true *black-box* approach. The user has only to *provide* the requirements the

component needs, and can use the services provided by the component without spending time to understand the internal structure of the component.

Software can be developed without even knowing, which version of the component shall be used, and from which *provider* shall the component be. And, if the binding process is automated enough, software can be easily updated by replacing only those components, which are to be updated.

The SOFA (SOFTware Appliances) project run by the *CORBA and Distributed Systems Research Group* at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, is pursuing this target. The goal of SOFA is to create an framework for an electronic market, which would allow buying software components, downloading them and using them to update existing applications.

One of the special approaches of SOFA is, that it should be possible to update applications at run-time. For very large systems, it is impossible to shutdown the whole system only because of a minor update. *DCUP* (Dynamic Component UPdating), a project ran as a part of the SOFA project, presents a framework for updating parts of an application (components) while the application is running.

To be able to reach this target, the application has to be developed according to several rules, the most important one is, that object-references crossing component boundaries have to be passed using special proxy-objects called *wrappers*. These wrappers-objects must be created as a part of the application, and there has to be one wrapper-object class for each interface type used for passing references. Also, for each **component type**, a *component builder* and a *component manager* has to be created. The code for all wrappers classes and all component builders and component managers is very similar, and therefore a code generator would be suitable. Also, it is nearly-necessary to have an architecture verification tool. That is, why *CDL* is to be created.

1.2 The goal of the thesis

The goal of this thesis is to propose an *Component Definition Language* (CDL), which would serve for the purposes of describing components and applications built of these components. The language should be implemented in a compiler program, which would both verify the architecture and components described

in a CDL source, and generate code for as much of the *control part* objects as possible.

The compiler should be able to generate code into any programming language, under which DCUP is implemented, currently, only java will be included.

The language should be strong enough to be allow support for future advances of DCUP, including proper versioning, and selection of components by the triplet (*component type, provider, version*). This way, descriptions of component written in CDL will allow storing of different versions of components from different providers, and will make possible component management bases like *template repositories* [PBJ98a].

1.3 Structure of the thesis

Chapter 2 will provide an introduction into the background of the thesis. Basic concepts of component oriented programming shall be introduced, together with a brief summary of existing architecture description languages. Also, the *SOFA* and *DCUP* projects shall be introduced there.

Chapter 3 shall present, how the thesis solves the task given. Chapter 4 will provide a summary of current status and planned future work. The thesis shall be concluded in chapter 5.

Chapter 2

Background

2.1 Component oriented software development

In the area of software engineering, the need to reuse existing software forces the developers to seek new approaches to the application development process. One of these approaches is component oriented programming, which seems to provide a good future for reuse.

Component oriented programming is an approach to application development, dividing the application development into two distinct tasks — first task being describing the *architecture* of the application in an *ADL* — *Architecture description language*, which defines how the application is assembled from individual *components*, and second task being specifying behaviour of the components. This allows to separate the *architecture description* from the implementation details. This separation makes architecture descriptions much easier to read and understand, as they are concerned with the architecture itself only.

A *component* is an autonomous unit of code, having an interface defining services provided and required by the component. The component appears as a black-box to the user. The user can ignore the component's implementation, and to use the component, it is enough for the user to understand the component's interface. The interface usually defines *provisions* and *requirements* of the component.

Provisions are service provided by the component. The form, in which they are specified, can vary, they can be either function calls, object interfaces, or message slots, every *ADL* uses a slightly different syntax, but all of them have a way to *provide* and *require* services.

Requirements are services required by the component to be able to operate and serve the services it provides. They again might take quite any form, usually, in an ADL, the form of requirements matches the form of provisions. A discussion of existing ADLs will be given in section 2.2.

Having components doing partial tasks available, an application can be easily developed by assembling the components together. If requirements defined in a component's interface are fulfilled, a component can be used for application development without having the component implementation at hand, as the component interface provides sufficient information about the component. The component implementation is needed either at the application assembling (linking) time for component models using static linking, or at application runtime, for dynamically linking models.

Using only the component interface brings in the idea of updating — the actual implementation component can be replaced by any other implementation of the component, which keeps to the same component interface. Possible updating schemes strongly depend on dynamicity provided by the programming environment, from recompiling the application with a different version of the component, through replacing the component and restarting the application, to *dynamic component updating*, when a component can be replaced during application lifetime.

The idea of *dynamic component updating* is pursued in the DCUP project (**D**ynamic **C**omponent **U**Pdating), which is run as a part of the SOFA (**S**OFtware **A**ppliances) project. This thesis proposes the *Component definition language (CDL)*, which is used for describing DCUP components, and for describing the architecture interconnecting these components. The SOFA and DCUP projects will be described in section 2.3.

2.2 Architecture description languages

There are many architecture description languages, each of them uses different approach to components, and each of them serves for different purposes. Some ADLs serve only for describing the architecture and having source codes generated, some are equipped with powerful tools for detecting deadlocks which could possibly occur at application runtime, and some even allow to simulate running the application.

As the task of this thesis is to design a new language, which shall be also an architecture description language, it was very useful and edifying to study existing architecture description languages. A lot of inspiration could be acquired from them, and also, many faults in the language design got avoided.

To provide a brief summary of the languages to the reader, a short description of the most important ADLs shall be given here. For a more detailed comparison, I recommend [Medv97], which gives an excellent summary of existing ADLs.

Darwin: Darwin is a language in many aspects similar to the CDL/DCUP environment. It uses a hierarchical model of components with provisions and requirements, and makes connections between these. The main differences between Darwin and CDL are:

- Darwin does not support component updating. On the other hand, by using the **dyn** clause, an application can be *dynamic* in the sense, that new instances of subcomponents may be created as needed — either by a *lazy* (on-demand) creation, or by creating a new instance for each request received.
- Darwin divides components into *primitive* and *composite* components. Primitive components encapsulate the functionality, and are not allowed to contain any subcomponents (that means, they are the leaves of the component hierarchy tree). Composite components are components containing subcomponents. The functionality they provide is always encapsulated in a subcomponent.

In CDL, there's no difference between a component which is a leaf of the component hierarchy tree, and a component which is not. The functionality may be contained in any component.

- CDL and Darwin also differ in the way of accessing information relating actual implementation objects. CDL uses the idea of interfaces representing implementation objects existing in a component, without touching the implementation objects, while Darwin uses the **export** and **import** clauses as “wormholes” to the “implementation space”. The CDL's approach is better suited for the task CDL is designed for, which is describing architectures designed for dynamic updating.

For those interested in learning more about Darwin, [DarwinO] can provide an introduction to Darwin.

Rapide: Rapide is a language designed for modelling the architecture before implementing it. Rapide possesses a powerful language for describing the application behaviour, so that the run of the application can be simulated. The user decides, how much of the application behaviour shall he describe, and the remainder can be replaced with sources of nondeterminism, which allow different branches of the application to be went through.

The output of an Rapide simulation is a *poset* (partially ordered set) of events, which can be later examined with supplied tools.

Rapide allows many changes in the architecture to take place at runtime, but, no new code can be supplied to the application simulation at runtime, and only new connections or component instances may be created, but no new component types can be added to the running system.

More information about Rapide can be acquired from [Rap96].

Wright: Wright is another architecture description language. In opposite to the languages described above, Wright uses *connectors* for connecting components.

Connectors are an abstraction used to embed functionality into a connection between components. Some languages use connectors, some don't, and when such functionality is necessary to be used, it is enclosed in a component, and that component is connected to both components, which would otherwise be connected by a connector. Whether connectors should be used or not is a question difficult to answer, and each language has its own answer to this question. Also, ADLs using connectors differ in the freedom user has when creating connectors. In some languages, new connectors can be defined as needed, in others, no user defined connectors are allowed, and in case of Wright, new connectors can be created only by modifying one of the five predefined connectors.

Wright is equipped with a language for specifying desired component behaviour, and many verifications of architecture correctness can be performed. For more information on wright, please refer to [AG94a] or [AG94b].

C2: C2 is another ADL introducing a new concept of interconnecting components. Instead of connecting components directly by a *binding*, or connecting them via a *connector*, C2 connects components by attaching them to a bus.

A bus might be considered an object, which has a upper and lower layer, and on each of the layers, an unlimited number of components may be attached. A message received by the bus on one layer is delivered to the other layer, and there either broadcast or routed to a particular component, according to individual preferences of all components.

Messages passed in the system are divided into *requests* and *notifications*. The whole system establishes a vertical hierarchy between components. In this vertical hierarchy, only *requests* are allowed to be sent upwards, and only *notifications* are allowed to be sent downwards.

The semantics hidden behind this constraint is, that messages going upwards (requests) correspond to calling a service on a object — the caller knows the component he is calling, and expects an either positive or negative acknowledgement to be sent in reply. Messages going downwards (notifications) correspond to replies to such requests — the object sending the notification does not know anything about the (potential) receiver of the message.

This concept proposed in C2 allows pure dynamicity to occur — objects can attach and detach the bus at any time, and adding new code to a running system is possible in C2. C2 could potentially be a very inspiring source for DCUP.

There are many other architecture description languages, but it is not the main task of this thesis to provide information about them. For those needing that information, [Medv97] does provide it.

2.3 SOFA and DCUP

The SOFA (**SO**ftware **A**ppliances) [PBJ98a] project run by the *CORBA and Distributed Systems Research Group* at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, is developing a software environment, which would allow to establish *software*

provider — user (consumer) relations. In SOFA, an application would be composed as a set of dynamically downloadable and updatable components. To achieve the goals of SOFA, many issues have to be solved. Some are already solved or being solved by DCUP (*dynamic component downloading, dynamic component updating, versioning*), and some are still to be researched (*component trading, licencing and billing and security support*).

DCUP (**D**ynamic **C**omponent **U**Pdating) [PBJ97] is a part of the SOFA project. DCUP allows components of an application to be updated at run-time, without letting the other components know, that an update took part. To be able to make the updates, all references to objects, which are to be passed across component boundaries, have to be passed through a *wrapper* object. Wrapper objects serve as proxies, and pass all requests they receive to their *target* — the original object. During update of a component, wrappers created by the component to hold references to its internal objects lock access to the inside of the component, and after updating the component and setting their *targets* to new objects, they open access to the component again.

In DCUP, a component contains exactly one *component manager*, and a *component builder*. Component manager exists for the whole lifetime of the component, and takes care of updates and other aspects of the life-cycle of a component, especially providing the other components with services supplied by the component.

The component builder is bound with a concrete version of the component, and takes care of initialization, shutdown of the component and storing and restoring the component state before and after updates.

The component builder forms, together with implementation objects and subcomponents embedded in the component, the *replaceable part* of the component, while the component manager, together with all wrapper objects, forms the *permanent part*.

The component can also be divided by the functionality of its parts, and in this view, the component manager and the component builder form the *control part*, while implementation objects, wrapper objects and subcomponents form the *functional part*. Both these schematic divisions are shown in 2.1.

DCUP is currently implemented for java environment, and is planned to be implemented for CORBA environment. To develop applications in these environments to be updatable, new approaches, different from traditional application development process, have to be applied. The application has to

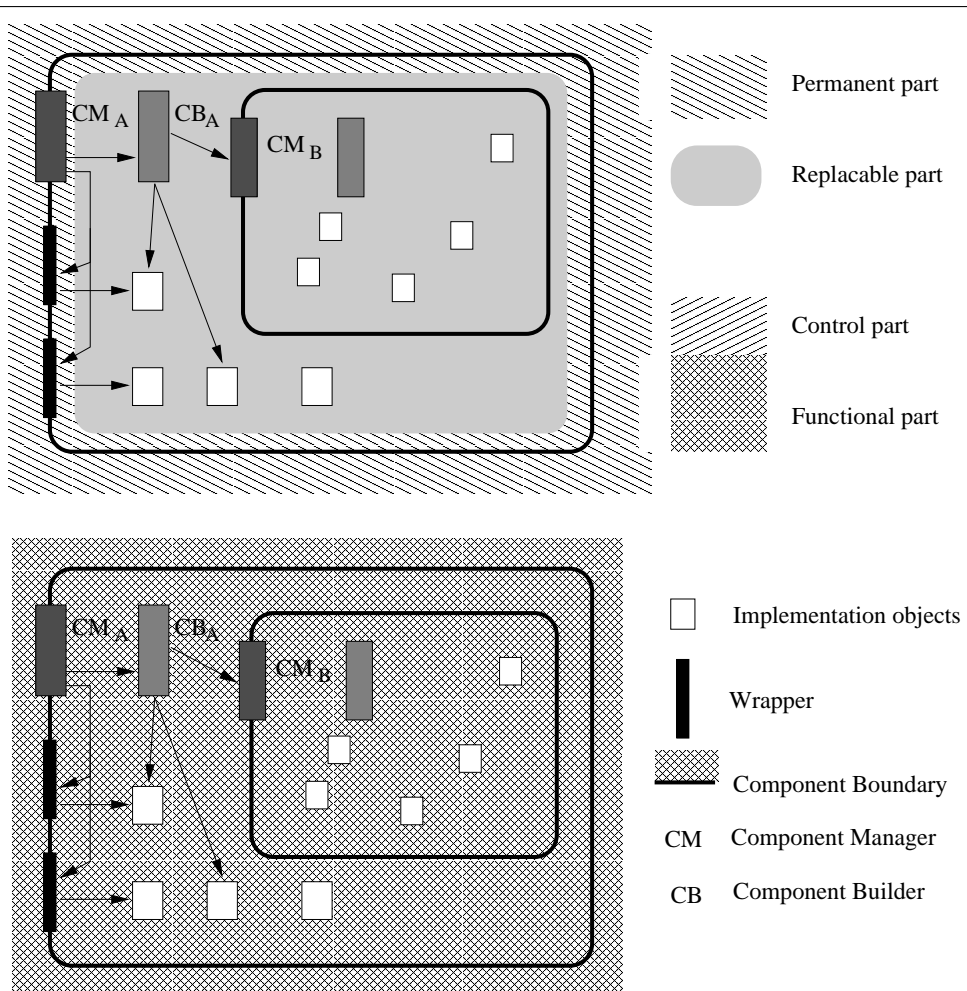


Figure 2.1: DCUP scheme of a component

be logically divided into components, and all references across the component boundaries have to be passed via a wrapper. A wrapper object class has to be created for every object class, of which references will be passed. (Or, more exactly, for every object interface. A wrapper class can be shared by different object types, if they all implement the interface which the wrapper is associated with.)

It would be very time-consuming and error-producing to write code these objects again and again, when this code could be easily generated. Also component manager and component builder classes must be created for every component type, and apart from several exceptions, this code can be generated too. This is one of the tasks done by this thesis — the CDL compiler generates all source code for all the objects mentioned here. Section 3.6 in chapter 3 will describe in detail all the code generated by the CDL compiler.

As it was already mentioned, DCUP is currently implemented only for java environments. Future implementations are planned, and although an environment supporting run-time code loading is necessary, even a C/C++ environment port could be possible, although implementing a dynamic code loader/linker would be a very hard work.

Chapter 3

Solution

This chapter describes the solution proposed by this thesis to the problem specification given in section 1.2.

First, in section 3.1, basic information about the language being proposed is provided. Then, in section 3.2, the approach of CDL to components is described, together with all attributes a CDL component has. Use of all component's attributes is illustrated by an evolving on-the-fly example. The examples will take form of fragments of code, but all these pieces of code will be fragments of the BankDemo application. Full sources and a figure will be provided for the BankDemo application in the example in section 3.4.2.

Next section (3.3) is devoted to describing bindings, which, although they're one of components' attributes, deserve a separate section due to the huge amount of topics to cover.

Then, in section 3.4, syntax of language elements introduced by CDL is described. Two larger examples are provided to show, how the CDL language is used. The first example is the BankDemo application, of which fragments of code were used to demonstrate use of individual language constructs of CDL. The second example will be a modification of the BankDemo application designed for demonstrating multiple bindings. Only differences to the original version shall be shown.

Section 3.5 shall describe what verifications of the architecture description are performed. Verifications of the language elements inherited from IDL will not be described, for they are apparent, and describing them would be a waste of space and time. Section 3.6 will describe all kinds of target language source code generated by the CDL compiler. Section 3.7 shall provide a summary of this chapter.

3.1 About the language

This section shall introduce the CDL language. Basic concepts and syntactical rules shall be introduced here, their detailed description will be provided in the following sections.

3.1.1 CDL based on IDL

Syntax of CDL — the *Component Definition Language*, is based on OMG’s IDL — the Interface Definition Language [IDLspec].

In CORBA, the OMG’s IDL is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in OMG IDL completely defines the interface and fully specifies each operation’s parameters. An OMG IDL interface provides the information needed to develop clients that use the interface’s operations.

CDL is derived from IDL both by adding new constructs and by modifying (extending) definitions of existing constructs. The new elements added are *component template definitions* and *component architecture definitions*. The constructs modified include expressions, which are extended with a possibility to refer to *component properties*, and interface definitions, which are extended to support *implementation objects requirements* — see 3.2.7.

Another new element in the language are *value generators*, which are used in **bind** statements. These allow bindings of arrays to be performed, and are replacing the earlier idea of *for-cycles*. See section 3.3.6 for a detailed discussion.

Appendix A gives the full listing of CDL grammar, only constructs not included in IDL will be described in detail in this chapter. The description of syntax is located in section 3.4.1, after concepts are introduced in sections 3.2 and 3.3.

In CDL, *components* and *interfaces* are declared. IDL was designed only for declaring interfaces. So, although the CDL grammar is a superset of the IDL grammar, pure IDL source files will not be correct CDL sources, because there are no components defined in them, and CDL is a language for defining components, interfaces are declared only “as a side effect”. A source not declaring any components is not considered a valid CDL source.

3.1.2 Language elements

CDL works with IDL elements (types, constants, interfaces + exceptions) and components, which consist of *template* (component interface) and *architecture* (concrete component type specification) parts.

3.2 Components

3.2.1 Attributes of a component

As it was mentioned earlier, a component is defined by its interface, consisting of *requirements* and *provisions*. This components interface is called **component template** in CDL, and is only a part of the component's definition. The other part is the **component architecture**, which represent a real component type, corresponds to a concrete component version.

3.2.2 Component template

The *component template* represents the component's interface. Requirements and provisions are declared here, together with *properties*, which will be passed to the component on its creation, and which can affect the requirements and provisions.

The component interfaces declared by the component template should remain unchanged during the component development and upgrading, or, at least, the new components should be backward compatible with this interface.

A component C with an component template CT is considered backward compatible with an older version of that component CO with component template CTO , if: $Req(CT) \subseteq Req(CTO)$ and $Prov(CT) \supseteq Prov(CTO)$, where $Req(T)$, resp. $Prov(T)$ is the set of requirements, resp. provisions of component template T .

In other words, a component can only be replaced by a component, which does not extend its requirements, and which provides all services provided by the component being replaced.

3.2.3 Component architecture

The component architecture describes a concrete version of an component. It *implements* a component template (which explains, why its named a template — it's a template for different version of the component).

A component has to implement a *component template* (which means to implement *provisions* declared in that interface), and is allowed to use services declared in the interfaces *requirements* section. This is accomplished by delegating calls to the services provided to either an *implementation object*, a *subcomponent*, or, as a special case, to a service *required* by the component. This delegation is done by **binding** (see 3.3). A service can be either provided by a *subcomponent* embedded in the component being declared, or serviced by an instance of an implementation object. Following subsections will describe each of these attributes of the component architecture.

3.2.4 Properties

Properties are “startup parameters” of an component. They’re passed to the component at the time of component creation by the DCUP system. They can be used internally by the implementation of the component, or they may be referred to from the CDL definition of the component, and consequently used in the generated component startup code.

Properties are a set of pairs of form $(name,value)$. The *name* element uniquely identifies the pair within the property set. The *value* element is the value of the property, and can be, in the general case, of any type, and it is up to the semantics of the component to access the property by the proper type. For properties referred to from CDL, there are special constraints on the type of *value*. Concretely, the value must be of a basic type (int, float, char, string) or a modification of such type in the particular implementation language.

The properties are usually shared by all the components, and are inherited by subcomponents from their parent components, but a component may pass its subcomponent a modified or completely different set of properties.

Properties can be referenced from expressions, and even from array dimension expressions. This allows the actual provisions and requirements of a component to be parameterized.

To allow a component to refer to a property, the component must first declare that property as a mandatory startup parameter. This is done by the **property** clause of the component template declaration.

Example: Let’s assume, we have an interface `TellerInterface` representing services provided by a teller in a bank. The CDL source code for a `Bank` component, providing a number of teller services depending on a property named `num_of_tellers`, would look like this:

```

interface TellerInterface {
    long CreateAccount();
    void DeleteAccount( in long anAccount );
    void Deposit( in long anAccount, inout float amount );
    void Withdraw( in long anAccount, inout float amount );
};

typedef TellerInterface teller_arr[$num_of_tellers];

template Bank {
    property num_of_tellers;
    provides:
        teller_arr tellers;
}

```

3.2.5 Requirements and provisions

These are attributes of the component template, and are shared by all versions of an component implementing that component template. They're specified as a pair (*type-spec*, *identifier*).

IDL-type-spec can be either a reference to an IDL interface, or to a type defined using the IDL **typedef** keyword, e.g., constructed using the *struct* and *array* IDL constructs from references to interface declarations.

If an constructed type is used, its definition is recursively interpreted up to interface declarations. *Structs* are converted to a sequence of statements, one for each member of the struct. Arrays can have their limits specified as an expression depending of property values, and therefore cannot be evaluated at compile time. They are converted into a *for-cycle* of requirements and/or provisions. For some purposes, they're treated as an array (binding), for other purposes, they're treated as a single requirement (architecture verification, see 3.5.1 for explanation), and in the generated code, they're interpreted as a for-cycle of operations managing a requirement and/or provision.

The reference takes form of an IDL *scope_name* — see section 3.4 for a definition.

Identifier can be any valid CDL identifier. CDL uses the same definition for an identifier as IDL: “An identifier is an arbitrarily long sequence of alphabetic, digit and underscore (“_”) characters. The first character must be an alphabetic character. All characters are significant.”

The identifier is used to refer to the service being provided or required in bindings, and is also used to create the name under which the service is registered in DCUP.

Example: A component Supervisor will provide a supervisor service, to assist tellers in deciding about critical transactions. To be able to provide this service, the component Supervisor requires access to the data store, this is expressed by having an requirement of type DataStoreAccess.

```
template Supervisor {
  provides:
    supervisorAccess supervisor;
  requires:
    dataStoreAccess ds;
};
```

3.2.6 Subcomponents

CDL uses a hierarchical model of components. A component can contain several subcomponents, which can implement the services provided by the component, or provide partial services, from which the component assembles the services provided. With subcomponents, it is easier and faster to develop complex components by putting small low-level components together.

Also, in DCUP, a subcomponent can be a unit of updating. An update message, received by a component, can address a subcomponent of that component. If so, the parent component will let that addressed subcomponent handle the update message.

If a subcomponent is updated, and its version changes, the parent behaviour of the parent component can be affected, and its version should change. This change is propagated up to an *updating root*. That is either the *primary template*, or an *autonomous component* (see 2.3 for explanation of these terms). A component can be made autonomous by prefixing its declaration with keyword **auto**.

A subcomponent is declared with the **inst** clause, which takes two arguments — identification of the component type, and an identifier.

The identifier is used to refer to the subcomponent from the *bind* clauses.

The component type is identified by a triplet *provider, component template, version*.

Provider is an identifier, it identifies the provider of the component, it is not interpreted anyhow, it must only match the provider identification used in the architecture section of the component being instantiated.

Version is a character string. It is enclosed in quotes, and may contain any characters. It is not interpreted anyhow, and is only used to select the right version of the component — it is matched against the version clause of the architecture section.

Component template reference takes form of an IDL *scope_name* (see section 3.4 gives a definition). Type specified here may be either a direct reference to an component template type, or it may reference an declared IDL type. This type may be constructed using the *struct* and *array* IDL constructs from references to component template types, same as for declarations of requirements and provisions in section 3.2.5.

Also here, if an constructed type is used, its definition is recursively interpreted up to component template types. *Structs* are converted to a sequence of statements, one for each member of the struct, and arrays are converted into a *for-cycle*. Due to the extensions to the definition of an expression, the size of the array can depend on actual values of the properties, and the limiting expressions of the for-cycle will refer to that propertie's value.

Arrays types allow a variable number of subcomponents to be instantiated. Struct types are not really necessary, they could be worked around, and under some circumstances can bind the user — if an struct type containing members of two different component types is used in an **inst** clause, same provider and version will be used for instantiating both the subcomponents.

Example: The Bank component can instantiate two subcomponents, DataStore and SuperVisor, both from provider *P*, each in different version. The Supervisor component will be declared as *autonomous*, and will be an *updating root*.

```
architecture P Bank version "v1" {
  inst P : BankDemo::DataStore version "v3" DS;
  auto inst P : BankDemo::Supervisor version "v4" SU;
  ...
};
```

3.2.7 Implementation objects

Implementation objects do all the functionality provided by a component — some may be done in subcomponents, but finally, every service is served by an implementation object.

It has been agreed, that implementation objects should not appear in the architecture description — they are only an implementation detail, with them, an architecture description would not be so easy to read as it is. But, to make the verification of an architecture possible (and meaningful), it is necessary to include some details relating to implementation objects in the architecture.

The reason for this is, that implementation objects may have requirements. Same as for provisions, every requirement finally ends up in an implementation object. If implementation object's requirements were not included in the architecture, any such requirement could be easily forgotten, and an architecture model could be completed and verified with serious errors undetected.

Services implemented by the implementation objects are referred to by interfaces. These interfaces declare calls which may be applied to the object implementing them. But they can also declare requirements. CDL has extended the IDL's syntax for specifying interfaces with a **requires:** clause. The syntax is same as for requirements of a component. See section 3.4 for a description of syntax, or the example bellow.

With the implementation objects not included in the architecture, but having interfaces extended to specify requirements, implementation objects' requirements can be included in the architecture.

Instead of implementation objects, component specification will contain list of interfaces implemented by all the implementation objects existing in the component — or at least those, which the designer decides to include. Objects communicating with other components (either being provided, or having requirements), shall be included always.

In the component specification, an interface will be specified only once, the number of implementation objects existing is not limited and may even change during the lifetime of the component. The presence of the interface in the component specification fore-declares the implementation objects, and ensures, that their potential requirements can be verified at the architecture-verification time.

This list of interfaces is used in two parts of architecture verification:

1. requirements of the interfaces have to be satisfied *at least once* with a binding.

- the interfaces themselves can be used in bindings to satisfy other requirements — those requirements will in fact be satisfied by an implementation object implementing that interface.

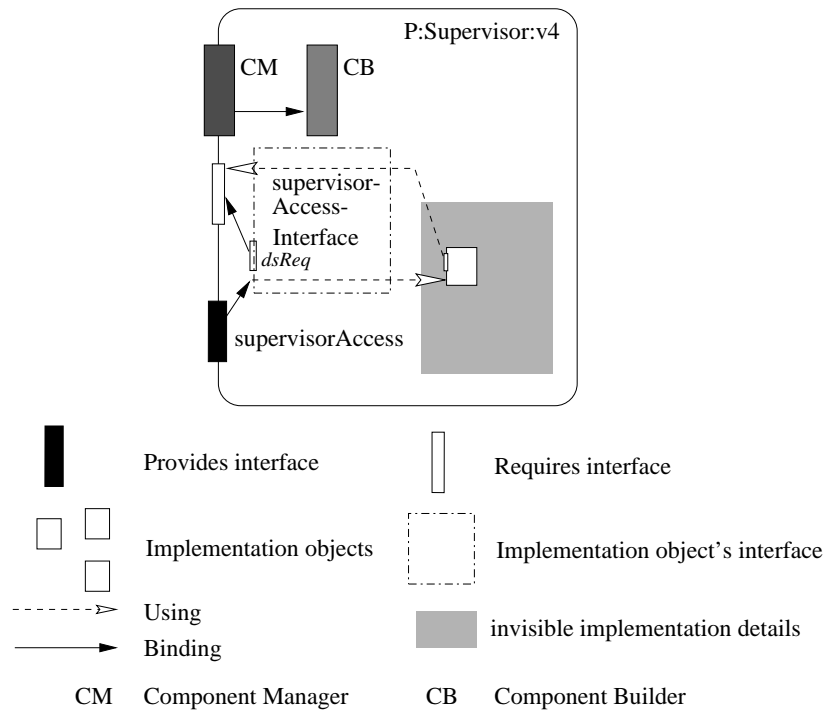


Figure 3.1: Example: Supervisor component

The interfaces will be specified in the component architecture description using the keyword **implements**, followed by the name of the interface (which is specified as a *scope-name*). Each interface should be listed at most once.

Example: Figure 3.1 shows an component Supervisor, containing an implementation object implementing the interface `supervisorAccess`. This object has a requirement, which is bound to a requirement of the component Supervisor.

Source for the component Supervisor:

```

module BankDemo {
  interface supervisorAccess {
    ...
    requires:
      BankDemo::datastoreAccess dsReq;
  };

  template Supervisor {

```

```

    provides:
      supervisorAccess supervisor;
    requires:
      datastoreAccess ds;
  };
};

architecture P Supervisor version "v4" {
  implements supervisorAccess;
  ...
};

```

3.2.8 Bindings

Bindings are used to connect entities which require binding (requirements) to entities which can be bound (provisions). They are specified as a **bind** statement in the *architecture* section of a component definition. A bind statement takes two arguments. The first one (left-hand-side one, LHS) refers to the entity which needs to be bound, this entity can be considered a *hole* in the component, which needs to be filled.

The second argument (right-hand-side, RHS) specifies the entity used to “fill in the hole”, that is, to supply a reference required by the LHS argument.

A description of bindings will be given in the following section.

3.3 Bindings

3.3.1 Introduction

As it was already described in 3.2.8, bindings connections in a component between a “hole to be filled in” and “something to fill the hole with”. Bindings are specified as a **bind** statement in the *architecture* section of a component definition, taking two arguments. The arguments will be referred to as *left-hand-side one*, *LHS* and *right-hand-side one*, *RHS* resp.

The following section will describe, which entities can be used as the LHS, resp. RHS arguments of a bind statement.

3.3.2 Entities to appear in bindings

In a component (named *C*), there are three kinds of entities, which need to be satisfied by a binding:

- provisions of component *C*

- subcomponents' requirements
- requirements of implementation object's interfaces

These entities will be together called *left-hand-side bind-operands (LHS-BO)*.

There are also three kinds of entities, which can be used in a binding to satisfy one of the above mentioned items:

- requirements of component *C*
- subcomponents' provisions
- implementation object instances

These entities will be together called *right-hand-side bind-operands (RHS-BO)*.

This makes 9 kinds of bindings, which will be described below, with examples and prototypes of code generated for the java implementation of DCUP.

For bindings having an implementation object's requirement on their left side, a multiple binding is allowed. This binding is accomplished by using the **one_of** keyword, followed by a list of possible bindings, separated by a comma (',') and enclosed in parentheses ('(' and ')'). This binding will be treated as a sequence of bindings of the nine described types.

The semantics of a multiple binding is that for every object implementing the interface containing the requirement, one of the bindings is chosen and applied. This choice is completely made by the user. For a more detailed discussion of multiple bindings, see the subsection [3.3.4 Multiple bindings](#).

3.3.3 Possible variants of a bind statement

For the following list, *C* refers to the component, in which the bindings are performed, *S*, *S1* and *S2* refer to its subcomponents, and *I*, *I1* and *I2* refer to implementation objects contained in *C*.

Types of binding:

1. binding *C*'s provision to *I*.

This exports implementation object *I*. Because in CDL, there are no implementation objects, *C*'s provision will be formally bound to an interface implemented by *I*.

```
architecture CUNI C {
    implements interfaceI;
    bind provA to implementation_of interfaceI; ...
```

For this case, it is impossible to generate complete working code, because a reference to the implementation object cannot be obtained. Therefore, we generate a partial code, and the user has to supply the reference to an implementation object implementing `interfaceI` (the object *I*).

Code in `CBuilder.onArrival`:

```
registerService(provName, !!! supply reference here !!!);
```

2. binding *C*'s provision to *S*'s provision.

This reexports service provided by *S*.

Example: `bind provA to S:provSA`;

Code in `CBuilder.onArrival`:

```
registerService(provName, S.bindToService(SProvName));
```

3. binding *C*'s provision to *C*'s requirement.

Example: `bind provA to reqA`;

This is a construct called switch, providing the same service which was required. It has been used for example in Darwin [[DarwinO](#)].

Code in `CBuilder.provideRequirements`:

```
registerService(provName,
    requirements.getRequiredReferenceNamed(reqName));
```

4. binding *S*'s requirement to *I*.

This satisfies a subcomponent's requirement with a direct reference to an implementation object. No wrapper object is created.

Example: `bind S:reqSA to implementation_of interfaceI`;

See paragraph 1 for a more detailed discussion and example.

Code in `CBuilder.provideRequirements`:

```
S.getRequirements(SRequirements);
SRequirements.setRequiredReferenceNamed(
    SReqName, !!! supply reference here !!!);
S.provideRequirements(SRequirements);
```

5. binding *S1*'s requirement to *S2*'s provision.

This satisfies a subcomponent's requirement with a service provided by some other component.

For the case when *S1* and *S2* refer to the same component, this binding is called a *loop-back*, and is also covered in Darwin [DarwinO].

Example: `bind S1:reqSA to S2:provSB;`

Code in `CBuilder.onArrival`:

```
S.getRequirements(SRequirements);
S1Requirements.setRequiredReferenceNamed( S1ReqName,
    S2.bindToService(S2ProvName));
S.provideRequirements(SRequirements);
```

6. binding *S*'s requirement to *C*'s requirement.

This reimports the service required by *C* to *S*.

Example: `bind S:reqSA to reqA;`

Code in `CBuilder.provideRequirements`:

```
S.getRequirements(SRequirements);
SRequirements.setRequiredReferenceNamed(SReqName,
    requirements.getRequiredReferenceNamed(reqName));
S.provideRequirements(SRequirements);
```

7. binding *I1*'s requirement to *I2*.

This satisfies an implementation object's requirement with a direct reference to an implementation object. This binding also makes sense, at the level of architecture verification, although at the level of code generation it's quite meaningless — we don't know either the reference to be used, or the place where it should be used, we can only know, that it should be bound, and tell the user to do so.

If this binding is a part of an multiple binding for this *I1*'s requirement, a call providing the required reference will be performed for every item in the multiple binding list. See the subsection 3.3.4 Multiple bindings for a detailed discussion.

The reference will be passed without creating a wrapper object.

If I1 and I2 refer to the same implementation object, this construct still makes sense, although caution should be taken to avoid infinite recursion.

Example:

```
bind interfaceI * reqAttrI to implementation_of interfaceI;
```

Code in `CBuilder.onArrival`:

```
provide_interfaceI_reqAttrI( !!! supply reference here !!! );
```

The user will have to provide the reference here, and also, the user will have to add code to the method `provide_interfaceI_reqAttrI`, which is designed to receive values for satisfying bindings of the requirement `reqAttrI`.

The function will be a method of the component builder object, and will take two arguments — the passed reference and a string identifying the reference for the case of multiple bindings.

This function will be called only once for each reference passed, the user has to ensure passing that reference to all implementation objects requiring this reference.

Example of such function:

```
void provide_interfaceI_reqAttrI( Object ref, String name) {
    // user code REQUIRED here
    !!! will not compile without user touching this code
}
```

Note: The name of the function is generated from the name of the interface and of the required attribute, by replacing all underscores in them with double underscores, and joining them with a single underscore. This **does** avoid naming conflicts.

8. binding *I*'s requirement to *S*'s provision.

This satisfies an implementation object's requirement with a service provided by a subcomponent.

Example: `bind interfaceI * reqAttrI to S:provSA;`

Code in `CBuilder.onArrival`:

```
provide_interfaceI_I( S.bindToService(provSA));
```

Again, body of method `provide_interfaceI` has to be provided by the user. See paragraph 7 or the subsection 3.3.4 Multiple bindings for a detailed discussion of multiple bindings.

9. binding *I*'s requirement to *C*'s requirement.

This satisfies an implementation object's requirement with a service required by the parent component.

Example: `bind interfaceI * reqAttrI to reqA;`

Code in `CBuilder.provideRequirements`:

```
provide_interfaceI_I( requirements.  
getRequiredReferenceNamed(reqName));
```

Again, body of method `provide_interfaceI` has to be provided by the user. See paragraph 7 or the subsection 3.3.4 Multiple bindings for a detailed discussion of multiple bindings.

3.3.4 Multiple bindings

As it was already mentioned, multiple bindings can be performed for a single interface requirement — because there can be several objects implementing this interface, and each can use a different service to satisfy the requirement.

Because implementation objects are at the architecture description level completely invisible, and the only notion of them is the **implements** clause of the component description, it is impossible to distinguish between them. Therefore, all possible bindings for an interface's requirement will be listed together, and for each object implementing that interface, the user will choose, which service will be used to satisfy that particular object's requirement.

The binding will be specified by the **one_of** clause on the right side of the `bind` statement, with all possible bindings given as a comma separated list enclosed in parentheses after the **one_of** keyword.

To select bindings for concrete implementation objects, user will enter code into the component builder's method corresponding to the interface and requirement name, and this code will be called once for each service, which appeared at the right hand side of the binding for this requirement. Parameters passed will be a reference to the service, and a string identification of the requirement being satisfied, and the service used for binding.

The code will be expected to call a implementation object's method corresponding to the requirement, making one call for each object. User is expected to provide an algorithm deciding how to make bindings, services will be distinguished by the string identification only.

An example of multiple bindings is given in section 3.4.2.

Also, for an complex example of usage of all attributes of an component, see section 3.4.3 Example 1 — sample banking application.

3.3.5 Bindings of complex structures

As it was already mentioned, requirements and provisions of a component can be complex structures consisting of *struct* and *array* constructs.

CDL has to provide a syntax strong enough to express bindings even between very complex structures, and has to allow even some non-trivial architectures, e.g., bindings between arrays of different sizes.

Requirements and provisions, taking form of an complex structure, are considered a set of requirements and/or provisions, consisting of primitive elements of that structure. The structure is recursively interpreted across all member type definitions, up to interface types, which are considered a *primitive* requirement and/or provision.

Inst statements can also use a complex type, with primitive elements being of type *component template*. The rules for expanding a complex component type are the same as for interface types.

To allow bindings between primitive requirements and provisions, CDL allows references to **struct** members in a bind statement, and, CDL also allows references to array members. The syntax for accessing these members is same as in C or Pascal.

Example: Type *TellerRoom* will be defined, being an array of structures, containing a member *tellers*, which is an array of interface references.

Component *BigBank*, with requirements *tellersRequired* and provisions *tellersProvided*, both of type *TellerRoom*, will make several bindings between primitive elements of these requirements and provisions.

```
typedef struct S1 { TellerInterface tellers[20];
                  TellerInterface MainTeller;
                  } TellerRoom[30];

template BigBank {
  provides:
    TellerRoom tellersProvided;
  requires:
```

```

    TellerRoom tellersRequired;
};

architecture P BigBank version "1" {

    ...
    bind tellersProvided[1].tellers[2] to
        tellersRequired[3].tellers[3];

    bind tellersProvided[2].tellers[3] to
        tellersRequired[4].tellers[5];

    bind tellersProvided[1].tellers[4] to
        tellersRequired[3].MainTeller;

    bind tellersProvided[2].MainTeller to
        tellersRequired[4].tellers[5];

    ...
};

```

This example shows, how the syntax for accessing array elements and structure members is used. Only few of the LHS-BOs are used, so the component would be incomplete, if no other bindings were specified. But already from this little of code, it is clear, that a language constructs must exist for binding all members of an array.

First reason is, that having one bind statement for each element of an array would be very difficult to read, difficult to modify, and extremely error-productive. The second reason is, that for arrays with parameterized size (depending on a property), it would be impossible to write the right amount of bind statements.

Following subsection describes, how this part is solved.

3.3.6 Bindings of arrays

Originally, a solution using *for-cycles* was used. The architecture section of a component could contain bind statements encapsulated in a for-cycle statement. The control variable of the for-cycle (which was a meta-variable), could be referenced from array index expressions, and quite complex interconnections could be expressed.

But, with this approach, describing architectures was turning into application programming, and architecture descriptions were losing their readability. The language was too much twisted to the form of the generated code, and a different solution had to be found.

To allow expressing bindings between array elements, but to keep the readability of an architecture description, a new language construct, *value generators*, was used. The idea of this construct is based on Ada's *array aggregates* ([AdaIntro]).

Syntax of value generators is described in 3.4.1, here, we'll just briefly state, that a value generator is a comma separated list of values and intervals, enclosed in angle brackets. Intervals are specified as a double dot (..) separated pair of values. Empty angle brackets denote the *full range*.

Value generators can be used in bind statements instead of array subscripts. In general, the use of a value generator states, that the bind statement should be processed for all index values generated from *processing* the value generator.

Processing a value generator means iterating through all of its values. The list of elements specified in the value generator is stepped through, for single values, the value is used, for intervals, all integer values between the interval bounds, including the bounds, are used.

In a bind statement containing value generators (VG for short), VGs on the left side are paired with VGs on the right side, starting from the outermost ones. If a bind operand consists of a subcomponent reference and a provision/requirement reference, and both of these elements contain value generators, VGs used in the subcomponent reference are considered to be *outer* than those in the provision/requirement reference, thus keeping the left-to-right order.

If, after the pairing, there are left unpaired VGs on the left side, the statement is still correct, the result will be many-to-one binding, which is acceptable. If there are unpaired VGs on the right side, the bind statement is reported as incorrect — such statement would result in one-to-many binding, which would be ambiguous and could not be correctly processed.

A bind statement containing VGs is implemented as a block of nested *for-cycles*, each for-cycle iterating through one pair of VGs, the number of the for-cycles equal to the number of VGs in the LHS bind operand.

At one level of iteration, a LHS value generator has to be completely processed. If the RHS VG is exhausted before the LHS VG is processed, the RHS VG is restarted. At each step of the iteration, values currently produced by both the left and right VG are used as array indexes, and the remaining inner part of the bind statement is processed according to these rules. In case of an unpaired LHS VG, no value is to be substituted on the right side, and no special rule is needed — the remaining inner part is also processed.

As special case should be treated VG bindings which have a reference to an interface's requirement on their left side. An interface requirement may be complex structure consisting of arrays too, and then, a question arises. Should bindings apply only to corresponding elements, or should each of them be applied to all elements of that requirement array?

This question is answered according to presence of the **one_of** clause. A single binding using a VG can be specified either alone, or as an argument of an **one_of** clause. Although these constructs look syntactically similar, their semantic meaning is very different.

If the **one_of** clause is not used, normal pairing of VGs occurs, and only bindings between corresponding members of the requirement array and RHS operand are made.

If the **one_of** clause is present, no pairing of VGs is done, and each element of the whole LHS operand — as produced by all VGs on the left side — is bound by a multiple binding to all elements of the RHS operand, as produced by all VGs on the right side.

No middle way is provided. If the user wanted to describe a architecture, where an one-dimensional requirement array would be bound to two-dimensional operand, and the outer VGs would be paired, with the inner RHS VG being “processed” by the **one_of** clause, a workaround would have to be used. Such architecture would not be directly supported, a *many-to-many* binding would have to be made, and the bindings filtered in the user-supplied code.

Examples:

1. Reexporting array of SupervisorAccess objects

```
bind SUprov[<0..$num_of_sup>] to SubC : SU[<0..$num_of_sup>];
```

2. Binding array of SupervisorAccess provisions to single SupervisorAccess provided by a subcomponent array of the same dimension, binding in a one-to-one manner.

```
bind SUprov[<0..$num_of_sup>] to SU_ARR[<0..$num_of_sup>] : SU;
```

3. Binding array of TellerInterface provisions to local implementation objects

```
bind tellers[<0..$num_of_tellers>] to  
  implementation_of BankDemo::TellerInterface;
```

4. Binding interfaces requirement to all objects provided by all members of a subcomponent array, plus a single other object

```
bind TellerInterface * dsReq to one_of (DS:ds,
    SU_ARR[<0..$num_of_sup>]:ds_restricted;
```

To make specifying bindings simpler, array indexes can be omitted in a bind operand, if the desired action is to iterate through all elements of that array. If an array index is omitted, the full range value generator is assumed, as if empty angle brackets were specified as the value generator.

Therefore, the examples 2 and 3 could be simplified to

```
bind SUprov to SubC : SU;
```

resp.

```
bind SUprov to SU_ARR : SU;
```

Also, it is legal to bind together two operands of the same type, which is an IDL defined type instead of an interface type. This type is recursively processed across *struct* and *array* definitions, arrays resulting into paired value generators, **struct** types result into a binding for each member of that structure.

3.4 CDL syntax

This section will define the syntax of language constructs used in CDL. Language constructs inherited from IDL shall not be described here, familiarity with OMG IDL is expected from the reader. The only thing to be stated here is, that CDL uses the IDL concept of namespace separated by use of **modules**. The **template** section may be nested in a module, and the component template declared in such way will be placed in the namespace at the correct point.

Component **architecture** sections can be entered only at the top level of the namespace, and the component template they refer to must be entered as a properly qualified *scoped name*. Type references occurring inside a component architecture section will be elaborated in the namespace as if they were enter in the context of the component template section corresponding to the architecture section.

3.4.1 Syntax of language constructs specific to CDL

For each of the language constructs specific to CDL, a description of syntax will be given here. Description of semantics and discussion of these concepts have been provided in the two previous sections, please refer to them as necessary.

1. Extensions of expressions.

To allow architectures to be parameterized by values not known until runtime, *properties* are introduced. They're meta-variables of any IDL type (although most usually integer), and are specific to each component. Each component declares, which properties it shall have, but instances of the same component can have different values of properties. See 3.2.4 for a detailed description of properties.

To use properties in architecture description (especially in type definitions), syntax of expressions inherited from IDL was extended with a *reference to property*. A reference to a property takes the form of a '\$' sign, optionally followed by a typecast, and always followed by the property name. If no typecast is specified, the property is expected to be of integer type. A typecast, which takes the form of type name enclosed in parentheses ('(' and ')'), can change the type of the property to any basic IDL type — that is, **float**, **char**, **string**, **boolean** or one of the integer types (**long**, **short**, **octet**).

Properties of non-integer types have very little use. Non-integer expressions can only be used as initializing values for constants, and constants have to be compile-time evaluatable. Therefore, typecast probably won't be used in the current version of the language, but is provided for completeness of the language, and for possible future extensions of the language.

Example: A property named `num_of_tellers` is referenced from a type declaration and from a `bind` statement.

```
typedef TellerInterface teller_arr[$num_of_tellers];

/* bind one-before-last element */

architecture ... {
    ...
    bind tellers[$num_of_tellers-1] to
```

```

        implementation_of TellerInterface;
    }

```

2. Value generators

Value generators are used in **bind** statements. These allow bindings of arrays to be performed, and are replacing the earlier idea of *for-cycles*. Section 3.3.6 shall provide a detailed description of their use, here, their syntax will be described.

A value generator is a construct for generating integer values, which will be used as indexes to an array. A value generator is a non-empty, comma-separated list of values and intervals.

A value is specified as an expression, an interval is specified as two expressions separated by a double dot ('..'). The expressions can fully use CDL's syntax for expressions, including references to properties.

Value generators are enclosed in angle brackets ('<' and '>'). If empty angle brackets are used, *full range* is assumed. Full range stands for all valid indexes of the array the index is used to select from, the set of valid indexes will be interval from 0 to size of the array minus one.

Example: sample *value generators*

```
<0..4,5,7..9> <0,5..$num_of_tellers> <>
```

For information on using value generators, and their semantics, see section 3.3.6.

3. Interfaces' requirements

Interfaces' requirements (which will be described in section 3.2.7), are specified by extending IDL's syntax for specifying interfaces. Requirements of interfaces are specified using the same syntax, which is used for requirements of a component, that is, by a keyword '**requires:**', followed by the list of requirements.

Example: Interface *TellerInterface* with requirements

```

interface TellerInterface {
    /* attributes and methods here */
    ...

    requires:
        BankDemo::datastoreAccess dsReq;
        BankDemo::supervisorAccess supReq;
};

```

4. Component templates

The *component template* represents the component's interface. It declares requirements and provisions of the component, together with *properties*, which the component is expecting to be set. These properties can then be referenced from expressions evaluated within the *context of the component*.

A component template is declared by a keyword **template** followed by the components name, and a list of property, provision or requirement declarations, enclosed in curly braces ('{' and '}').

Properties are declared by keyword **property** followed by the name of the property. The property declarations must precede any provision or requirement declarations.

Provisions, resp. requirements are declared by keyword **provides**, resp. **requires**, followed by a colon (':'), and a list of provision and or requirement declarations. These elementary declarations consist of a IDL-type reference (which must respect the scopes of visibility), and an identifier, unique between all requirements and provisions of that component.

Example: Component Bank with one property, two provisions and one requirement.

```
template Bank {
    property num_of_tellers;
    provides:
        teller_arr tellers;
        supervisorAccess supervisor;
    requires:
        DataStoreAccess ds;
};
```

5. Component architectures

The component architecture describes a concrete version of an component. A component template can be implemented by several different component architectures, and that is, how different versions of a component are created. Currently, there can be only one version of a component in a CDL source file, and different versions have to be specified in different source files. That is due to the lack of proper versioning support in DCUP.

A component architecture is declared by a keyword **architecture**, followed by identification of *component provider*, reference to the component

template being implemented, and **version** keyword followed by component version specified as a string literal. Then, enclosed in curly braces ('{' and '}'), the *component architecture body* follows.

The *component architecture body* may consists of several statements:

(a) **implementation_version**

This keyword takes one string literal as an argument, is mandatory, and must appear exactly once in each component. The argument passed serves for internal purposes of DCUP.

(b) **implements**

This keyword takes a reference to an IDL interface as an argument, and states, that the component shall contain local implementation objects implementing this interface. That will result in 1) ability to bind to implementations of this interface 2) obligation to satisfy requirements of that interface, if such exist. See section 3.2.7 Implementation objects for further discussion.

(c) **inst**

This keyword takes a reference to a component type and an identifier as arguments. Such statement means, that the component should create a subcomponent instance of the given component type, identifying this subcomponent by th given identifier. An IDL defined structured type having basic elements of a component type may be used here.

(d) **bind**

The **bind** keyword establishes a binding. Two operands are required here, with keyword **to** located between them. For a discussion of bindings, see section 3.3 Bindings.

3.4.2 Example 1 — sample banking application

The sample banking application, whose parts have been used as examples throughout this chapter, will be presented in whole here. Figure 3.2 shows the scheme of all the components and objects and their mutual bindings.

Here follows the CDL source code for that application. Also fragments of generated code will be presented — see section 3.6 for more information about code generation.

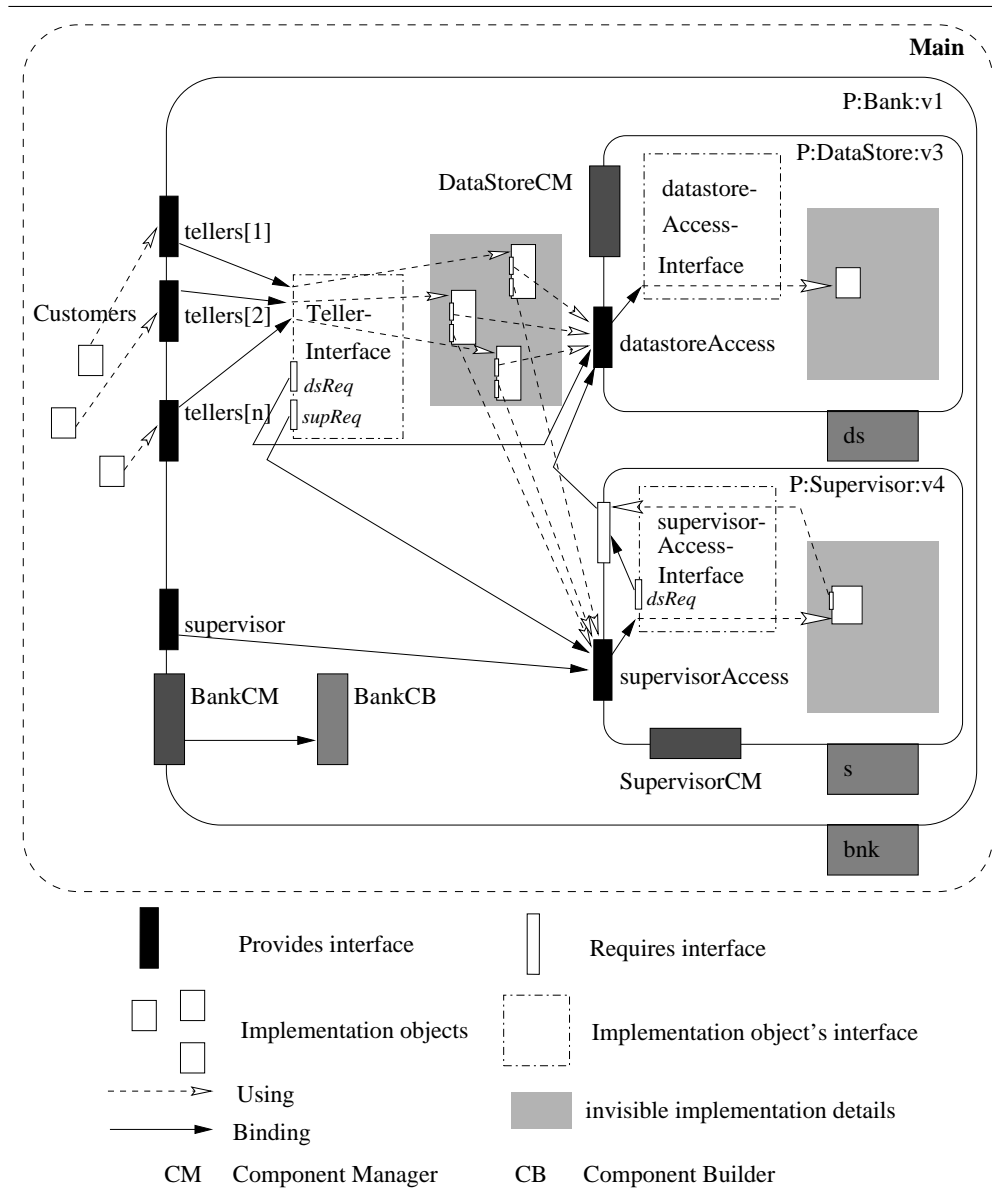


Figure 3.2: Sample banking application scheme

```

module BankDemo {

    interface TellerInterface {
        long CreateAccount();
        void DeleteAccount( in long anAccount );
        void Deposit( in long anAccount, inout float amount );
        void WithDraw( in long anAccount, inout float amount );

        requires:
            BankDemo::datastoreAccess dsReq;
            BankDemo::supervisorAccess supReq;
    };
    interface datastoreAccess { ... };

    interface supervisorAccess {
        ...
        requires:
            BankDemo::datastoreAccess dsReq;
    };

    typedef TellerInterface teller_arr[$num_of_tellers];

    template Bank {
        property num_of_tellers;
        provides:
            teller_arr tellers;
            supervisorAccess supervisor;
    };

    template Supervisor {
        provides:
            supervisorAccess supervisor;
        requires:
            datastoreAccess ds;
    };

    template DataStore {
        provides:
            datastoreAccess ds;
    };
};

architecture P Supervisor version "v4" {
    implements BankDemo::supervisorAccess;

    bind supervisor to implementation_of BankDemo::supervisorAccess;
    bind BankDemo::supervisorAccess * dsReq to ds;

    implementation_version "1";
};

```

```

architecture P DataStore version "v3" {
    implements BankDemo::datastoreAccess;

    bind ds to implementation_of BankDemo::datastoreAccess;

    implementation_version "1";
};

architecture P Bank version "v1" {
    inst P : BankDemo::DataStore version "v3" DS;
    auto inst P : BankDemo::Supervisor version "v4" SU;

    implements BankDemo::TellerInterface;

    bind SU:ds to DS:ds;
    bind supervisor to SU.supervisor;
    bind tellers[<0..$num_of_tellers-1>] to
        implementation_of BankDemo::TellerInterface;
    bind BankDemo::TellerInterface * dsReq to DS:ds;
    bind BankDemo::TellerInterface * supReq to SU:supervisor;

    implementation_version "1";
};

```

Code generated:

```

class BankBuilder {
    ...
    void provide_TellerInterface_dsReq( Object ref, String name) {
        // user code REQUIRED here
        // !!! will not compile without user touching this code
        // changed by user
        for (int i=0; i<property.GetValue("num_of_tellers"); i++) {
            tellers[i].set_dsReq(ref);
        };
    };

    void provide_TellerInterface_supReq( Object ref, String name) {
        // user code REQUIRED here
        // !!! will not compile without user touching this code
        // changed by user
        for (int i=0; i<property.GetValue("num_of_tellers"); i++) {
            tellers[i].set_supReq(ref);
        };
    };

    public void onArrival(String DataStoreID) throws CreateException {
        ...

        provide_TellerInterface_dsReq(DS.getService("ds"),"DS:ds");
        provide_TellerInterface_supReq(SU.getService("supervisor"),
            "SU:supervisor");
        ...
    };
};

```

};

};

3.4.3 Example 2 — multiple bindings demonstration

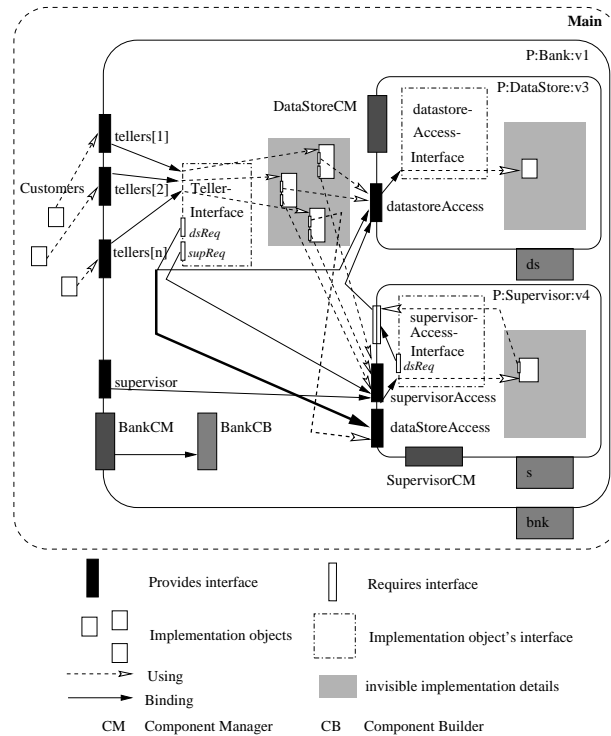


Figure 3.3: Sample banking application modified to demonstrate multiple bindings

In figure 3.3, an alternative scheme for the banking application, featuring multiple bindings of interface's requirements, is presented.

The scheme represents a situation, where the supervisor can restrict data-access for some tellers (for example, for those in training, which are not fully trusted yet), therefore, the supervisor component also reexports the *datastore-Access* service. Some tellers use this service, and some use the original service provides by the DataStore component.

This is a reasonable architecture, and CDL should be able to express it. It does so using the **one_of** keyword in the **bind** clause. Bindings new to this example are drawn in bold, here's the corresponding source.

Only parts which are changed from previous example are shown, changes are marked by the > character.

```

template supervisor {
  provides:
    supervisorAccess supervisor;
>   datastoreAccess ds_restricted;
  requires:
    datastoreAccess ds;
};
...

architecture P Supervisor version "v4" {
  implements BankDemo::supervisorAccess;

> bind ds_restricted to implementation_of BankDemo::datastoreAccess;
  bind supervisor to implementation_of BankDemo::supervisorAccess;
  bind BankDemo::supervisorAccess * dsReq to ds;

  implementation_version "1";
};

architecture P Bank version "v1" {
  ...
> bind BankDemo::TellerInterface * dsReq to
>   one_of (DS:ds, SU:ds_restricted);
  ...
};

```

In the generated source, body of method `provide_TellerInterface_dsReq` will have to be provided, with code to decide which binding to use for each teller. The final code could be for example:

```

class BankBuilder {
  ...
  void provide_TellerInterface_dsReq( Object ref, String name) {
    // user code REQUIRED here
    // !!! will not compile without user touching this code
    // changed by user
    for (int i=0; i<property.GetValue("num_of_tellers"); i++) {
      if ((isTrainee(tellers[i]) && (name="SU:dsProv")) ||
          (!isTrainee(tellers[i]) && (name!="SU:dsProv")))
        tellers[i].set_dsReq(ref);
    };
  };
  ...

  public void onArrival(String DataStoreID) throws CreateException {
    ...

    provide_teller_dsReq(DS.getService("ds"),"DS:ds");
    provide_teller_dsReq(SU.getService("dsProv"),"SU:dsProv");
  };
};

```

```

    ...
};

};

```

See section 3.6 for more information on code generation.

3.5 Architecture verifications performed

3.5.1 Preface to verification

CDL performs several tests to check the validity of an architecture specification. These tests can detect most errors made in architecture specification, and provide good help to the application developer. But these tests cannot detect all errors. First, because there are some errors, which cannot be detected by any test — they’re a fault in the logical design. Second, some errors could be well defined, but there’s no general algorithm for detecting them. This is especially true for bindings between members of arrays.

CDL provides a quite strong language for specifying bindings, allowing a *value generators* iterating through array elements in a **bind** clause.

Without knowing actual values of properties (which are most likely to appear in the array-size expression, and also in the limits of intervals in value generators), it is impossible to verify an architecture in the general case. Of course, usually, the expressions will be very simple, like the array size being only a reference to the property, and the interval limits zero for the lower limit and the array size minus one for the upper limit. But even in this simple case, where a verification algorithm seems to be possible to construct, architecture cannot be verified reliably.

The reason is, that the actual values of properties can be different in each subcomponent. Imagine the following example:

```

typedef TellerInterface teller_arr[$num_of_tellers];

template Bank {
  property num_of_tellers;
  provides:
    teller_arr tellers;
    supervisorAccess supervisor;
};

template TellerDepartment {
  property num_of_tellers;
  provides:
    teller_arr tellers;

```

```

    requires:
      supervisorAccess supervisor;
  };

architecture P Bank version "v2" {
  inst P : TellerDepartment version "v1" td;
  bind tellers[<>] to td:tellers[<>];
};

```

Although this example at the first sight seems like a correctly defined architecture, it is necessary to stress, that this correctness is based on assumption, that the value of property *num_of_tellers* will be the same for both components *Bank* and *TellerDepartment*.

But this assumption is not based on any facts. In fact, these values can easily be different — if the *Bank* component passes a different set of properties to the *TellerDepartment* component.

If these values are different, the architecture is incorrect. For values of *num_of_tellers* in *Bank* greater than in *TellerDepartment* subcomponent, a *naming exception* would be thrown by the *TellerDepartment* subcomponent, because a nonexistent services would be requested.

So, bindings between arrays cannot be verified completely. Following subsections describe each of the tests performed (that is *Type-checking*, *Completeness check*, *Redundant bindings check* and *Cycle detection*). Each of the tests deals with the arrays-problem in a different way, see the relevant subsection to see how.

3.5.2 Type-checking

For every binding made, the types of both operands are checked. Types of operands are compared according to *subclass*, not *subtype* rules.

Subtype rules have been considered, but bringing them in could cause too many problems, and it has been put off. It is possible, that in a future version, subtype relations will be introduced. See chapter 4 for more information on future work.

Subclass relation is evaluated according to the inheritance of interfaces. Either both operands have to be of the same type, or the right-hand-side operand of the binding has to be a subclass of the left-hand-side operand. If the type-check fails, an error is reported, and although the verification continues, the architecture is considered erroneous and the code generation phase is not reached.

Arrays do not present any problem here, because a type-check is made once for each bind clause, regardless on how many arrays are included in definitions of operands of the clause and how many value generators are used.

3.5.3 Completeness check

For every *left-hand-side bind-operand* (see 3.3 for definition), a check is made, whether a binding with this operand at its left side exists. This check is very easy and clearly definable for non-array operands.

But arrays bring in two problems together. First is, as it was indicated in the preface of this section, that for a general bind controlled by value generators, we cannot verify, that the range of values of the array index expression will always cover indexes of all array members. Although for some trivial case this could be verified, in the general case, it is not possible.

After deciding not to include a different verification method for “*special (trivial) cases*”, which would have completely different behaviour than the default verification method for complex (general) cases, a solution was chosen. This solution might be considered too naive and simplifying, but it is the only solution working for the general situation.

A *left-hand-side bind-operand* will be considered satisfied, if at least one bind statement is present for this operand. For array LHS-BOs, a warning will be issued, stating, that it is not possible to reliably verify the architecture. An extra warning will be issued, if an array LHS-BO is bound only by bind statements not having a value generator for that array. This should be only a warning, not an error, because an architecture can be correct — if always, only a small fixed number is used as the array size, and all the elements are bound by a corresponding number of bind statements.

3.5.4 Redundant bindings check

This is rather an unnecessary check, and detects rather mistypes than architecture design faults, but is easy to implement, and can aid the developers.

For every non-array LHS-BO, which is not an interface’s requirement (for these, multiple bindings are allowed), a check is made, to detect multiple binding clauses with this operand at the left side.

According to the previous paragraph, this test involves only components’ provisions and subcomponents’ requirements. From these, all provisions and/or

requirements embedded in an array are omitted, and also, requirements of components embedded in a component array are omitted too. For the remaining items, the check is performed according to the above mentioned rules.

3.5.5 Cycle detection

With the range of bind clauses available in CDL, it is possible to make an cycle in the graph of bindings. If an application containing cycles was run, and a any service included in the cycle would be called, an infinite recursion would happen. Therefore, it is necessary to detect cycles.

Only provisions and requirements of components and subcomponents can be included in a cycle, implementation objects cannot be a part of an cycle (or, at least, not of an detectable one, for nothing is known about connections inside the implementation objects).

It is nearly impossible to detect cycles passing through array elements, due to the reasons indicated in the preface of this section (3.5.1), in the general case. The solution chosen is, to detect cycles only between provisions, resp. requirements which are not included in any array (that is, the provision resp. requirement is not embedded in an array, and if it is a subcomponents provision resp. requirement, that subcomponent is not embedded in a subcomponent array. These will be called *array-free provisions*, resp. *requirements*. The others will be called *array-bound provisions*, resp. *requirements*.

Having removed these algorithmically nearly-unsolvable elements, the remainder can be easily solved by an graph cycle-detecting algorithm, and so it is implemented. If an cycle is detected, an error is reported, and the verification is considered failed.

Although some sever design errors can be detected this way, some danger of cycles still remains. First, if complicated expression in array indexes are used, and an error is made, a cycle can easily happen, and can be algorithmically undetectable. The only way, how to test a architecture, would be to let the user enter values of properties for all instances of all components, and interpret the value generators in all bind clauses, that is, to simulate the application startup. But this would be an unreliable verification, which verifies only for the given set of property values. It is impossible to verify the architecture in general.

Another possible source of errors in bindings (cycles) is updating. A component, applying a loop-back on a subcomponent, could update this subcompo-

ment with a new version, using a switch. This would result into a cycle, which could be detected by an architecture verification mechanism, but the cause is, that the verification was never run on this particular combination of component versions.

But this touches an new interesting area open for research - determining component compatibility, and verifying update's correctness without applying the update. After research advances in this area, CDL might get revised to supply some kind of *compatibility information* to DCUP, but, this area is still very open.

3.6 Generated code (java implementation)

Because DCUP is currently implemented only for java, CDL code generation is limited to java too. But the CDL compiler is language independent, and a code generator can be easily added for any other language, for which DCUP is implemented.

The current java code generator generates code of several categories — classes corresponding to **typedefed** IDL types, java interface declarations corresponding to IDL interface declarations, and *component builder* and *component manager* classes for CDL components. Each of these categories will be described in a separate section.

3.6.1 Built-in IDL-to-java compiler

Because java does not support defining types (only classes and interfaces can be defined in java), types defined in IDL (and CDL) either using the **typedef** keyword, or by a *structured type declaration* (a **struct**, **union** or **enum**) have to be mapped to java classes. A recommendation for this mapping has been published by IONA in [OrbixWeb], and CDL's IDL-to-java mapping follows this recommendation.

For every defined type, a new class is defined. For details on the mapping, see [OrbixWeb], only basic scheme will be described here.

1. If the type declared is a *structured type declaration*, the mapping depends on the construct used:
 - (a) **struct**: a class, containing an attribute for each member of the struct

- (b) **union**: a class, containing an attribute for each member of the union, and an attribute named *discriminator* with type corresponding to the type of the union discriminator.
 - (c) **enum**: a class, containing a an attribute of type **int** named *value*, which represents the value of the enum, plus for each of the enum's values, a constant (**public static final**) attribute of type **int**, name equal to the name of the enum's value, and value equal to the order in the enum definition, starting from zero.
2. Basic types are mapped to the their java equivalent, according to [\[OrbixWeb\]](#). The only exception is IDL type **any**, which is mapped to java type **Object**. That is more appropriate for DCUP than the IONA suggested mapping to `IE.Iona.Orbix2.CORBA.Any`.
 3. Strings are mapped to java type **string**.
 4. Sequences are mapped to classes implementing the *sequence* concept. This class has public attributes `int length` and `buffer[]`, and method `ensureCapacity`, plus constructors for initialization with or without a starting buffer size.

Because sequence types can be created anonymous, a name is generated for the class, in form `T_sequence_<basetype>`, where `<basetype>` is the name of the base type of the sequence.

This way, only one class (and therefore, only one mapping) is generated for all sequence types of the same base-type, ignoring the maximum size expression, if present. Therefore all sequences are treated as unbounded. This does not result in any problem, since a program, expecting an bounded sequence, can easily handle an unbounded one.
 5. Types declared as an array are mapped to a class containing a single attribute named *value*, of type `<basetype> []`. This attribute represents the array, and is initialized to the array-size specified in type declaration.
 6. The only remaining type is declaring a type equal to an type already declared, no array modifiers are involved. This type declaration is implemented as a class inheriting from the class created for the base type, with no new methods or attributes.

To allow initialization of objects created from these generated classes, especially arrays, it is necessary, to pass the objects the properties of the component they are created in. Therefore, constructor of every class, generated according to the above listed rules, will take a *property* parameter as its first parameter, with any other parameters following. If a complex type's creation causes several objects to be created one from another's constructor, the property parameter will be propagated up to the innermost objects.

3.6.2 DCUP code for interfaces and objects

For every interface defined in CDL, a corresponding java interface is generated. If the interface contains private attributes (which are the internal representation of interface's requirements), an extra interface for accessing these attributes (setting the required references) is generated. And of course, the wrapper object class for accessing the implementation objects from other components.

Interfaces

The IDL interface is mapped to a java interface, extending exactly the same set of interfaces, as was the set of inheritances of the IDL interface. All attributes are mapped to a pair of methods for reading and modifying value of that attribute.

Operations are mapped to equivalent java methods. Sequences used as parameters are passed as references to relevant sequence classes. Output parameters (those declared **inout** or **out** in IDL, resp. CDL), require to be passed by reference. Because java supports only arguments passed by value, such parameters are mapped to passing *holder objects* by value. The holder objects contain a reference to a object of the respective type. Holder object classes are provided for basic IDL types. For user defined types, they're generated as necessary.

The **oneway** attribute of operations is discarded in the mapping process. Although a non-blocking form of method invocation could be found in java (by creating a new thread for each call), oneway method invocation has not been introduced to DCUP yet, and some synchronization faults could appear during updates, if oneway calls were used without proper analysis.

Private interfaces

For IDL interfaces with requirements (or, generally, private attributes, but currently, the only instance of private attributes are requirements), a “private” interface is generated. This interface extends the regular interface, and contains methods for accessing private attributes. For requirements, only methods for setting the requirement’s value are provided, from outside the object, it is not necessary and not even suitable to read that value.

Wrapper objects

Wrapper objects are used to mediate access from outside a component to an implementation object.

A wrapper object class is generated for every IDL interface defined. The wrapper object implements the language-mapped interface for that IDL interface, plus `WrapperInterface`, which contains methods common to all wrapper objects.

The wrapper implements every method of the language-mapped interface, by calling the same method on the target object. If the method returns references to objects (either by its return value, or via output parameters), this reference is translated to reference to a newly created wrapper, having the original return value as its target.

Holder objects

Holder objects are generated for every defined IDL type, which is used as an output parameter. The Holder object contains no methods and only a single public attribute named `value`, of type equal to the type for which the holder object is generated.

3.6.3 DCUP code for components

For every component defined, a component manager and a component builder are generated.

Component manager

The component manager does not depend on the architecture section of the components declaration.

Although the amount of tasks a component manager does is huge, and the component manager class is very large, there are very few differences between component manager classes generated for different components. The only part of component manager depending on the component declaration are the `getRequirements` method, which returns the list of component's requirements, and the constructor, which, besides other tasks, registers all services provided by the component. These service need to be bound by the component builder, at the time of registering, the wrappers' targets point to `null`, and any call to these services would hold until a the service is bound.

Component builder

The component builder contains all the code representing the architecture section of an component. `inst` statements result into subcomponent creation code in the `onArrival` method, `implements` statements cause additional methods of the component builder to be generated, for the purpose of providing values to implementation objects' requirements, and `bind` statements result into possibly very complex code generated in either the `onArrival` or `provideRequirements` methods.

Also, the component builder contains `public static final` fields holding component version information, including identification of the component provider.

3.6.4 Modifications of generated code required from the user

Most of the code generated can be used right as it is generated. The only exception is binding of implementation object, which takes part "at the border of architecture description area", and some information necessary for completing the binding is not included in the architecture description, and therefore, not available to the CDL compiler. Here, user has to provide some code.

1. `provide_<interfacename>_<requirementname>` methods of component builder.

These methods are designed to be an exchange point between the component builder and implementation objects. These methods will be defined for every implementation object's requirement, and will be called to provide references for that requirement. This method will be called once for

every value provided for that requirement (that is, once for each parameter of an **one_of** clause, with respect to value generator rules), and it is up to the user to select correct values for each implementation object from the values provided, and to deliver these references to the implementation objects.

2. bindings to local implementation objects

For every binding using the **implementation_of** clause, the user has to supply a reference to an implementation object. The user has to edit the generated code, and in the `onArrival` method, supply the reference at the proper place, which is clearly marked.

To avoid omissions, the code is generated uncompileable, if the user does not modify the place, where the reference is to be supplied, compiler will return a syntax error.

For the above described situations, user was required to modify the code. But, optionally, user code may be entered at several other places, especially in the component builder's `onArrival` method. The range of architectures, which can be described in an ADL, is small, and some users' intentions cannot expressed in any other way than in code written in the implementation language doing what the user wants. To allow, e.g., dynamicity and creation of new subcomponents on conditions defined by the user, there's no other way than letting the user enter code directly into the component builder. Therefore, user code may appear anywhere in generated code.

3.7 Summary

The description of both syntax and semantics of the CDL language proposed by this thesis has been given here. The detail level of provided here should be sufficient for a reader to understand the basic concepts forming the CDL, including the new paradigms of an interface representing an unlimited number of implementation objects instantiated in a component, and the value generators used in bindings.

The detail level might not be sufficient for a full understanding of the language syntax. Additional sources of information are provided as appendices of this thesis, the CDL grammar as Appendix A, and the CDL compiler with examples on a diskette attached to the thesis.

Chapter 4

Current status and future work

The design of the *Component definition language* has been brought to a consistent state, and is presented in this thesis. The language allows any reasonable architecture to be expressed, and the architecture description possesses a high level of readability and clarity.

CDL allows the architecture description to be relieved from information about implementation objects, but still contain bindings for services required by those implementation objects. This is accomplished by the *interfaces' requirements*, and gives the architecture description a high readability and a strong verification tool at the same time. Also, readability is increased by using the *value generators* for bindings of arrays, instead of meta-code loops used in other architecture description languages.

Most of the aspects of the language have been implemented in compiler, although some parts, especially of architecture verification, are still to be implemented. For those parts not implemented yet, a clear idea of implementation exists, and no modification to the language will be needed during implementation.

As the research of the whole research group shall proceed, CDL is very likely to be revised. Versioning and version compatibility is an area open to research, and after a concept of versioning gets created, CDL will be extended for expressing version compatibility information. Also, possibly, the syntax for selecting component types by version might get altered.

Another field, on which attention could be focused, is verification. Additional verifications could be processed, and current verification algorithms could get improved.

Also, in the area of code generation, a mechanism for keeping changes made by the user to the generated code might be established, to transfer the changes to the newly generated code, in case of recompilation and code re-generation.

Although DCUP is currently implemented only for java environment, it can be implemented for other environments (a CORBA implementation is planned). Then, support for the new environment will have to be added to the CDL compiler, and code generation modules will have to be developed. The compiler is developed target environment independent, and this extension should not be difficult.

Chapter 5

Conclusion

The goal of the thesis, as it was described in section 1.2, has been reached. A language suitable for describing components and architectures of component based applications has been designed. In the design of the language, many questions had to be answered, in many situations, a decision had to be made. One of the main problems solved during working on the thesis were the implementation objects. Although they're should not be a part of the architecture description, the language should provide a way to ensure, that services required by the objects will be available in the component where the objects will be instantiated. This problem has been solved in a more than satisfactory way, by introducing the approach to represent an unlimited amount of implementation objects with a single interface shared by all the objects, holding information about services required by these objects.

Also, bindings between arrays of objects have been solved in an inventive way by introducing the *value generators*, and binding statements have been made much easier to read.

The language has been implemented in the CDL compiler, which can read a architecture description written in CDL, apply several verifications on the architecture, and in case that the architecture is correct, generate source code for all the control objects needed by DCUP.

While working on the thesis, I learned a lot about object-oriented and component-oriented programming, the CORBA environment, and other topics related to distributed and object oriented systems. I also gained some technical skills, e.g. in using bison for generating LALR(1) parsers, which is used for analyzing the CDL source files.

Bibliography

- [AG94a] R. Allen and G. Garlan, *Formal Connectors*, Technical Report, CMU-CS-94-115, Carnegie Mellon University, March 1994.
- [AG94b] R. Allen and G. Garlan, *Formalizing Architectural Connection*, In Proceedings of the Sixteenth International Conference on Software Engineering, pages 71-80, Sorrento, Italy, May 1994.
- [AdaIntro] S.J.Young: *An introduction to Ada*, Halsted Press Edition ISBN 0-470-20112-6
- [Bosch97b] J. Bosch: *Adapting Object-Oriented Components*, Proceedings of the 2nd Workshop on Component-Oriented Programming.
- [DarwinO] Darwin Overview,
<http://outoften.doc.ic.ac.uk/~regis/ftp/darwin.ps.gz>,
<http://www-dse.doc.ic.ac.uk/research/darwin/darwin.html>
- [DarwinRef] The Darwin Language Reference Manual,
<http://www-dse.doc.ic.ac.uk/research/darwin/darwin-lang.html>
- [IDLspec] OMG: *OMG IDL Syntax and Semantics*,
<ftp://www.omg.org/pub/docs/formal/98-02-08.ps>
- [JavaDoc] James Gosling, Bill Joy, Guy Steele: *The Java Language Specification — Documentation Comments*, Sun Microsystems,
<http://java.sun.com/docs/books/jls/html/18.doc.html>
- [LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann: *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, pages 336-355, April 1995.
- [LV95] D. C. Luckham and J. Vera *An Event-Based Architecture Definition Language*, IEEE Transactions on Software Engineering, pages 717-734, September 1995.
- [MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer: *Specifying Distributed Software Architectures*, In Proceedings of the Fifth European Software Engineering Conference (ESEC'95), Barcelona, September 1995.

- [MDK93] J. Magee, N. Dulay, and J. Kramer *Structuring Parallel and Distributed Programs*, IEE Software Engineering Journal, pages 73-82, March 1993.
- [MIC95] Microsoft Common Object Model Specification - Oct 24, 1995, Draft
- [MK95] J. Magee and J. Kramer: *Modelling Distributed Software Architectures*, In Proceedings of the First International Workshop on Architectures for Software Systems, pages 206-222, Seattle, WA, April 1995.
- [MK96] J. Magee and J. Kramer. *Dynamic Structure in Software Architectures*, In Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4), pages 3-14, San Francisco, CA, October 1996.
- [MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. *Using object-oriented typing to support architectural design in the C2 style*. In Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4), pages 24-32, San Francisco, CA, October 1996.
- [MTW96] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. *Formal Modeling of Software Architectures at Multiple Levels of Abstraction*. In Proceedings of the California Software Symposium 1996, pages 28-40, Los Angeles, CA, April 1996.
- [Med96] N. Medvidovic. *ADLs and Dynamic Architecture Changes*. In A. L. Wolf, ed., Proceedings of the Second International Software Architecture Workshop (ISAW-2), pages 24-27, San Francisco, CA, October 1996.
- [Medv97] Neno Medvidovic: *A Classification and Comparison Framework for Software Architecture Description Languages*, Technical Report UCI-ICS-TR-97-02, 1997
- [NDH97] Nguyen Duy Hoa, *Distributed Object Computing with TINA and CORBA*, Tech Report No. 97/7, Charles University, Prague, June 1997
- [OMG94] OMG 94-10-09, *Common Object Model Specification*
- [OMG95] OMG 95-03-34: *Specification Languages for Component-Based Software Engineering*
- [OMG96] OMG 96-03-04, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1995
- [OrbixWeb] IONA Technologies: *OrbixWeb Reference Guide*, Chapter 5 — IDL to Java Mapping

- [PBJ97] F. Plasil, D. Balek, R. Janecek: *DCUP: Dynamic Component Updating in Java/CORBA Environment*, Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague.
- [PBJ98a] F. Plasil, D. Balek, R. Janecek: *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, Accepted for presentation at ICCDS'98, May 4-6, 1998, Annapolis, Maryland, USA
- [Rap96] Rapide Design Team: *Rapide 1.0 Language Reference Manual*, Program Analysis and Verification Group, Computer Systems Lab, Stanford University, January 1996.
- [Regis] J. Magee, N. Dulay, J. Kramer: *Regis: A Constructive Development Environment for Distributed Programs*, In *Distributed Systems Engineering Journal*, September 1994
- [SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik: *Abstractions for Software Architecture and Tools to Support Them*, *IEEE Transactions on Software Engineering*, pages 314-335, April 1995.
- [SDZ96] M. Shaw, R. DeLine, and G. Zelesnik: *Abstractions and Implementations for Architectural Connections*, In *Proceedings of the Third International Conference on Configurable Distributed Systems*, May 1996.
- [TINA96b] *TINA Object Definition Language Manual* , July 22nd 1996

Appendix A

CDL-grammar

```
rule 1    specification -> /* empty */
rule 2    specification -> specification definition
rule 3    specification -> specification architecture
rule 4    definition -> type_dcl ';'
rule 5    definition -> const_dcl ';'
rule 6    definition -> except_dcl ';'
rule 7    definition -> interface ';'
rule 8    definition -> module ';'
rule 9    definition -> template ';'
rule 10   template -> TEMPLATE_KEYWORD IDENTIFIER
           '{' template_body '}'
rule 11   template_body -> /* empty */
rule 12   template_body -> template_body t_body
rule 13   t_body -> provisions
rule 14   t_body -> requirements
rule 15   t_body -> property_dcl
rule 16   property_dcl -> PROPERTY_KEYWORD IDENTIFIER ';'
rule 17   provisions -> PROVIDE_KEYWORD ':' provision_dcls
rule 18   provision_dcls -> provision_dcl
rule 19   provision_dcls -> provision_dcls provision_dcl
rule 20   provision_dcl -> scoped_name simple_declarator ';'
rule 21   requirements -> REQUIRE_KEYWORD ':' requirement_dcls
rule 22   requirement_dcls -> requirement_dcl
rule 23   requirement_dcls -> requirement_dcls requirement_dcl
rule 24   requirement_dcl -> scoped_name simple_declarator ';'
rule 25   architecture -> ARCHITECTURE_KEYWORD IDENTIFIER
           scoped_name VERSION_KEYWORD STRING_LITERAL
           '{' arch_body '}' ';'
rule 26   arch_body -> /* empty */
rule 27   arch_body -> arch_body arch_statement
rule 28   arch_statement -> instantiation
rule 29   arch_statement -> binding
rule 30   arch_statement -> IMPLEMENTATION_VERSION_KEYWORD
           STRING_LITERAL ';'
rule 31   arch_statement -> IMPLEMENTS_KEYWORD scoped_name ';'
rule 32   instantiation -> inst_auto INST_KEYWORD IDENTIFIER
           ':' scoped_name VERSION_KEYWORD
           STRING_LITERAL IDENTIFIER ';
```

```

rule 33  inst_auto -> /* empty */
rule 34  inst_auto -> AUTO_KEYWORD
rule 35  bind_element_reference -> IDENTIFIER sub_references
rule 36  expr -> const_exp
rule 37  value_generator_elements -> value_generator_element
rule 38  value_generator_elements -> value_generator_elements
        ',' value_generator_element
rule 39  value_generator_element -> expr
rule 40  value_generator_element -> expr DOT_DOT expr
rule 41  value_generator -> '<' '>'
rule 42  value_generator -> '<' value_generator_elements '>'
rule 43  sub_references -> /* empty */
rule 44  sub_references -> sub_references sub_reference
rule 45  sub_reference -> '.' IDENTIFIER
rule 46  sub_reference -> '[' expr ']'
rule 47  sub_reference -> '[' value_generator ']'
rule 48  bindee -> bind_element_reference
rule 49  bindee -> bind_element_reference ':' bind_element_reference
rule 50  lhs_bindee -> scoped_name '*' bind_element_reference
rule 51  lhs_bindee -> bindee
rule 52  rhs_bindee -> ONE_OF_KEYWORD '(' bindees ')'
rule 53  rhs_bindee -> rhs_i_bindee
rule 54  bindees -> bindees rhs_i_bindee
rule 55  bindees -> rhs_i_bindee
rule 56  rhs_i_bindee -> IMPLEMENTATION_OF_KEYWORD scoped_name
rule 57  rhs_i_bindee -> bindee
rule 58  binding -> BIND_KEYWORD lhs_bindee TO_KEYWORD rhs_bindee ';'
rule 59  @1 -> /* empty */
rule 60  module -> MODULE_KEYWORD IDENTIFIER @1 '{' module_body '}'
rule 61  module_body -> definition
rule 62  module_body -> module_body definition
rule 63  interface -> interface_dcl
rule 64  interface -> forward_dcl
rule 65  interface_dcl -> interface_header '{' interface_body
        interface_requirements '}'
rule 66  interface_requirements -> /* empty */
rule 67  interface_requirements -> requirements
rule 68  forward_dcl -> INTERFACE_KEYWORD IDENTIFIER
rule 69  interface_header -> INTERFACE_KEYWORD IDENTIFIER
        inheritance_spec
rule 70  interface_body -> /* empty */
rule 71  interface_body -> interface_body export
rule 72  export -> type_dcl ';'
rule 73  export -> const_dcl ';'
rule 74  export -> except_dcl ';'
rule 75  export -> attr_dcl ';'
rule 76  export -> op_dcl ';'
rule 77  inheritance_spec -> /* empty */
rule 78  inheritance_spec -> ':' inheritance_list
rule 79  inheritance_list -> scoped_name
rule 80  inheritance_list -> inheritance_list ',' scoped_name
rule 81  scoped_name -> IDENTIFIER
rule 82  scoped_name -> TWO_DOTS IDENTIFIER

```

```
rule 83  scoped_name -> scoped_name TWO_DOTS IDENTIFIER
rule 84  const_dcl -> CONST_KEYWORD const_type IDENTIFIER
           '=' const_exp
rule 85  const_type -> integer_type
rule 86  const_type -> char_type
rule 87  const_type -> boolean_type
rule 88  const_type -> floating_pt_type
rule 89  const_type -> string_type
rule 90  const_type -> scoped_name
rule 91  const_exp -> or_expr
rule 92  or_expr -> xor_expr
rule 93  or_expr -> or_expr '|' xor_expr
rule 94  xor_expr -> and_expr
rule 95  xor_expr -> xor_expr '^' and_expr
rule 96  and_expr -> shift_expr
rule 97  and_expr -> and_expr '&' shift_expr
rule 98  shift_expr -> add_expr
rule 99  shift_expr -> shift_expr SHIFT_RIGHT add_expr
rule 100 shift_expr -> shift_expr SHIFT_LEFT add_expr
rule 101 add_expr -> mult_expr
rule 102 add_expr -> add_expr '+' mult_expr
rule 103 add_expr -> add_expr '-' mult_expr
rule 104 mult_expr -> unary_expr
rule 105 mult_expr -> mult_expr '*' unary_expr
rule 106 mult_expr -> mult_expr '/' unary_expr
rule 107 mult_expr -> mult_expr '%' unary_expr
rule 108 unary_expr -> unary_operator primary_expr
rule 109 unary_expr -> primary_expr
rule 110 unary_operator -> '-'
rule 111 unary_operator -> '+'
rule 112 unary_operator -> '~'
rule 113 primary_expr -> scoped_name
rule 114 primary_expr -> literal
rule 115 primary_expr -> '(' const_exp ')'
rule 116 primary_expr -> '$' typename IDENTIFIER
rule 117 typename -> /* empty */
rule 118 typename -> '(' base_type_spec ')'
rule 119 literal -> INTEGER_LITERAL
rule 120 literal -> STRING_LITERAL
rule 121 literal -> CHARACTER_LITERAL
rule 122 literal -> FLOATING_PT_LITERAL
rule 123 literal -> boolean_literal
rule 124 boolean_literal -> TRUE
rule 125 boolean_literal -> FALSE
rule 126 positive_int_const -> const_exp
rule 127 type_dcl -> TYPEDEF_KEYWORD type_declarator
rule 128 type_dcl -> struct_type
rule 129 type_dcl -> union_type
rule 130 type_dcl -> enum_type
rule 131 type_declarator -> type_spec declarators
rule 132 type_spec -> simple_type_spec
rule 133 type_spec -> constr_type_spec
rule 134 simple_type_spec -> base_type_spec
```

```

rule 135 simple_type_spec -> template_type_spec
rule 136 simple_type_spec -> scoped_name
rule 137 base_type_spec -> floating_pt_type
rule 138 base_type_spec -> integer_type
rule 139 base_type_spec -> char_type
rule 140 base_type_spec -> boolean_type
rule 141 base_type_spec -> octet_type
rule 142 base_type_spec -> any_type
rule 143 template_type_spec -> sequence_type
rule 144 template_type_spec -> string_type
rule 145 constr_type_spec -> struct_type
rule 146 constr_type_spec -> union_type
rule 147 constr_type_spec -> enum_type
rule 148 declarators -> declarator
rule 149 declarators -> declarators ',' declarator
rule 150 declarator -> simple_declarator
rule 151 declarator -> complex_declarator
rule 152 simple_declarator -> IDENTIFIER
rule 153 complex_declarator -> array_declarator
rule 154 floating_pt_type -> FLOAT
rule 155 floating_pt_type -> DOUBLE
rule 156 integer_type -> signed_int
rule 157 integer_type -> unsigned_int
rule 158 signed_int -> signed_long_int
rule 159 signed_int -> signed_short_int
rule 160 signed_long_int -> LONG
rule 161 signed_short_int -> SHORT
rule 162 unsigned_int -> unsigned_long_int
rule 163 unsigned_int -> unsigned_short_int
rule 164 unsigned_long_int -> UNSIGNED LONG
rule 165 unsigned_short_int -> UNSIGNED SHORT
rule 166 char_type -> CHAR
rule 167 boolean_type -> BOOLEAN
rule 168 octet_type -> OCTET
rule 169 any_type -> ANY
rule 170 struct_type -> STRUCT_KEYWORD IDENTIFIER '{' member_list '}'
rule 171 member_list -> member
rule 172 member_list -> member_list member
rule 173 member -> type_spec declarators ';'
rule 174 union_type -> UNION IDENTIFIER SWITCH '('
    switch_type_spec ')' '{' switch_body '}'
rule 175 switch_type_spec -> integer_type
rule 176 switch_type_spec -> char_type
rule 177 switch_type_spec -> boolean_type
rule 178 switch_type_spec -> enum_type
rule 179 switch_type_spec -> scoped_name
rule 180 switch_body -> case
rule 181 switch_body -> switch_body case
rule 182 case -> case_label_part element_spec ';'
rule 183 case_label_part -> case_label
rule 184 case_label_part -> case_label_part case_label
rule 185 case_label -> CASE_KEYWORD const_exp ':'
rule 186 case_label -> DEFAULT_KEYWORD ':'

```

```

rule 187 element_spec -> type_spec declarator
rule 188 enum_type -> ENUM_KEYWORD IDENTIFIER '{' enumerators '}'
rule 189 enumerators -> IDENTIFIER
rule 190 enumerators -> enumerators ',' IDENTIFIER
rule 191 sequence_type -> SEQUENCE_KEYWORD '<' simple_type_spec
        ',' positive_int_const '>'
rule 192 sequence_type -> SEQUENCE_KEYWORD '<' simple_type_spec '>'
rule 193 string_type -> STRING '<' positive_int_const '>'
rule 194 string_type -> STRING
rule 195 array_declarator -> IDENTIFIER array_size
rule 196 array_size -> fixed_array_size
rule 197 array_size -> array_size fixed_array_size
rule 198 fixed_array_size -> '[' positive_int_const ']'
rule 199 attr_dcl -> READONLY_KEYWORD attribute
rule 200 attr_dcl -> attribute
rule 201 attribute -> ATTRIBUTE_KEYWORD param_type_spec
        attr_declarator
rule 202 attr_declarator -> simple_declarator
rule 203 attr_declarator -> attr_declarator ',' simple_declarator
rule 204 except_dcl -> EXCEPTION_KEYWORD IDENTIFIER '{' members '}'
rule 205 members -> /* empty */
rule 206 members -> members member
rule 207 op_dcl -> operation
rule 208 op_dcl -> op_attribute operation
rule 209 operation -> op_body
rule 210 operation -> op_body raises_exp
rule 211 operation -> op_body context_exp
rule 212 operation -> op_body raises_exp context_exp
rule 213 op_body -> op_type_spec IDENTIFIER parameter_dcls
rule 214 op_attribute -> ONEWAY_KEYWORD
rule 215 op_type_spec -> param_type_spec
rule 216 op_type_spec -> VOID
rule 217 parameter_dcls -> '(' params ')''
rule 218 params -> /* empty */
rule 219 params -> param_dcl
rule 220 params -> params ',' param_dcl
rule 221 param_dcl -> param_attribute param_type_spec
        simple_declarator
rule 222 param_attribute -> IN_KEYWORD
rule 223 param_attribute -> OUT_KEYWORD
rule 224 param_attribute -> INOUT_KEYWORD
rule 225 raises_exp -> RAISES_KEYWORD '(' raises_list ')''
rule 226 raises_list -> scoped_name
rule 227 raises_list -> raises_list ',' scoped_name
rule 228 context_exp -> CONTEXT_KEYWORD '(' context_list ')''
rule 229 context_list -> STRING_LITERAL
rule 230 context_list -> context_list ',' STRING_LITERAL
rule 231 param_type_spec -> base_type_spec
rule 232 param_type_spec -> string_type
rule 233 param_type_spec -> scoped_name

```