Automatic Synthesis of Deadlock free connectors for COM/DCOM Applications

Paola Inverardi
University of L'Aquila
Dip. Matematica Pura ed Applicata
via Vetoio 1, 67100 L'Aquila, Italy
inverard@univaq.it

Massimo Tivoli
University of L'Aquila
Dip. Matematica Pura ed Applicata
via Vetoio 1, 67100 L'Aquila, Italy
tivoli@univaq.it

ABSTRACT

Many software projects are based on the integration of independently designed software components that are acquired on the market rather than developed within the project itself. Sometimes interoperability and composition mechanisms provided by component based integration frameworks cannot solve the problem of binary component integration in an automatic way. Notably, in the context of component based concurrent systems, the binary component integration may cause deadlocks within the system. In this paper we present a technique to allow connectors synthesis for deadlock-free component based architectures [2] in a real scale context, namely in the context of COM/DCOM applications. This technique is based on an architectural, connector-based approach which consists of synthesizing a COM/DCOM connector as a COM/DCOM server that can route requests of the clients through a deadlock free policy. This work also provides guide lines to implement an automatic tool that derives the implementation of routing deadlock-free policy within the connector from the dynamic behavior specification of the COM components. It is then possible to avoid the deadlock by using COM composition mechanisms to insert the synthesized connector within the system while letting the system COM servers unmodified. We present a successful application of this technique on the (COM version of the) problem known as *"The dining philosophers"*. Depending on the type of deadlock we have a strategy that automatically operates on the connector part of the system architecture in order to obtain a suitably equivalent version of the system which is deadlock-free.

1. INTRODUCTION

Component based integration frameworks (COM/ DCOM, Sun's JavaBeans, CORBA) provide interoperability and composition mechanisms that cannot solve all the problems of binary component integration in an automatic way. Notably, in the context of component based concurrent systems, the binary component integration may cause deadlocks within the system [6]. In this paper we present a technique to allow connectors synthesis for deadlock-free component based architectures [2] in a real scale context, namely in the context of COM/DCOM. Recently, software architecture has emerged as an area that places significant importance on components interaction. In the software architecture domain, components interaction is embodied in the notion of software connectors. Software connectors mediate interactions among components [8, 11]. For example in the context of distributed systems, connectors manifest themselves in a software system as shared variables, procedure calls, pipes, SQL links between a database and an application. Typically in the context of component based architectures, people distinguish components (computation) from connectors (interaction) in a system. Connectors are often considered to be explicit at the architectural level, but intangible in a system implementation. In a COM/DCOM architecture, we should consider the following mechanisms as software connectors: RPC, marshaling/unmarshaling, Windows Registry, binding and the COM architecture itself with all interoperability and location mechanisms. Sometimes connectors are also deliberately modelled as components further obscuring their distinct nature. Special purpose Architecture Description Languages (ADL) have been devised to provide support for modelling more sophisticated and powerful connectors. A predominant focus of existing ADLs has been on verifying the properties of the modelled behavior. In this paper we assume an explicit notion of connector, although we will model it as a component in the COM context. Thus a COM connector will be a component easily insertable within the system by using a COM composition mechanism. Since we aim at proving properties about the dynamics of the composed system, i.e. deadlock freeness, we also need to suitably extend the *Microsoft Interface Definition Language* (MIDL) with information on the dynamic behavior of the single components. Our approach is based on the theoretical framework introduced in [2]. The idea there is to build applications by assuming a defined architectural style, namely a modified version of the C2 architectural style [7]. The method starts off a set of components, and builds a connector following the reference style constraints. Components are enriched with additional information on their dynamic behavior which takes the form of graphs. Then deadlock analysis is performed. If the synthesized connector contains deadlock behaviors, these are removed. Depending on the kind of deadlock, this is enough to obtain a deadlock-

ESEC/FSE 2001, Vienna, Austria
© ACM 2001 1-58113-390-1/01/09…$5.00

free version of the system. Otherwise, the deadlock is due to some component internal behavior and cannot be fixed without directly operating on the component. This work also provides guide lines to implement an automatic tool that derives the implementation of a routing deadlock-free policy within the connector from the dynamic behavior specification of the COM components. This technique avoids the deadlock by using COM composition mechanisms to insert the synthesized connector within the system while letting the system COM servers unmodified. At present our technique requires the system COM clients to be slightly modified. In Section 7, we mention a way to avoid the clients code updating. Applying the rules that we provide at the end of Section 7, the technique avoids the deadlock letting all the system COM components unmodified.

The paper is organized as follows. In Section 2 we introduce the problem, in Section 3 we present some notions that are important to understand the paper and in Section 4 we describe the COM/DCOM model with respect to COM component interoperability, composition, evolution and communication. Section 5 presents the technique to allow connectors synthesis [2] for a COM/DCOM single-layered application and for a COM/DCOM multi-layered application. Section 6 presents an application of connectors synthesis for a COM single-layered system; this system implements an instance of the problem known as *The dining philosophers* [14] with two philosophers and two forks. In Section 8 we present conclusions and possible extensions to the techniques discussed in this paper, in order to work in a more general COM/DCOM context and to leave all the system COM components unmodified. It's worth noticing that at present DCOM does not exist anymore and it has been subsumed by COM+. Thus all we present can be applied in the context of COM/COM+ as well.

2. PROBLEM DESCRIPTION

In [2] the following definition of the deadlock problem in a component based context is reported:

Definition 1. A set of components is deadlocked if each component in the set is waiting for an event that only a component in the set can cause.

In that paper there is also a separation of deadlock problems in two classes:

- deadlocks caused by a wrong coordination policy (such as *The dining philosophers*);
- deadlocks caused by components internal behavioral assumptions (such as *Compressing Proxy* [3]).

The problems of the first class can be solved through deadlock detection and recovery. The problems of the second class can only allow for deadlock detection. We develop an automatic technique to connectors synthesis as discussed in [2] in a COM/DCOM context. In order to do this we proceed as follows: first we must understand how the COM model can reflect the reference architectural style in [2]; this problem is treated in Section 5. Then the connectors synthesis assumes that each component is represented with an ACtual behavior graph (*AC-Graph*) that describes dynamic behavior of the component [3, 4]. From these automata we can obtain the connector transition graph. Thus we must find a way to accomodate the notion of *AC-Graph* in the COM component interface description (MIDL code); this problem is treated in Section 5.1 by extending the MIDL code with CCS [9] statement. In Section 5.1 we provide a general CCS model for a server side request and a client side request. The next step is to understand how we should build a COM connector from an architectural point of view. In Section 5 the COM connector has been synthesized as a COM server that contains all servers of the connector free system. This server provides atomic requests and his interface represents the union of all services provided by all servers within the connector itself. At this point it is possible to use the connectors synthesis discussed in [2] to obtain the COM connector transition graph. This graph represents a model of the right routing policy of the requests. The last problem is to understand how we can obtain the implementation of a connector service simply by observing the structure of the connector transition graph; this problem is treated in Section 7.

3. BACKGROUND

The reference architectural style discussed in [2], called *Connector Based Architecture* (CBA), is derived from the C2 style [7] and [15]; the main characteristics of this style are:

- component based;
- scalability;
- two types of elements: components and connectors;
- both components and connectors have defined *top* and *bottom*;
- the *top* is connected to the *bottom* of a single connector;
- the *bottom* is connected to the *top* of a single connector;
- there is no bound on the number of components or connectors that may be attached to a single connector;
- two types of messages: *request* from *bottom* to *top* and *notification* from *top* to *bottom*;
- the *top* domain specifies the set of notifications a component responds to, and the set of requests it emits upward in the architecture;
- the *bottom* domain specifies the set of notifications that a component forwards down in the architecture, and the set of requests it responds to;

The precedent items represent characteristics that we find also in the C2 style. The main differences between C2 style and CBA style are:

- synchronous message passing;

- connectors cannot directly communicate;
- connectors are only routing devices, without any filtering policies;
- connectors have a strictly sequential input-output behavior.

The precedent items represent the CBA style distinguished characteristics that we do not find in C2 style. In the following Section we present the COM/DCOM features that we will use in Section 5 to model the characteristics of the CBA style.

4. COM MODEL ARCHITECTURE

A COM/DCOM application is based on a client/server organization. A client of a component is any program that can execute component code. To allow clients to access its functions, a COM/DCOM server implements one or more interfaces. COM/DCOM supports multiple interfaces of a component. An interface is a binary structure in the address space of a component, whose layout is defined as a table of pointers to functions. A client refers with a pointer to an interface. The specification of all functions that can be called through an interface is defined by the type of that interface. A type may inherit from another type by extending its list of function specifications. All interface types are organized in a single inheritance hierarchy with a special type *IUnknown* as root. Typically, COM/DCOM does take Microsoft version of the DCE IDL (MIDL) as preferred notation to define an interface type. The DCE/IDL and MIDL are two different languages. MIDL is a Microsoft's extension of the standard DCE/IDL. Because *IUnknown* is the root of the interface type hierarchy, its operations are accessible through any interface. Thus, having a pointer to an interface of any component enables calls to the *IUnknown* methods: *QueryInterface*, *AddRef* and *Release*. The *QueryInterface* provides the primitive to navigate between component interfaces. The latter two operations maintain reference counts on interfaces, enabling components to control their lifetimes and the lifetimes of individual interface tables. The *QueryInterface* operation is responsible for dynamic interface negotiation: it is an interoperability and introspection mechanism. It provides an introspection service through which it is possible to make sure at runtime that the server interfaces are of the type clients need. The implementation of all functions that can be called through an interface are defined by a COM class. A client can invoke a server method using information contained in the *Type Library*. A *Type Library* specification is defined by MIDL code. A COM client has all the information to use a COM server after the MIDL compiler has processed the *Type Library* code and the marshaling/unmarshaling code defined in the server MIDL file. A COM server is represented by a server object which is the instance of a given COM class. COM imposes several rules on how *QueryInterface* must behave:

- **time-invariance**: the set of interface types that a component exposes through *QueryInterface* does not change at runtime;
- **reachability**: if a component exposes a given type of interface, that type of interface should be accessible from any interface of the component;
- **component identity**: two interfaces belong to the same component if and only if the pointers returned by calls to *QueryInterface* on these interfaces for *IUnknown* universal unique identifier are equal.

COM defines two approaches to use existing components as building blocks of new ones:

- **containment/delegation**: an outer component encapsulates one or more inner components and uses their services. It is a general mechanism in object oriented programming but it adds overhead on the performance of a request;
- **aggregation**: the outer component can directly expose to its clients its pointer to the inner interface. Clients use this pointer as it is a pointer to an interface of the outer component. There is no overhead on the performance of a request. Thus, an inner component is said to be *aggregated* if an outer component exposes pointers to one or more of its (inner) interfaces. Actually in order for an inner component to be accessible, the outer component must expose all the interfaces of the inner one. This happens because otherwise clients of the outer component would be able to observe a component that does not follow the reachability rule of COM interface negotiation, in violation of the standard. For example, a client could require an inner interface which is not exposed from the outer object.

Both these mechanisms are designed so that the internal structure of a composite component is hidden from its clients. In the literature there are many papers that analyze the conflicts between aggregation and interface negotiation in COM [12, 5, 13]. In general, the COM interface negotiation mechanism fails to correctly discover interfaces of some types exposed by an aggregated component, and to distinguish the identities of aggregated components within the same aggregate. The problem arises because outer and inner components share an interface. The problem is that this kind of sharing, which is the essence of aggregation, does not work well with interface negotiation. So the aggregation is not a general composition mechanism.

A client and a server communicate to each other through a procedure call mechanism (RPC). Using this mechanism we can say that a client connects itself to a server through a unique type of communication channel that we have called PCC (***P**rocedure **C**all **C**hannel*). A channel of this type can connect a client to a server as well as a server to a server by using the containment/delegation mechanism. We assume that the requests on PCC are always synchronous. In COM+ it is possible to define asynchronous interfaces but that procedure call mechanism is not really an asynchronous mechanism. When a client invokes a service of a server object, which implements an asynchronous interface, the request invocation follows an asynchronous communication model but when the method returns the control to the client it follows a synchronous communication model. Hence in COM+ asynchronous calls are simulated by synchronous mechanisms.

A COM server has four types of threading models [1] and [10]:

- **single**: in any process there is only one thread. There is an unique thread that manages all objects created by the thread itself;
- **apartment**: in any process there are one or more threads. A thread manages one or more objects, but a server does not execute at the same time two requests coming from two different threads;
- **free**: in any process there are one or more threads. A thread manages one or more objects, and a server can execute at the same time two requests from two different threads;
- **both**: **apartment** + **free**.

The services provided by a COM server have a parameters list. Any parameter in the list has a given passing mode defined by IDL code. There are two types of passing modes: [*in*] and [*out*]. [*in*] means that a parameter is an input parameter for the server on PCC. The services provided by a COM server also have a return value. The type of this value is called **HRESULT**. The return value is considered, from the server, as an output element on PCC. The **HRESULT** type assumes a unique success value (**S_OK**) and a given set of error values (**E_...**). Typically if a request returns **S_OK** it means that the server has provided to the client the service the client requested for, so the server is in a state in which it expects a different request. If the server returns some **E_...** value it means that it has not provided to the client the requested service, so the server is in a state in which he expects the same request again.

5. COM CONNECTORS SYNTHESIS

In sections 3 and 4 we observed that COM/DCOM like CBA is component based and its building elements are components and connectors. The difference is that while in COM/DCOM the connectors are auxiliary mechanisms provided by the model itself, in CBA the connectors are explicit entities at the system implementation level. We will implement an explicit COM connector as a composite server that provides a coordination service and that it is a tangible entity at system implementation level. We map the concepts of *top* and *bottom* of a CBA component with the concepts of client side and server side of a COM component respectively. We use the general composition mechanism called containment/delegation. We can then synthesize a COM connector as a server that contains the servers of the connector free system and that delegates to these servers the requests of the clients using a deadlock-free routing policy. The connector interface is the union of all interfaces of the servers contained within the connector itself. In this way we reflect the composition rules of a CBA system. Furthermore, COM/DCOM like CBA has a synchronous communication model. The COM/DCOM communication model reflects the CBA communication model by assuming the following mapping: a CBA request maps to the request of a COM client (marshaling), a CBA notification to the modification of return values (unmarshaling) and to the change of the server state, the *top* domain to the set of IID specified in the *QueryInterface* calls and the *bottom* domain to the MIDL code for a *Type Library*. Then we can also say that the COM/DCOM connector has a strictly sequential input-output behavior (such as a CBA connector) introducing in the architecture the following restrictions: the connector threading model is only the apartment (atomic request) and the implementations of the requests are single-threaded for each server of the system. Summarizing the architectural requirements of a COM coordination connector are:

1. server with apartment threading model;
2. composite server through containment/delegation mechanism;
3. outer interface as union of inner interfaces;
4. connect itself to the other components through synchronous procedure call channels (PCC).

5.1 COM servers and clients as CCS processes

In the introduction we mentioned that our approach assumes to enhance a component interface with dynamic information. We model the component dynamic behavior as a graph. The textual representation of this graph is given by using a CCS-like process algebra [9] which differs from CCS only for the use of slightly different syntactic notations. A graph is therefore represented as a CCS process. For the purposes of this paper, no specific knowledge of CCS is required. It is enough to know that the CCS processes we will use are defined as (possibly recursive) equations and define in a very intuitive way a graph, as we will informally explain later on. CCS processes can then be composed in parallel and they communicate synchronously. A more comprehensive explanation of the use of CCS in the component-based setting we are using can be found in [2, 3].

We associate a CCS process to each COM server and client. From restrictions imposed in Section 5 a server process is single-threaded; a client process may be made of one or more parallel execution threads. Intuitively, we associate a CCS process to each execution thread, the whole system will then be the parallel composition of the CCS processes associated to the clients and servers execution threads. The CCS process of the server is defined in terms of the requests for a service that the server object might receive. The CCS process of the client is defined in terms of the requests that the client might send to the server object. Informally, for a server we associate in the corresponding CCS process requests to input actions and notifications to output actions. The symmetric situation applies to clients. Thus we provide a general CCS model for a server side request and for a client side request in order to define the CCS process associated to a server or to a client. A server side request travels across PCC following two directions: the server object receives, in input, the request from the client (CBA request) and it sends in output a notification (CBA notification). The actual output action on PCC for the server is modelled as the result of a non-deterministic composition of output actions. This models the fact that it might receive a request from a client for any of his services. Below is the CCS schema for a service request server side. The first line defines a request (Req) for an exposed service. The string $!SrvId$ stands for an input action (!), to the server Srv, of the service Id. The string $SrvId$ represents one term that we parse by the following pattern: $[A|..|Z][0|..|9]+$. $[A|..|Z]$ is the letter for the substring Srv and $[0|..|9]+$ is the number for the substring Id. The whole term represented by $SrvId$ is build by the concatenation of the two substrings Srv and Id. This action is followed by a notification action (Not), where . means action prefixing. The second line reads analogously, but in addition uses the non-deterministic operator + to model the fact that the server can notify any request for one of its services. ? denotes that these are output actions. Note that we use !, ? instead of the CCS notation, action - coaction.

In Section 6 we will see an instantiation of these schemas.

$Req = !SrvId.Not$
$Not = ?SrvId_1.Req_1 + .. + ?SrvId_n.Req_n$

where:

- $Req, Req_1, .., Req_n, Not \in \{X, Y, ..\}$ are process variables
- Srv ranges over upper case letters (process variable associated to server)
- $Id, Id_1, .., Id_n$ range over positive integers (id IDL of the services provided by server)
- $!SrvId$ ranges over In that is the set of input actions on PCC
- $?SrvId_1, .., ?SrvId_n$ range over Out that is the set of output actions on PCC
- $Vis = In \cup Out$ is the set of visible actions on PCC
- $Act = Vis \cup \{tau\}$ is the set of all actions where tau denotes the so-called internal action

We define the dynamic behavior of a COM client in terms of the services **id** IDL that it can require to a server. Obviously if for a server a request is an input action and the notification is an output action for a client a request is an output action and the notification is an input action:

$Req = ?SrvId.Not$
$Not = !SrvId_1.Req_1 + .. + !SrvId_n.Req_n$

5.2 Connectors synthesis for a single-layered COM application

A single-layered COM architecture is a system composed of a set of server components that are not composite servers and of a set of exclusively client components. For an architecture of this type, we build a single connector which contains all the servers of the connector free system. The requests issued from a client will be sent to the connector. The MIDL code is extended by means of *commented* code representing the CCS process associated to the server. We also define the CCS processes associated to the execution threads of each client.

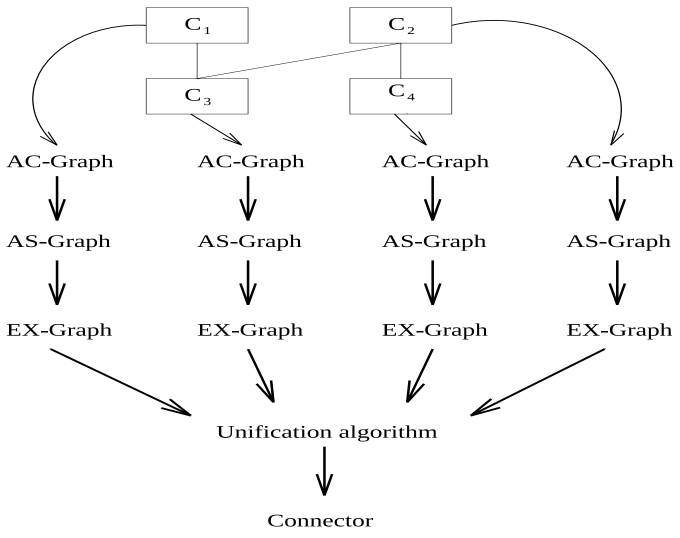

Figure 1: Connector synthesis

Referring to Figure 1, the technique defined in [2] starting from the component graph that models the component dynamics (AC-Graph), associates to a component a set of other graphs, which characterize different aspects of the component dynamics, from the actual component behavior to its assumptions on the environment. Referring to [2], we informally define the notion of Actual Behavior (AC) Graph for modelling component behavior. The term *actual* emphasizes the difference between component behavior and the intended, or assumed, behavior of the environment. AC graphs model components in an intuitive way. Each node represents a state of the component and the root node represents its initial state. Each arc represents the possible transition into a new state where the transition label is the action performed by the component. The strategy illustrated in [2], from the AC graphs derives the corresponding AS (ASsumption) graphs. These graphs describe the interaction behavior of each component with the external environment. First, we wish to derive from a component behavior the requirements on its environment that guarantee deadlock freedom. A system is in deadlock when it cannot perform any computation, thus in our setting, deadlock means that all components are blocked waiting for an action from the environment that is not possible. Given the way components are combined together, a component will not block if the environment can always provide the actions it requires for changing state. Thus we can informally define the notion of component assumption in the context of parallel composition and deadlock freedom by a graph (*AS-Graph*) which is different from the corresponding *AC-Graph* only in the arcs labels. In fact these labels are symmetric since they model the environment as each component expects it. Given the CBA style, the component environment can only be represented by one or more connectors, thus in [2] they refines the definition of AS-Graph in to a new graph, the EX-Graph, that represents the behavior that the component expects from the connector. We know that the connector performs strictly sequential input-output operations only, thus if it receives an input from a component it will then output the received input message to the destination component. Analogously, if the connector outputs a message, this means that immediately before it input that message. Intuitively, for each transition labelled with a visible action $!SrvId$ ($?SrvId$) in the AS graph, in the corresponding EX graph there are two strictly sequential transition labelled $!SrvId$ and $?SrvId_{unk}$ ($!SrvId_{unk}$ and $?SrvId$), respectively. The string unk denotes an action that the connector is expected to do on a communication channel which is not handled by the component we are deriving the *EX-Graph* for. Each component *EX-Graph* represents a partial view of the connector expected behavior. It is partial since it only reflects the expectations of a single component. The global connector behavior will be derived by taking into account all the *EX-graphs.* This will be done through an unification algorithm [2]. Informally, we attempt to matching known actions (terms) in a *EX-Graph* with unknown actions (variables) in another *EX-Graph.*

From the CCS annotations it is possible to build the *AC-Graphs* [2] for each component and run the following algorithm:

1. let K be the connector to build;
2. for each component C_i build the EX-Graph EX_i for C_i;
3. if it is impossible to unify the EX_i for each component C_i then $exit(FAILURE)$;
4. if sink nodes exist within transitions graph of K then delete the branches that terminate with these stop nodes;
5. for each component C_i if $CBSimulation(AS_i, CB_i)$ does not successfully terminate then $exit(FAILURE)$;
6. $exit(SUCCESS)$;

where:

AS_i is the *AS-Graph* [2] of the component C_i which is connected to the connector; CB_i is the *CB-Graph* [2] for C_i. Referring to [2] this graph represents the portion of the connector graph that communicates with the component C_i; We obtain the connector *CB-Graph* regarding the communication with a given component C_i by labelling with τ all the actions on a component different to C_i and by labelling with the action itself all the actions on C_i. $CBSimulation(AS_i, CB_i)$ successfully terminates if the expected behavior of the environment for the component C_i (AS_i) is *CB*-simulated [2] from the portion of the connector behavior regarding the communication with a given component (C_i). Referring to [2], the *CB-Simulation* informally is a notion of simulation based on observational equivalence [9].

5.3 Connectors synthesis for a multi-layered COM application

A multi-layered COM architecture is a system built from a set of COM components that have both the client and server side (components built through containment/delegation or aggregation mechanism). At a generic layer of the architecture the client side of a component is considered as a client of the layer itself and the server side is considered as a server of the precedent layer. In this context the connectors synthesis works by building, for each layer of the architecture, a connector between the clients of a layer and the servers of the same layer. For this reason we introduce two new data structures that allow to reduce a multi-layered system to a set of single-layered subsystems; for each one we build a connector. Informally we derives the *SAC-Graph (Server ACtual behavior Graph)* of a component C from his *AC-Graph* labelling with tau all the actions on a component different to C and with the action itself all the actions on C. In this way the *SAC-Graph* of C identifies the dynamic behavior of C on the server side. Analogously we derives the *CAC-Graph (Client ACtual behavior Graph)* of a component C from his *AC-Graph* labelling with tau all the actions on C and with the action itself all the actions on a component different to C. In this way the *CAC-Graph* of C identifies the dynamic behavior of C on the client side. Considering the definitions given in [2] we can obtain: *SAS-Graph (Server ASsumption Graph)*, *SEX-Graph (Server EXpected Graph)*, *CAS-Graph (Client ASsumption Graph)* and *CEX-Graph (Client EXpected Graph).*

6. THE DINING PHILOSOPHERS

This example is a COM instance of the problem known as *The dining philosophers* in which we consider two philosophers and a table with two forks. The experiment is based on a COM application composed of two clients and one server. The clients represent the two philosophers and the server is the table with two forks. We observed that running together these COM components without a coordination policy, the application comes to a deadlock as expected. Applying the technique presented in the following, we insert a COM connector in the system. Then the application is composed of two clients (the philosophers) and one server (the connector) that contains the old server and delegates to it the client requests consistently with the coordination policy encapsulated in the connector. We obtain that the connector based application is deadlock-free. We derive the coordination policy from the behavior specification of the components by applying the following technique.

We have a COM server whose MIDL file has been extended with the following *commented* code:

```
...
interface IHandler : IDispatch {
   [propget, id(1), helpstring("property Fork1")]
   HRESULT Fork1([out, retval] int *pVal);
   [propget, id(2), helpstring("property Fork2")]
   HRESULT Fork2([out, retval] int *pVal);
   [id(3), helpstring("method ReleaseForks")]
   HRESULT ReleaseForks();
};
...
library TABLELib {
...
   coclass Handler {
      [default] interface IHandler;
      //§
```

```
        //T = A + B;\\
        //A = !T1.E;\\
        //E = ?T1.A + ?T2.C;\\
        //C = !T2.F;\\
        //F = ?T2.C + ?T3.X;\\
        //X = !T3.N;\\
        //N = ?T1.T;\\
        //B = !T2.G;\\
        //G = ?T2.B + ?T1.D;\\
        //D = !T1.H;
        //H = ?T1.D + ?T3.J;
        //J = !T3.K;
        //K = ?T2.T;
        //§
    };
};
```

The extension made to MIDL code is represented by the *commented* code portion included between line //§ and line //§. The server which represents the table (T) expects two requests (A and B): the request for the first fork (method $Fork1$ with IDL *id* equal to 1) and the request for the second fork (method $Fork2$ with IDL *id* equal to 2). We recall that the notation $!SrvId$ means the request of the service with IDL *id* equal to Id to the server Srv. The requests A and B are defined, according to the CCS model discussed in Section 5.1, by defining the notifications (E, F, N, G, H e K) and the requests Req_i (A, B, C, D, X e J) within the notifications. For example the request A causes the reception in input of the data to invoke the method $Fork1$ with IDL *id* equal to 1; T as notification on A may expect in a non-deterministic way the request A again (the first fork is busy) or the request C that is, the request associated to the second fork. If C successfully terminates it means that both forks are busy so T can notify in output from PCC the request for the release of the forks (X). Analogously we work for the request B; B is the request for the second fork in a state in which both forks are free.

We associate to the client that represents the set of two philosophers a CCS process labelled with I; I is the parallel composition of two CCS processes (L and M). So $I = L \mid M$. The following are the two client processes specifications:

§
$L = ?T1.O;$
$O = !T1.L + !T2.P;$
$P = ?T2.Q;$
$Q = !T2.P + !T3.R;$
$R = ?T3.S;$
$S = !T1.I;$
§

and

§
$M = ?T2.U;$
$U = !T2.M + !T1.V;$
$V = ?T1.Z;$
$Z = !T1.V + !T3.Y;$
$Y = ?T3.W;$
$W = !T2.I;$
§

L requires the first fork sending, in output on PCC, to T the parameters to invoke the implementation of the method, with IDL *id* equal to 1 ($Fork1$), provided by T. Then L receives, in input on PCC, the notifications from T which may receive the same request again or the request for the second fork (P). Analogously happens for P and for the other requests (R, V e Y). The client application is built as a main process that enables the parallel execution of two threads. Parsing the precedent CCS statements we can obtain the *AC-Graphs* of T, L and M (Figure 2 and 3). Then we can derive the corresponding *AS-Graphs* by simply relabelling the input and output actions on the *AC-Graph* with the correspondent output and input actions. We consider the same system with a connector that contains the server T. T connects itself to the connector through the PCC $\{c\}$. The clients L and M connect themselves to the connector through the PCC $\{a\}$ and $\{b\}$. We use the PCC to model the communication channel by which the components connect themselves to the connector. The connector knows the component that requires his services by using the information about the PCC of this component. This channel is a synchronous procedure call channel.

From the *AS-Graph* we can build the *EX-Graph* of T, L and M (Figure 4 e 5). From the *EX-Graph* unification we obtain the actual behavior graph of the connector (Figure 6).

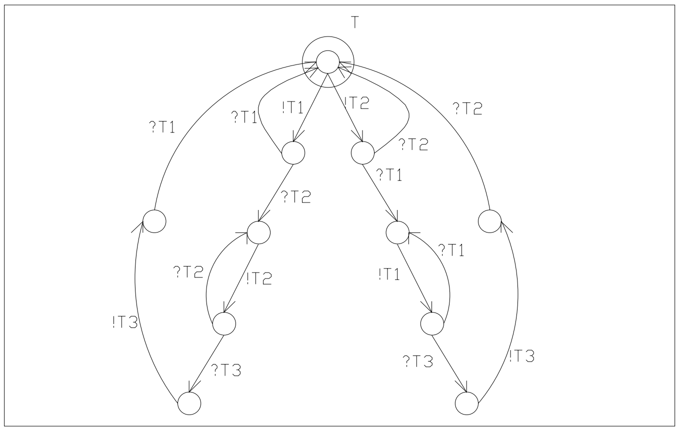

Figure 2: AC-Graph of T

The stop nodes within the transition graph of the connector immediately identify the deadlocks, so we delete the branches that terminate with these nodes. Then we run the algorithm of *CB-simulation* between the *AS-Graph* of the component connected to the connector through a given channel and the *CB-Graph* of the connector concerning the communication on this channel. We repeat this step for each channel that connects a given component to the connector. At this point the connector is deadlock-free if the

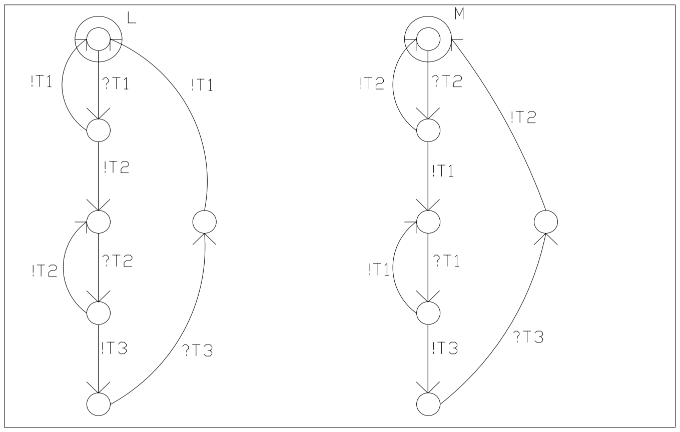

Figure 3: AC-Graph of L and M

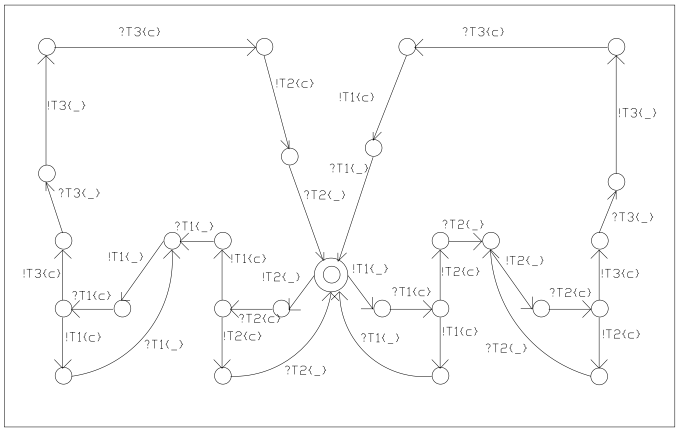

Figure 4: EX-Graph of T

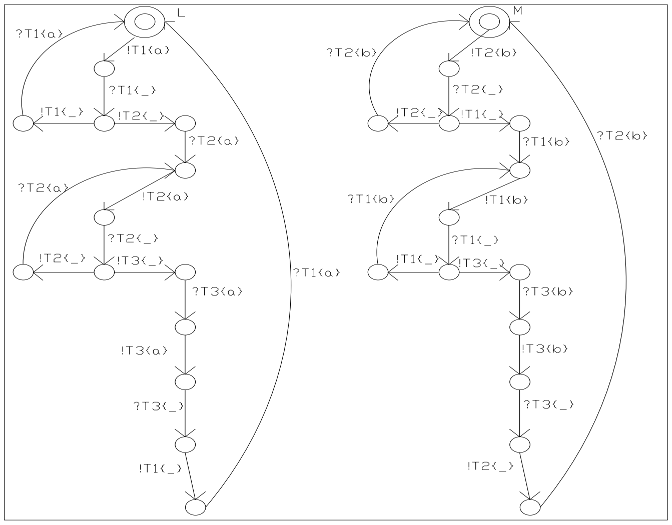

Figure 5: EX-Graph of L and M

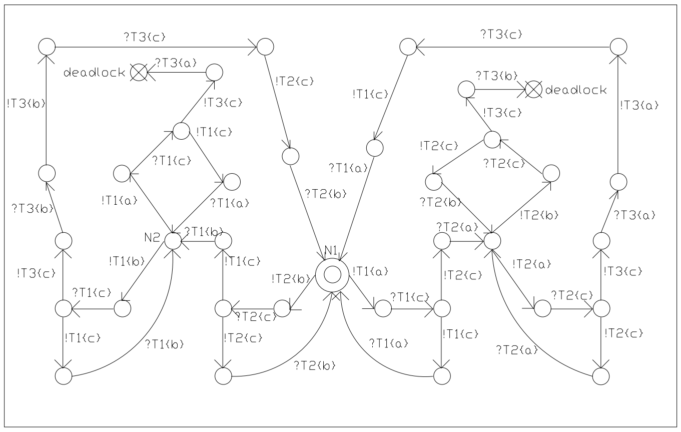

Figure 6: Transitions graph of the problem known as *The dining philosophers*

AS-Graphs of the components connected to the connector through a given set of channel, are *CB-simulated* by the portions of the connector behavior regarding the communication on this set of channels. Otherwise we can detect a deadlock in the nodes in which the *CB-simulation* relation does not hold.

7. FROM TRANSITION GRAPH TO ROUTING POLICY DEVELOPMENT

Let us suppose that we want to derive the code for the method provided by the connector which represents the request for the first fork (get_Fork1). The idea is to produce a code which represents the connector transition graph structure and avoids the deadlock paths. In the code structure states from which more arcs depart are represented by *if* conditions. Transitions are client requests to a certain server. This code consists in the possible delegation of the method get_Fork1 to the old server object, contained in the connector, depending on the state of the server object (the state of two forks) and by the channel the request comes from (the philosopher that does the request). These conditions map on some *if statements*. Each *if statement* has a given logic expression as guard. This expression is built by using the condition variables associated to the state of the two forks (*FREE* or *BUSY*) and the PCC identifiers that connect the clients L and M to the connector ($PCCA$ e $PCCB$). The following is the get_Fork1 code for the connector:

```
STDMETHODIMP CConn::get_Fork1(int *pVal, int pcc) {

  $if - statement_1$
...
  $if - statement_n$

  return E_HANDLE;
};
```

where $CConn$ is the COM class of which the server object representing the connector is instance; for the connector we extend the old get_Fork1 parameters list by adding a parameter that represents a PCC identifier. This parameter is specified by a client when he does this request; it identifies the channel with which a given client connects itself to the connector; the following is the $if - statement_i$ code:

```
if((fork1State == X_{var_i}) && (fork2State == Y_{var_i})) {
  if(pcc == Z_{var_i}) return E_HANDLE;

  fork1State = BUSY;

  return pHandler->get_Fork1(pVal);
}
```

where $X_{var_i}, Y_{var_i} \in \{FREE, BUSY\}$, $Z_{var_i} \in \{PCCA, PCCB, PCCUNKNOWN\}$, $i \in \{1, .., n\}$ and n is equal to the number of nodes within the connector transition graph the delegation of the request depends on (from these nodes it is possible to find a deadlock delegating the request without condition). We use the value $PCCUNKNOWN$ in the case the request can be done from any client. To find the values of X_{var_i}, Y_{var_i}, Z_{var_i} for each $i \in 1, .., n$ and of n we consider the nodes representing distinct states for the forks (two nodes are the same state if both represent the state in which the two forks are free/busy respectively) and such that at least an arc labelled by an input action towards T, on PCC a or b and with IDL id equal to the get_Fork1 IDL id, exit. These actions are in the set $\{!T1\{a\}, !T1\{b\}\}$ and are the unique actions that identify the requests for the get_Fork1 to the connector. Within the graph of Figure 6 there are two nodes ($N1$ and $N2$) from which at least an arc labelled by an action of the set $\{!T1\{a\}, !T1\{b\}\}$ exits. So $n = 2$. Each state of the connector corresponds to a tuple of the system components states. The node $N1$ corresponds to the system state in which both forks are free; since L requires first the first fork and M requires first the second fork, in the node $N1$, the component which requires the first fork is L that is connected to the connector by the PCC a. We can observe this by looking at the graph of Figure 6; the unique action, on PCC a, that exits from node $N1$ is $!T1\{a\}$. So we can derive the following assignments:

- $X_{var_1} = FREE$
- $Y_{var_1} = FREE$
- $Z_{var_1} = PCCB$

The node $N2$ corresponds to the system state in which the second fork is busy because M has required it; the first fork is free, L and M can both require it. So node $N2$ corresponds to the system state in which if the first philosopher requires the first fork and this request successfully terminates, then the system will deadlock (Figure 6):

- $X_{var_2} = FREE$
- $Y_{var_2} = BUSY$
- $Z_{var_2} = PCCA$

The following is the get_Fork1 implementation for the connector derived by the connectors synthesis automatic tool:

```
STDMETHODIMP CConn::get_Fork1(int *pVal, int pcc) {
   if((fork1State == FREE) && (fork2State == FREE)) {
      if(pcc == PCCB) return E_HANDLE;

      fork1State = BUSY;

      return pHandler->get_Fork1(pVal);
   }

   if((fork1State == FREE) && (fork2State == BUSY)) {
      if(pcc == PCCA) return E_HANDLE;

      fork1State = BUSY;

      return pHandler->get_Fork1(pVal);
   }

   return E_HANDLE;
};
```

analogously the tool works to derive the get_Fork2 implementation for the connector:

```
STDMETHODIMP CConn::get_Fork2(int *pVal, int pcc) {
   if((fork1State == FREE) && (fork2State == FREE)) {
      if(pcc == PCCA) return E_HANDLE;

      fork2State = BUSY;

      return pHandler->get_Fork2(pVal);
   }

   if((fork1State == BUSY) && (fork2State == FREE)) {
      if(pcc == PCCB) return E_HANDLE;

      fork2State = BUSY;

      return pHandler->get_Fork2(pVal);
   }

   return E_HANDLE;
};
```

for the $ReleaseForks$ we can see that the connector receives this request when it is in states in which both forks are busy. Within the graph of Figure 6 it is easy to observe that there are only two nodes from which at least an arc labelled by an action of set $\{!T3\{a\}, !T3\{b\}\}$ exits. However $n = 1$ because the precedent two nodes correspond to a state in which both forks are busy (these nodes are the same node). So $X_{var_1} = BUSY$ and $Y_{var_1} = BUSY$. Then this request is always deadlock-free and this result does not depend on the client issuing the request. Hence we can assign the value $PCCUNKNOWN$ to Z_{var_1} for the $ReleaseForks$. In this manner the connector delegates the $ReleaseForks$ to the old server for any client that issues the request. For this method the connector might simply delegate the request after the forks state updating. Our automatic tool derives a syntactically different but totally equivalent code:

```
STDMETHODIMP CConn::ReleaseForks(int pcc) {
   if((fork1State == BUSY) && (fork2State == BUSY)) {

      if(pcc == PCCUNKNOWN) return E_HANDLE;

      fork1State = FREE;
      fork2State = FREE;

      return pHandler->ReleaseForks();
   }

   return E_HANDLE;
};
```

In Section 5 we said that the connector interface is the union of the interfaces of the servers contained in the connector itself. In this example, the connector exhibits one interface which provides the same methods of the interface $IHandler$. This interface contains the methods get_Fork1, get_Fork2 and $ReleaseForks$ whose implementations have been derived, from the connector transition graph by our tool. This interface implements also a method get_PCC which is used by a client to get a PCC to connect itself to the connector. This PCC is the PCC identifier that appears in the parameters list of any method provided by the connector. The following is the $IConn$ interface specification code:

```
interface IConn : IDispatch {
      [propget, id(1), helpstring("property Fork1")]
         HRESULT Fork1([out] int *pVal, [in] int pcc);

      [propget, id(2), helpstring("property Fork2")]
         HRESULT Fork2([out] int *pVal, [in] int pcc);

      [id(3), helpstring("method ReleaseForks")]
         HRESULT ReleaseForks([in] int pcc);

      [propget, id(4), helpstring("property PCC")]
         HRESULT get_PCC([out, retval] int *pVal);
};
```

The get_PCC implementation is easy to derive because it depends only from the number m of the PCC that connects the clients to the connector:

```
STDMETHODIMP CConn::get_PCC(int *pVal) {

   $if - statement_1$
   ...
   $if - statement_m$

   return E_HANDLE;
}
```

where $if - statement_j$ is as follows:

```
if(($pcc_j State$ == NOTCONNECTED) {
   $pcc_j State$ = CONNECTED;
   *pVal = $PCC_j$;

   return S_OK;
}
```

where $pcc_j \in \{pcca, pccb, ..\}$ and $PCC_j \in \{PCCA, PCCB, ..\}$. The following is the get_PCC complete implementation:

```
STDMETHODIMP CConn::get_PCC(int *pVal) {
   if((pccaState == NOTCONNECTED) {
      pccaState = CONNECTED;
      *pVal = PCCA;

      return S_OK;
   }

   if((pccbState == NOTCONNECTED) {
      pccbState = CONNECTED;
      *pVal = PCCB;

      return S_OK;
   }

   return E_HANDLE;
}
```

we must modify every client code according to the following steps:

1. insert within local variables declaration a PCC identifier (*int pcc*);
2. replace the string *Handler* with the string *Conn* for each client code;
3. at the beginning of a client code introduce a connection phase ($pConn \rightarrow\ get\_PCC(\&pcc)$);
4. add to the parameters list a PCC identifier (*pcc*) for each call to the methods of the connector.

Our solution has the drawback of slightly modify the client code. This can be easily automated by simply observing the clients code. When we have the clients in a binary format we must find a technique to avoid the clients code updating. A possible way to achieve this is by updating the registry keys values of the old server with the registry keys values of the connector and by handling the PCC numbering directly into the synthesized connector for example by using an instance counter. In this manner we could avoid step 1, 2, 3 and 4.

8. CONCLUSIONS AND FUTURE WORK

In the paper we presented a technique to transform a COM/DCOM application which deadlocks in a COM/DCOM application which has the same clients-server structure augmented with a new server that *contains* the old one and filters all the clients requests to the old one, by using a deadlock-free policy. In our present implementation clients have to be slightly modified as suggested above. The technique relies on an enhanced component interface that specifies its dynamic interaction behavior. This information should be specified by the component developer and only reflects the component dynamic behavior at the architectural level. The new server acts as a connector and its specification and implementation is automatically synthesized. The technique does not always succeed, since it depends on the deadlock nature. Informally we classified the deadlocks that can be solved as *coordination deadlocks* as opposed to deadlocks that occur because of some component internal behavior and cannot be solved by externally coordinating components interactions. In a component-based setting in which we are assuming black-boxes components, this is the best we can expect to do.

At present we have developed an implementation of the automatic tool for COM connectors synthesis (called *COM Connectors Generator*). This tool works well for single-layered COM systems in which the server components are single-threaded servers. The tool derives the connector transition graph, from the dynamic behavior specification of system components, and then proceeds with the detection and the recovery for deadlocks caused by a wrong coordination policy while it only detects the deadlocks caused by internal behavior. This implementation includes also the connector code generation phase. A possible extension is to let it work also for multi-layered COM systems in which, for example, some servers are multi-threaded servers.

Besides extending the tool applicability, future work concern the application of the technique to other case studies and to real case COM/DCOM applications. Future work also concern the implementation of the techniques sketched at the end of Section 7 to avoid all the system clients components modifications.

Acknowledgements

This work has been partially supported by Progetto MURST SALADIN. We thank the referees of the paper for their careful reviews and relevant suggestions.

9. REFERENCES

[1] *Microsoft Developer Network Library*, July 2000.

[2] P. Inverardi and S. Scriboni. Connectors syntesis for deadlock-free component based architectures. *Technical Report, Università dell'Aquila. Submitted for pubblication*, February 2001.

[3] P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. *Proceed. FASE 2001, LNCS 2029 pp. 60-75*, April 2001.

[4] P. Inverardi, D. Yankelevich, and A. Wolf. Static checking of system behaviors using derived component assumptions. *ACM TOSEM Volume 9, Number 3*, July 2000.

[5] D. Jackson and K. J. Sullivan. Com revisited: Tool-assisted modelling of an architectural framework. *Foundations of Software Engineering, San Diego, CA*, 2000.

[6] N. Kaveh and W. Emmerich. Deadlock detection in distributed object system. *8th FSE/ESEC, Vienna*, September 2001.

[7] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *In Proceedings of the 1997 Symposium on Software Reusability and Proceedings of the 1997 International Conference on Software Engineering*, May 1997.

[8] N. R. Metha, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *In Proceedings of the 2000 International Conference on Software Engineering*, 2000.

[9] R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

[10] D. S. Platt. *Understanding COM+*. Microsoft Press, 1999.

[11] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging Discipline.* 1996.

[12] K. J. Sullivan, M. Marchucov, and J. Socha. Analysis of a conflict between aggregation and interface negotiation in microsoft's component object model. *IEEECS Log Number 106183*, 1999.

[13] K. J. Sullivan, J. Socha, and M. Marchucov. Using formal methods to reason about architectural standards. In *In Proceedings of the 1997 International Conference on Software Engineering*, 1997.

[14] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Inc., 1992.

[15] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.