ELSEVIER

The Journal of Systems and Software 65 (2003) 173–183

The Journal of Systems and Software

www.elsevier.com/locate/jss

Deadlock-free software architectures for COM/DCOM Applications ☆

Paola Inverardi, Massimo Tivoli *

Dip. Informatica, University of L'Aquila, via Vetoio 1, Coppito, I-67100 L'Aquila, Italy

Received 8 December 2001; received in revised form 1 March 2002; accepted 25 March 2002

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. In this paper we present a technique to allow connectors synthesis for deadlock-free component based architectures [IEEE Proceedings of the 16th ASE, 2001] in the context of COM/DCOM applications. This work also provides guidelines to implement an automatic tool that derives the implementation of routing deadlock-free policies within the connector from the dynamic behavior specification of the COM components. Deadlock is then prevented by inserting the synthesized connector within the system via COM composition mechanisms 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''*.

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 (Szyperski, 1998; Boehm and Abts, 1999; Inverardi and Uchitel, 2001; Kaveh and Emmerich, 2001). In this paper we present a technique to allow connectors synthesis for deadlock-free component based architectures in the COM/DCOM context. Recently, software architecture (SA) has emerged as an area that places significant importance on components interaction. In the SA domain, components interaction is embodied in the notion of software connectors. Software connectors mediate interactions among components (Shaw and Garlan, 1996). In a COM/DCOM architecture, we should consider the following mechanisms as implicit 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 explicit components. Special purpose architecture description languages (Allen and Garlan, 1997; Kramer and Magee, 1997) (ADL) have been devised to provide support for modelling more sophisticated and powerful connectors. A predominant focus of existing ADLs (Allen and Garlan, 1997; Kramer and Magee, 1997) has been on verifying the properties of the modelled behavior. The essence of these approaches is to build a model of the system behavior and then analyze it. This allows for detecting dynamic problems like deadlock. No support is then provided to automatically fix the problem. 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 COM composition mechanisms. Since we aim at proving properties about the dynamics of the composed system,

☆ This is a revised and extended version of a paper presented at FSE/ESEC 2001.

* Corresponding author. Tel.: +39-0862433734.

E-mail addresses: inverard@univaq.it (P. Inverardi), tivoli@univaq.it (M. Tivoli).

0164-1212/03/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
doi:10.1016/S0164-1212(02)00037-7

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. The extension that we apply to the MIDL code of the component specification represents an innovative aspect of this work. We give the information about the component dynamic behavior as commented code. To do that, we use a language that allows to translate the commented code into the automata representing the component dynamic behavior. Our approach is based on the theoretical framework introduced in Inverardi and Scriboni (2001). The idea there is to build applications by assuming a defined architectural style, namely a modified version of the C2 architectural style (Medvidovic et al., 1997). 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-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 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. In this work we are interested only in one specific behavioral property that a COM/DCOM system must hold to work correctly. This property is deadlock freeness. In Section 8 we mention a possible technique to deal with other behavioral properties. In this way we could assign suitable types of routing policies to the connector. 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 (Inverardi and Scriboni, 2001) for a COM/DCOM single-layered application. Section 6 illustrates an application of connectors synthesis for a COM single-layered system; this system implements an instance of the problem known as *The dining philosophers* (Tanenbaum, 1992) with two philosophers and two forks. Section 7 deal with the COM/DCOM implementation of the technique. In Section 8 we present conclusions and future works, in order to address a more general COM/DCOM context and to leave all the system COM components unmodified. It's worth noticing that although at present DCOM does not exist anymore, 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 Inverardi and Scriboni (2001) 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 component based architectures, we can roughly identify two types of deadlock problems: (i) deadlocks caused by a wrong coordination policy; (ii) deadlocks caused by a component internal behavior. The first class of problems can be treated in the component setting by operating on the architectural context, namely on the connector, the second type of deadlocks, instead, cannot be solved unless we directly modify the component internal code. 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. This is why we focus on the first class of problems. We develop an automatic technique to connectors synthesis as discussed in Inverardi and Scriboni (2001) in a COM/DCOM context. In order to do this we proceed as follows: first we understand how the COM model can reflect the reference architectural style of Inverardi and Scriboni (2001); 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 the component dynamic behavior (Inverardi and Uchitel, 2001; Inverardi et al., 2000). From these automata we can obtain the connector transition graph. Thus we need a way to accommodate 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 (Milner, 1989) statement. In Section 5.1 we provide a general CCS model for a server side request and a client side request. At this point it is possible to use the connectors synthesis discussed in Inverardi and Scriboni (2001) to obtain the deadlock-free COM connector transition graph. This graph represents a model of an deadlock-free routing policy of the requests. The last problem is to understand how we can obtain the implementation of this connector by simply observing the structure of the connector transition graph; this is treated in Section 7. Summarizing, our synthesis technique centralizes the handling of the requests in a unique big component, i.e. the connector. This can represent a scalability weakness. The centralization of the requests routing policy is inevitable from a logic point of view.

Adding to the system a component that acts as a components interaction broker inevitable means that it must have a global vision of all the system. Thus the connector (the communication broker) encapsulates the whole requests routing policy. On the other hand, from the point of view of the connector implementation we could think of producing a distributed implementation of the transition graph connector. In this way the components interaction broker is logically unique but it is physically decomposed in several modules. We focused on deadlock because deadlock detection is a prerequisite for architectural property prediction. If a system deadlocks, any predictions concerning its reliability or timing are invalid. Hence, any prediction mechanism must include also deadlock detection.

3. Background

A SA specification describes the high-level design of a system in terms of the structural and communication relationships of its components. At this level of description, the specification focuses on the interaction behaviors exhibited by the components, not the algorithms used internally by components to carry out their functional roles. In recent years the study of SA has emerged as an autonomous discipline requiring its own concepts, formalisms, methods and tools and as a powerful means for handling the design and analysis of complex distributed systems. The peculiarity of SA abstractions is to model a system in terms of components and connectors, where components represent a suitable abstraction of computational subsystems and connectors formalize the interactions among components. Thus in the discipline of SA we can distinguish two types of building blocks: (a) components and (b) connectors. A component is a unit of computation or data store. A connector is an architectural building block used to model interactions among components and rules that govern those interactions. SAs support the formal modeling of a system allowing for both a topological (static) description and a behavioral (dynamic) one. SA represents the most promising approach to tackle the problem of scaling up in software engineering, because, through suitable abstractions, it provides the way to make large applications manageable.

3.1. The reference architectural style

According to Medvidovic et al. (1997) and Taylor et al., 1996], we define an architectural style as a set of constraints on a SA that identify a class of architectures with similar features. An architectural software style is determined by the following: (i) a set of component types that perform some function at runtime; (ii) a topological layout of these components indicating their runtime interrelationships; (iii) a set of semantic constraints; (iv) a set of connectors that mediate communication, coordination or cooperation among components. In this paper we introduce an architectural style called *Connector Based Architecture* (CBA). CBA is derived from the architectural style C2 (Medvidovic et al., 1997; Taylor et al., 1996). In this style the components and the connectors are modeled as explicit entities of an architecture. This style reflects the rules defined by an implementation independent architecture description language. The main characteristics of C2 style are: (i) component based, (ii) scalability, (iii) two types of elements: components and connectors, (iv) both components and connectors have defined *top* and *bottom*, (v) the *top* is connected to the *bottom* of a single connector, (vi) the *bottom* is connected to the *top* of a single connector, (vii) there is no bound on the number of components or connectors that may be attached to a single connector, (viii) two types of messages: *request* from *bottom* to *top* and *notification* from *top* to *bottom*, (ix) the *top* domain specifies the set of notifications a component responds to, and the set of requests it emits upward in the architecture, (x) 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 CBA style constrains C2 by imposing the following characteristics: (i) synchronous message passing, (ii) connectors cannot directly communicate, (iii) connectors are only routing devices, without any filtering policies, (iv) connectors have a strictly sequential input–output behavior. In the following Section we present the COM/DCOM features that we 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. To allow clients to access its functions, a COM/DCOM server implements one or more interfaces. An interface is a binary structure in the address space of a component, whose layout is defined as a table of pointers to functions. The specification of all functions that can be called through an interface is defined by the type of that interface. 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. 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 defines two approaches to use existing components as building blocks of new ones: (i) *containment/delegation*: an outer component encapsulates one or more inner components and uses their services, (ii) *aggregation*: the outer component can directly expose to its clients its pointer to the inner interface.

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.

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*] (input on PCC) and [*out*] (output on PCC). The services provided by a COM server also have a return value. The return value is considered, from the server, as an output element on PCC. The return value type assumes a unique success value (**S_OK**) and a given set of error values (**E_**...).

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 (through containment/delegation mechanism) server that provides a coordination service and that it is a tangible entity at system implementation level. 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, mapping top and bottom with client side and server side respectively, 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.

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 (Milner, 1989) 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 Inverardi and Scriboni (2001) and Inverardi and Uchitel (2001).

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. The actual output action on PCC for the server is modelled as the result of a non-deterministic composition of output actions. This because, after that a server has received and performed a request, the server object could expect, in an unpredictable way, one request 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: $[A|\ldots|Z][0|\ldots|9]+$. $[A|\ldots|Z]$ is the letter for the substring *Srv* and $[0|\ldots|9]+$ 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.

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

where $Req, Req_1, \ldots, Req_n, Not \in \{X, Y, \ldots\}$ are process variables; *Srv* ranges over upper case letters (process variable associated to server); $Id, Id_1, \ldots, Id_n$ 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; $?SrvId_1, \ldots, ?SrvId_n$ 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:

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

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. Referring to Fig. 1, the technique defined in Inverardi and Scriboni (2001) 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 Inverardi and Scriboni (2001), 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 Inverardi and Scriboni (2001), 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 freeness. 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 freeness 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

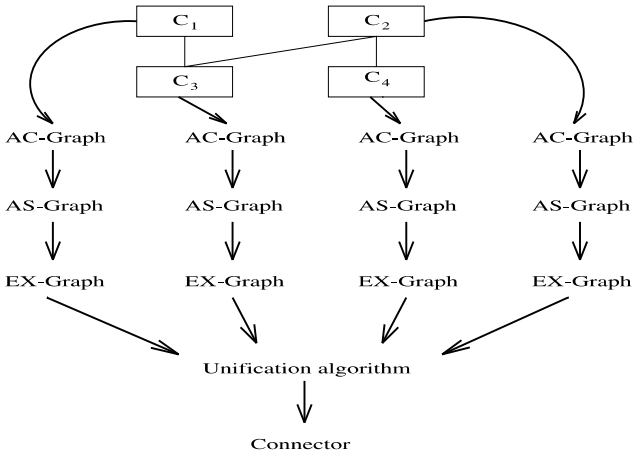

Fig. 1. Connector synthesis.

expects it. Given the CBA style, the component environment can only be represented by one or more connectors, thus in Inverardi and Scriboni (2001) they refines the definition of AS-Graph into 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 inputted 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 (Inverardi and Scriboni, 2001). 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 (Inverardi and Scriboni, 2001) for each component. This step is performed by using a parser which translates the CCS code in a graph. After we have built the AC-Graphs (Inverardi and Scriboni, 2001) for each component we 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 (Inverardi and Scriboni, 2001) of the component C_i which is connected to the connector; CB_i is the *CB-Graph* (Inverardi and Scriboni, 2001) for C_i. Referring to Inverardi and Scriboni (2001) 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 (Inverardi and Scriboni, 2001) from the portion of the connector behavior regarding the communication with a given component (C_i). Referring to Inverardi and Scriboni (2001), the *CB-Simulation* informally is a notion of simulation based on observational equivalence (Milner, 1989).

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 since there are no timeouts defined on the clients. 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 Release-
  Forks")]
  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 *Fork*1 with IDL *id* equal to 1) and the request for the second fork (method *Fork*2 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 *Fork*1 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 $I = L|M$. The following are the two client processes specifications:

$L = ?T1.O;$	and	$M = ?T2.U;$
$O = !T1.L + !T2.P;$		$U = !T2.M + !T1.V;$
$P = ?T2.Q;$		$V = ?T1.Z;$
$Q = !T2.P + !T3.R;$		$Z = !T1.V + !T3.Y;$
$R = ?T3.S;$		$Y = ?T3.W;$
$S = !T1.L;$		$W = !T2.M;$

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 (*Fork*1), 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. To each component corresponds a labelled transition system (LTS). This LTS is handled as a graph.

Definition 2 (*Graph*). Let $(S, L, \rightarrow, s)$ the LTS for a component C. The graph G for C is a tuple of the form: $(N_G, LN_G, A_G, LA_G, s)$ where $N_G = S$ is a set of nodes; LN_G is a set of state labels; LA_G is a set of arc labels plus the label $\{\tau\}$; $A_G \subseteq N_G \times LA_G \times N_G$ is a set of arcs; s is the root node.

We want a structure to describe the dynamic behavior of a component (the CCS statement in the MIDL code):

Definition 3 (*AC-Graph* (*Actual Behaviour Graph*)). Let $(N_G, LN_G, A_G, LA_G, s)$ the graph for a component C, a graph of the form $(N_G, LN_G, A_G, LA_G, s)$ is the AC-Graph. Typically a node of the AC-Graph is indicated by v.

This graph is the graph of the actual behavior of a component C. Every node represents one state of the component labelled by the CCS term which describes the CCS process associated to that state. The root node is the initial state. Every arc represents a possible transition to a new state. The label on the arc is the action performed by the component.

Parsing the precedent CCS statements we obtain the AC-Graphs of T, L and M. 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. The AS-Graph represents the (deadlock-free) expected behavior of the environment derived from the actual behavior of the component. A component is not in deadlock if the environment always provides the actions requested to change its state:

Definition 4 (*AS-Graph* (*ASsumption Graph*)). Let $(N_{AC}, LN_{AC}, A_{AC}, LA_{AC}, s)$ l'AC-Graph for the component C then the corresponding AS-Graph is: $(N_{AC}, LN_{AC}, A_{AS}, LA_{AC}, s)$ where $A_{AS} = \{(v, \bar{a}, v') | (v, a, v') \in A_{AC}\}$. Typically a node in an AS-Graph is indicated by μ.

The AS-Graph is obtained simply by applying the complement to the actions of the AC-Graph. If a component performs an input action, the component, to avoid the deadlock, expects that his environment performs in output the same action. Analogously for the output actions performed by the component. In this manner the component, for a given action, is able to synchronize with the environment.

Let us now 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.

If we think of the connector and of the components connected to it, in order to avoid the deadlocks, the connector must provide to the connected components all the actions that they expect to change their current state. It means that the connector must simulate the *AS-Graphs* of the connected components. The connector performs only input and output actions. Thus if the connector receives a message in input by a component, the connector must send this message in output to the receiver. Analogously if the connector outputs a message to a component, it has received in input this message from the sender:

Definition 5 (*EX-Graph* (*EXpected Graph*)). Let $(N_{\mathrm{AS}}, LN_{\mathrm{AS}}, A_{\mathrm{AS}}, LA_{\mathrm{AS}}, s)$ the AS-Graph for the component C_i we define the graph of the expected behavior of the connector by C_i, the graph $(N_{\mathrm{EX}}, LN_{\mathrm{EX}}, A_{\mathrm{EX}}, LA_{\mathrm{EX}}, s)$ where $N_{\mathrm{EX}} = N_{\mathrm{AS}}$ and $LN_{\mathrm{EX}} = LN_{\mathrm{AS}}$; A_{EX} and LA_{EX} are empty; $\forall(\mu, \alpha, \mu') \in A_{\mathrm{AS}}$ with $\alpha \neq \tau$:

- *create a new node* μ_{new} *with a new label* $LN_{\mathrm{G}}(\mu_{\mathrm{new}})$,
- *add the node to* N_{EX} *and the label to* LN_{EX};
- *if* (μ, α, μ') *is such as* $\alpha = a$ *for some* a:
  - *add the label* a_i *and* $\overline{a_?}$ *to* LA_{EX};
  - *add* $(\mu_i, a_i, \mu_{\mathrm{new}})$ *and* $(\mu_{\mathrm{new}}, \overline{a_?}, \mu')$ *to* A_{EX};
- *if* (μ, α, μ') *is such that* $\alpha = \overline{a}$ *for some* a:
  - *add the label* $\overline{a_i}$ *and* $a_?$ *to* LA_{EX};
  - *add the label* $(\mu_i, a_?, \mu_{\mathrm{new}})$ *and* $(\mu_{\mathrm{new}}, \overline{a_i}, \mu')$ to A_{EX};

  $\forall(\mu, \tau, \mu') \in A_{\mathrm{AS}}$ *add* τ *to* LA_{EX} *and* (μ, τ, μ') *to* A_{EX}.

Each EX-Graph describes the expected behavior of the connector for one of the connected components. It means that an EX-Graph is a partial view of the expected behavior of the connector. This partial view represents the point of view of the component for which we build the EX-Graph. Thus we will find some actions on the channel which connects the component to the connector and some actions on an unknown channel for the given component.

Then from the EX-Graph unification we obtain the complete actual behavior graph of the connector (Fig. 2). 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.

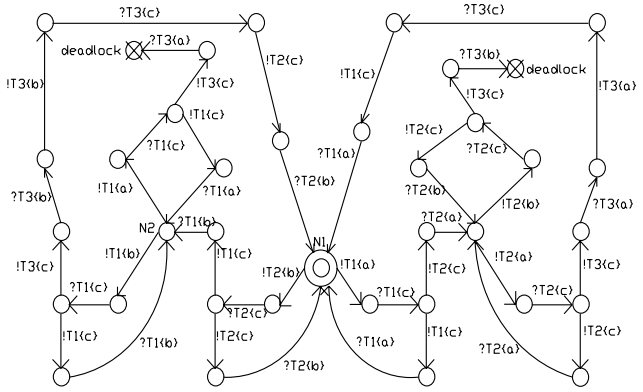

Fig. 2. Transitions graph of the problem known as *The dining philosophers*.

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 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_Fork*1). 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_Fork*1 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_Fork*1 code for the connector:

```
STDMETHODIMP CConn::get_Fork1(int *pVal,
int pcc) {
  if − statement1
...
  if − statementn
  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_Fork*1 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 $X_{\mathrm{var}_i}, Y_{\mathrm{var}_i} \in \{FREE, BUSY\}$, $Z_{\mathrm{var}_i} \in \{PCCA, PCCB, PCCUNKNOWN\}$, $i \in \{1, \ldots, n\}$ 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 $i \in 1, \ldots, n$ 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_Fork*1 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_Fork*1 to the connector. Within the graph of Fig. 2 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 Fig. 2; the unique action, on PCC a, that exits from node $N1$ is $!T1\{a\}$. So we can derive the following assignments: (i) $X_{var_1} = FREE$, (ii) $Y_{var_1} = FREE$, (iii) $Z_{var_1} = PCCB$. 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 (Fig. 2): (i) $X_{var_2} = FREE$, (ii) $Y_{var_2} = BUSY$, (iii) $Z_{var_2} = PCCA$. The following is the *get_Fork*1 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_Fork*2 implementation for the connector. 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 Fig. 2 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. 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_Fork*1, *get_Fork*2 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 *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. 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: (i) insert within local variables declaration a PCC identifier (*int pcc*), (ii) replace the string *Handler* with the string *Conn* for each client code, (iii) at the beginning of a client code introduce a connection phase ($pConn \rightarrow get\_PCC(\&pcc)$), (iv) 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 clients in binary format we must find a technique to avoid the clients code updating. A possible solution to avoid changes in the client code is to update the Windows registry automatically to point to a factory that returns COM interface pointers to the synthesized server. This object encapsulates the interaction between some client and the connector server. It exports the original interface and the generated implementation manages the PCC numbering so that the appropriate parameter can be passed to the *IConn* server implementation. The factory automatically implements the PCC numbering. In this way we can avoid step (i), (ii), (iii) and (iv) above.

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-servers structure augmented with a new server that *contains* the old ones and filters all the clients requests by using a deadlock-free policy. In our present implementation clients have to be slightly modified. 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. 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 concerns 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. Other relevant future research directions are considered below: (i) How to assign any routing policy to the connector, (ii) Synthesizing dynamic information.

Acknowledgements

We thank the reviewers for their careful reading of the paper. This work has been partially supported by Progetto MURST SALADIN and MIUR SAHARA.

References

Allen, R., Garlan, D., 1997. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6 (3), 213–249.

Boehm, B., Abts, C., 1999. Cots integration: plug and pray? IEEE Computer 32 (1).

Inverardi, P., Scriboni, S., 2001. Connectors synthesis for deadlock-free component based architectures. IEEE Proceedings of the 16th ASE.

Inverardi, P., Uchitel, S., 2001. Proving deadlock freedom in component-based programming. Proceedings of the FASE 2001, LNCS 2029 pp. 60–75.

Inverardi, P., Yankelevich, D., Wolf, A., 2000. Static checking of system behaviors using derived component assumptions. ACM TOSEM 9 (3).

Kaveh, N., Emmerich, W., 2001. Deadlock detection in distributed object system. 8th FSE/ESEC, Vienna.

Kramer, J., Magee, J., 1997. Exposing the skeleton in the coordination closet, Coordination languages and Models, 2nd Int. Conf. Coordination '97, Berlin, 1997.

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

Milner, R., 1989. Communication and Concurrency. Prentice Hall, New York.

Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an emerging Discipline. Prentice Hall, Englewood Cliffs, NJ.

Szyperski, C., 1998. Component Software. Beyond Object Oriented Programming. Addison-Wesley, Harlow England.

Tanenbaum, A.S., 1992. Modern Operating Systems. Prentice Hall Inc, New Jersey.

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

Paola Inverardi is full professor at University of L'Aquila. Previously she has worked at IEI-CNR in Pisa and at Olivetti in Pisa. Paola Inverardi's main research area is in the application of formal methods to software development. In recent years her research interests mainly concentrated in the field of SAs and in techniques, either rewriting-base and model checking-based, for the verification and analysis of complex systems.

Massimo Tivoli is a PhD student in computer science at University of L'Aquila. His research interests mainly concentrated in the field of SA and of COTS components assembly.