Synthesis of Concurrent and Distributed Adaptors for Component-Based Systems

Marco Autili, Michele Flammini, Paola Inverardi,
Alfredo Navarra, and Massimo Tivoli

Computer Science Department, University of L'Aquila
Via Vetoio I-67100 L'Aquila, Italy
{marco.autili,flammini,inverard,navarra,tivoli}@di.univaq.it

Abstract. Building a distributed system from third-party components introduces a set of problems, mainly related to compatibility and communication. Our existing approach to solve such problems is to build a *centralized adaptor* which restricts the system's behavior to exhibit only *deadlock-free* and *desired interactions*. However, in a distributed environment such an approach is not always suitable. In this paper we show how to automatically generate a *distributed adaptor* for a set of black-box components. First, by taking into account a specification of the interaction behavior of each component, we synthesize a behavioral model of a centralized *glue adaptor*. Second, from the synthesized adaptor model and a specification of the desired behavior, we generate a set of adaptors local to the components. They cooperatively behave as the centralized adaptor restricted with respect to the specified desired interactions.

1 Introduction

Nowadays, a growing number of software systems are built as composition of reusable or *Commercial-Off-The-Shelf* (COTS) components. *Component Based Software Engineering* (CBSE) is a reuse-based approach which addresses the development of such systems. One of the main goals of CBSE is to compose and adapt third-party components to make up a system [1]. Building a distributed system from reusable or COTS components introduces a set of problems. Often, components may have incompatible or undesired interactions. A widely used technique to deal with these problems is to use adaptors and interpose them between the components forming the system that is being assembled.

One existing approach (implemented in the *SYNTHESIS* tool [2]) is to build a *centralized adaptor* which restricts the system's behavior to exhibit only a set of *deadlock-free* or *desired interactions*. However in a distributed environment it is not always possible or convenient to insert a centralized adaptor. For example, existing legacy distributed systems might not allow the addition of a new component (i.e., the adaptor) which coordinates the information flow in a centralized way. Moreover, the coordination of an increasing number of components can cause loss of information andbottlenecks, with corresponding

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 17–32, 2006.

increase of the response time of the centralized adaptor. In contrast, building a distributed adaptor might increase the applicability of the approach in real-scale contexts.

In this paper we describe an approach for automatically generating a *distributed adaptor* for a set of black-box components. Given (i) a specification of the *interaction behavior* of each component with its environment and (ii) a specification of the *desired behavior* that the system to be composed must exhibit, it generates *component local adaptors* (one for each component). These local adaptors suitably communicate in order to avoid possible deadlocks and enforce the specified desired interactions. They constitute the distributed adaptor for the given set of black-box components.

Starting from the specification of the components' interaction behavior, our approach synthesizes a behavioral model (i.e., a Labeled Transition System (LTS)) of a centralized *glue adaptor*. This is done by performing a part of the synthesis algorithm described in [2] (and references therein). At this stage, the adaptor is built only for modeling all the possible component interactions. It acts as a simple router and each request/notification it receives is strictly delegated to the right component. By taking into account the specification of the desired behavior that the composed system must exhibit, our approach explores the centralized glue adaptor model in order to find those states leading to deadlocks or to interactions different from the desired ones. This process is used to automatically derive the set of local adaptors that constitute the *correct*[1] and *distributed* version of the centralized adaptor. It is worth mentioning that the construction of the centralized glue adaptor is required to deal with deadlock in a fully-automatic way. Otherwise we should make the stronger assumption that the specification of the desired behaviors itself ensures also deadlock-freeness. The approach presented in this paper has various advantages with respect to the one described in [2] concerning the synthesis of centralized adaptors. The most relevant ones are: (a) no centralized point of information flow exists; (b) the degree of parallelism of the system without the adaptor is now maintained. Conversely, the approach in [2] does not permit parallelism due to the adaptor centralization; (c) all the domain-specific deployment constraints imposed on the adaptor can be removed. In [2] we applied the synthesis of centralized adaptors to COM/DCOM applications. In this domain, the centralized adaptor and the server components had to be deployed on the same machine. On the contrary, the approach described in this paper allows one to deploy each component (together with its local adaptor) on different machines.

The remainder of the paper is structured as follows: Section 2 describes the application domain. In Section 3 the synthesis of decentralized adaptors is firstly described and then formalized by also proving its correctness. Section 4 describes our approach at work by means of a running example. Section 5 discusses related work, and finally, Section 6 concludes and discusses future work.

[1] With respect to deadlock-freeness and the specified desired behavior.

2 The Context

In our context, a distributed system is a network of interacting black-box components $\{C_1, \ldots, C_n\}$ that can be simultaneously executed. Components communicate each other by message passing according to synchronous communication protocols. This is not a limitation because it is well known that with the introduction of a buffer component we can simulate an asynchronous system by a synchronous one [3]. We distinguish between *standard communication* and *additional communication*. The first denotes the messages that components can exchange. The latter denotes the messages that the local adaptors exchange in order to coordinate each other. Due to synchronous communication, a deadlocking interaction might occur whenever components contend the same request. Furthermore, by letting components interact in an uncontrolled way, they might perform undesired interactions. To overcome this problem we promote the use of additional components (called local adaptors). Each local adaptor is a wrapper that performs the component' standard communication and mediates it by exchanging synchronizing information (i.e., additional communication), when needed. Synchronizing information allow components to harmonize their interaction on requests and notifications. Each component is directly connected to its local adaptor through a synchronous channel; each local adaptor is connected to the other ones, through asynchronous channels, in a peer-to-peer fashion (see for instance the right-hand side of Figure 1). For the sake of clarity, we assume the components are single-threaded and hence all the requests and notifications can be totally ordered to constitute a set of sequences (i.e., a set of traces). Note that this is not a restriction since a multi-threaded component can always be modeled as a set of single-threaded (sub)components simultaneously executed. Interaction among components is modeled as a set of *linearizations* obtained by means of interleaving [4]. It is worth noting that, in such a concurrent and distributed context, we cannot assume either a single physical clock or a set of perfectly synchronized ones in order to determine whether an event a occurs before an event b or vice versa. We then need to define a relationship among the system events by abstracting both on the absolute speed of each processor and on the absolute time. In this way we ignore any absolute time scale and we use the well known *happened-before relation* and *time-stamps method* (see [5] for a detailed discussion).

3 Method Description and Formalization

In this section we first describe our method to deal with the adaptation problem in a component-based setting. Then, we gradually formalize it by means of a detailed discussion and pseudo-code description of the setup and local adaptors interaction procedures. This section also proves the correctness of our approach and concludes with a brief discussion about the additional communication overhead.

3.1 Method Description

Our method (see Figure 1) assumes as input: (i) a behavioral specification of the system formed by interacting components. It is given as a set $\{AC_1, \ldots, AC_n\}$ of LTS (one for each component C_i). The behavior of the system is modeled by composing in parallel all the LTS and by forcing synchronization on common actions; (ii) the specification of the desired behavior that the system must exhibit. It is given in terms of a LTS, from now on denoted by P_{LTS}.

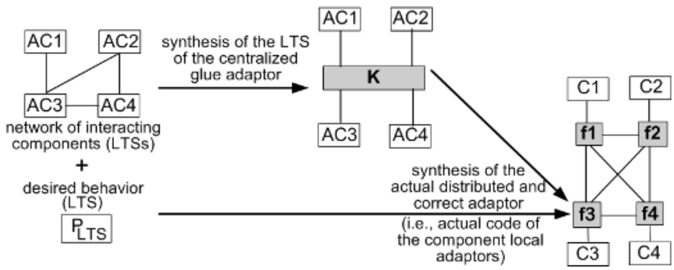

Fig. 1. 2-step method

These two inputs are then processed in two main steps. (1) By taking into account all component LTSs, we automatically derive the LTS K that models the behavior of a centralized glue adaptor. K, at this stage, models all the possible component interactions and it does not apply any adaptation policy. In other words, K performs standard communication simply routing components requests and notifications. In this way, it represents all possible linearizations by using an interleaving semantics. K is derived by performing the *graph unification* algorithm described in [2]. It is worth mentioning that each state of K (i.e., a global state) is a tuple $< S_1, \ldots, S_n >$ where each S_i is a state of AC_i (see for instance Figure 2). Hereafter, when the current state of a component appears in a tuple representing a global state we simply say that the component is in that global state.[2] This first step is taken from the existing approach [2] for the synthesis of centralized adaptors. As already mentioned in Section 1, whenever P_{LTS} ensures itself deadlock-freeness, such a step is not required. For the sake of presentation we will always assume that K exists. The novel contribution of this paper is represented by the second step. (2) If K has been generated, our method explores it looking for those states representing the *last chance* before entering into an execution path that leads to deadlock. The restriction with respect to the specified desired behavior is realized by visiting P_{LTS}. The aim is to split and distribute P_{LTS} in such a way that each local adaptor knows which actions the wrapped component is allowed to execute. The sets of last chance states and *allowed actions* are stored and, subsequently, used by the local adaptors as basis for correctly exchanging synchronizing information. In other words, the local

[2] In general, a component might be in more than one global state.

adaptors interact with each other (by means of both standard and additional communication) to perform the correct behavior of K with respect to deadlock-freeness and P_{LTS}. Decentralizing K, the local adaptors preserve parallelism of the components forming the system. In the following subsection we formalize the second step of our method by also providing its correctness.

3.2 Second Step Formalization

As described before, the second step gets in input: (i) the set $\{AC_1, \ldots, AC_n\}$, (ii) K and (iii) P_{LTS}. In order to detect deadlocks, our approach explores K and looks for sinks. A deadlock state (see Figure 2) is in fact a sink of K. We call *Forbidden States* (FS) the set of deadlock states[3] and all the ones within *forbidden paths* necessarily leading to them. A forbidden path in K is a path that starts at a node which has no transitions that can avoid a forbidden state and thus necessarily ends in a sink (see for instance Figure 2). The states in FS can be avoided by identifying a specific subset of K's states that are critical with respect to FS (see for instance S in Figure 2). In this way we can avoid to store the whole graph at runtime as we just need to store the critical states. More precisely, in order to avoid a state in FS, we are only interested in those nodes representing the last chance before entering into a forbidden state.

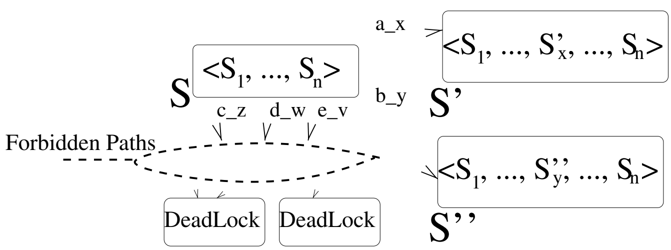

Fig. 2. A last chance node S of K

The last chance nodes have some outgoing edges leading to a forbidden state, the *dead* edges, and other ones, the *safe* edges (see for instance the edges labeled with *a_x* and *b_y* in Figure 2). According to the labels of the dead edges we store in the local adaptors associated to the corresponding components the last chance node, and the *critical action* that each component should not perform in order to avoid a state in FS (in Figure 2, the action c is critical for the component z). From the implementation point of view, each local adaptor F_{C_i} uses a table $F_{C_i}^{LC}$ (*Last Chance table* of F_{C_i}) of pairs <*last chance state of K, critical action of* AC_i>. Thus, once all the graph has been visited, each local adaptor knows the critical actions of the corresponding component. Before a component can perform a critical action its local adaptor has to ask permissions to the other components

[3] Abusing notation, sometimes we refer to the states as nodes.

(see procedure *KVisit*). The following procedure computes and distributes the last chance node tables among the local adaptors. Given in input the centralized glue adaptor K of n components, the procedure makes use of the following variables: $F^{LC}_{C_i}$ is the table of last chance nodes associated to the component C_i; $Flag\_Forbidden_S$ is a flag to check whether the current node S eventually leads to deadlock or not; $Dead\_Sons_S$ counts the number of sons of the current node S that eventually lead to forbidden states of K; $Safe\_Sons_S$ counts the number of sons of the current node S that may lead to allowed states of K.

```
procedure KVisit(state of K: S;)
for each i := 1 to n do
    F^{LC}_{C_i} := ∅;
end for
Flag_Forbidden_S := False;
Dead_Sons_S := 0;
Safe_Sons_S := 0;
mark S as Visited;
for each son S' of S do
    if the edge (S, S') is not visited then
        mark the edge (S, S') as Visited;
        if S' is not visited then
            KVisit(S');
        end if
        if Flag_Forbidden_S' then
            Dead_Sons_S++;
        else
            Safe_Sons_S++;
        end if
    end if
end for
if Safe_Sons_S == 0 then
    Flag_Forbidden_S := True;
end if
if Safe_Sons_S > 0 && Dead_Sons_S > 0 then
    for every dead edge, let α_x be the associated action, F^{LC}_{C_x} = F^{LC}_{C_x} ⋃ < S, α >;
end if
```

Before starting a critical action (that might lead to a state in FS), a local adaptor has to verify (by performing additional communication) if the global state represents a last chance state with respect to that action. Since at runtime we do not store K, this verification is made by enquiring the other local adaptors about the states of the corresponding components, hence deriving the appropriate consequences. If a component is not in the enquired last chance state, its associated local adaptor immediately replies ensuring that the component will not reach such a state. In some way it is *self-blocked* with respect to the enquired state. If the component is already in the enquired last chance state or it is interested in reaching it, its local adaptor defers the answer and hence, it attempts to block the enquiring local adaptor. The only case in which an enquiring local adaptor has to ask the permission to all the others is when the global state is exactly a last chance one. Once the enquiring local adaptor receives an answer it allows its corresponding component to proceed with its standard communication by delegating the critical action. After that, it sends a message to unblock all the other local adaptors previously enquired (additional communication). The unblock message is needed because once a local adaptor allows an enquiring one to perform a critical action, it ensures also that it will not reach the last chance state before receiving an unblock message with respect to such a state (see code lines 7 and 14 of Procedure *Ack* below). In practice it is self-blocked just with respect to the enquired state.

Concerning P_{LTS}, we visit and distribute it among the local adaptors (see Procedure *PVisit* reported below). Such a distribution is made by means of another table $F_{C_i}^{UA}$ for each local adaptor F_{C_i} (called *Updating and Allowed actions table* of F_{C_i}) of tuples <*state of* P_{LTS}*, allowed action of* AC_i*, state of* P_{LTS}*, set of components, set of components* >. The first three elements of each tuple represent an edge of P_{LTS}. The fourth (fifth) is the set of *active components*, i.e., the ones that can perform some action "matching" with a transition outgoing from the state of P_{LTS} specified by the first (third) element of each tuple. By means of *PVisit* each local adaptor knows its allowed actions that can change the state of P_{LTS}. Moreover, a local adaptor knows also which are the active components that can move and which must be blocked according to the current state of P_{LTS}. Let us assume that a component C_i is going to perform an action contained in the table $F_{C_i}^{UA}$. If it can proceed according to the current state of P_{LTS}, then all the other active components are blocked by sending a blocking message to the corresponding local adaptors. Once C_i has performed the action, all the components that can move in the new state of P_{LTS} are unblocked. Note that if an action of an active component does not change the state of P_{LTS}, it can be performed without exchanging messages among the system components, hence maintaining pure parallelism (this is realized by Procedure *Ask*, code line 34). The setup of the Last Chance and the Updating and Allowed action tables is realized by means of two procedures *KVisit* (see above) and *PVisit* (see below). They are depth-first visits of K and P_{LTS}, respectively. These procedures are executed at design-time in order to setup the corresponding tables. After their execution, K and P_{LTS} can be discarded. Procedures *Ask* and *Ack*, instead, implement the local adaptors interactions at runtime. Referring to the table of updating allowed actions, let $Lookahead(state\ of\ P_{LTS} : p)$ be a procedure that given a state p of the P_{LTS} automaton, returns the set of components that are allowed to perform an action in the state p. The following procedure distributes P_{LTS} among the local adaptors. Given in input P_{LTS} referred to n components, the procedure makes use of the following variables: $Active\_Components$ is the set of components that are allowed to make a move in the current state p of P_{LTS}; $Next\_Components$ is the set of components that must be allowed to move once the current state of P_{LTS} has changed; $F_{C_i}^{UA}$ is the table of updating and allowed actions of the component C_i.

procedure PVisit(state of P_{LTS}: p;)
for each $i := 1$ **to** n **do**
  $F^{UA}_{C_i} := \emptyset$;
end for
$Active\_Components := Lookahead(p)$;
$Next\_Components := \emptyset$;
mark p as Visited;
for each son p' of p **do**
  if the edge (p, p') is not visited **then**
    mark the edge (p, p') as Visited;
    $Next\_Components := Lookahead(p')$;
    for each $C_i \in Active\_Components$ allowed to perform an action α by the label of the edge (p, p') **do**
      $F^{UA}_{C_i} := F^{UA}_{C_i} \bigcup < p,\ \alpha,\ p',\ Active\_Components,\ Next\_Components >$;
      if p' is not visited **then**
        $PVisit(p')$;
      end if
    end for
  end if
end for

Once this procedure is performed, each local adaptor knows in which state of P_{LTS} it can allow the corresponding component to perform a specific action. Moreover, once the component performs such an action, it knows also which are the components that must be blocked and which ones must be unblocked in order to respect the behavior specified by P_{LTS}.

In the following we describe how a local adaptor uses the tables to correctly interact with each other (i) in a deadlock-freeness and (ii) as specified by P_{LTS}. On the exchanged messages, when needed, we use the standard time-stamps method in order to avoid problems of synchronization. In this way an ordering among dependent messages is established and starvation problems are also addressed. Note that also a priority ordering among components is *a priori* fixed. This solves ordering problems concerning messages with the same time-stamps. A local adaptor, whose current time-stamp is TS, whenever receives a message with associated a time-stamp ts, it makes use of the following simple procedure in order to update TS.

procedure UpTS(timestamp: ts;)
if $TS < ts$ **then**
  $TS := ts + 1$;
end if

Let C_x be an active component that is going to perform action α (i.e., in AC_x there is a state transition labeled with α and α does not collide with respect to P_{LTS}). The associated local adaptor F_{C_x} checks if α is either (i) a critical action (i.e., α appears in $F^{LC}_{C_x}$) or (ii) an updating and allowed action (i.e., α appears in $F^{UA}_{C_x}$). If it is not, F_{C_x} delegates α with associated the current time-stamp TS

increased by 1 to synchronize itself with the rest of the system. If (i) then F_{C_x} enters in the following procedure in order to ask for the permission to delegate α. This is done by checking if for any pair $< S, \alpha > \in F^{LC}_{C_x}$ there is at least one local adaptor F_{C_y} whose corresponding component C_y is not in S. If (ii) then F_{C_x} enters in the following procedure in order to try to block all the active components and after having performed α, it unblocks the components that can be activated with respect to the new state reached over P_{LTS}.

procedure Ask(action: α;)

Let C_x be the current component that would perform action α and let S_{C_x} be its current state and p be the current state of P_{LTS};
Let $< t_i >^{UA}_x$ be the i-th tuple contained in the table $F^{UA}_{C_x}$ and $< t_i >^{UA}_x [j]$ be its j-th element;
$flag\_forbidden := 0$;
if $\exists\ i \mid < t_i >^{UA}_x [1] == p$ && $< t_i >^{UA}_x [2] == \alpha$ **then**
  if α appears in some pair of $F^{LC}_{C_x}$ **then**
    for every entry $< S, \alpha > \in F^{LC}_{C_x}$ **do**
      $i := 1$;
      $TS++$;
      while no "ACK, α,ts" received && $i \leq n$ **do**
        Let $S \equiv < S_{C_1}, \ldots, S_{C_n} >$; F_{C_x} asks to local adaptor F_{C_i} if it is in or approaching the state S_{C_i} with associated TS;
        $i++$;
      end while
      if $i > n$ **then**
        WAIT for an "ACK, α, ts" message
      end if
      $UpTS(ts)$;
      if $i > n$ **then**
        $i := n$;
      end if
      for $j := 1$ **to** i **do**
        send "UNBLOCK, α, TS" to F_{C_i};
      end for
    end for
  end if
  $TS++$;
  if $< t_i >^{UA}_x [1] != < t_i >^{UA}_x [3]$ **then**
    for each component $C_j \in < t_i >^{UA}_x [4]$ **do**
      send "BLOCK, TS" to F_{C_j};
    end for
    perform action α;
    for each component $C_j \in < t_i >^{UA}_x [5]$ **do**
      send "UNBLOCK, $< t_i >^{UA}_x [3]$, TS" to F_{C_j};
    end for
  else
    perform action α;
  end if
end if

Note that, by code line 13, the present local adaptor is self-blocked till some local adaptor gives the permission to proceed, i.e. an "ACK". The *"UNBLOCK"* messages of code line 20 say to all the local adaptors that were blocked with respect to the enquired forbidden states, to proceed. The *"UNBLOCK"* messages of code line 31 are instead to unblock components due to the change of state of P_{LTS} occurred after having performed action α. On the other hand, when a local adaptor receives a request for a permission, after having given such a permission, it is implicitly self-blocked in relation to the set of states it was enquired for. The following procedure describes the *"ACK"* messages exchanging method.

procedure Ack(last chance state: S; action: α; timestamp: $ts1$;)

1: Let F_{C_y} be the local adaptor (performing this Ack) that was enquired with respect to the state S and the action α that C_x would perform; let S'_{C_y} be the current state of F_{C_y} and S''_{C_y} be the state that F_{C_y} would reach with the next hop.
2: $UpTS(ts1)$;
3: **if** $S'_{C_y} \neq S$ && F_{C_y} didn't ask the permission to get in S **then**
4:    send "ACK, α, TS" to F_{C_x} that allows C_x to perform the action α;
5:    **if** $S''_{C_y} == S$ **then**
6:       WAIT for "UNBLOCK, α, $ts2$" from F_{C_x};
7:    **end if**
8:    S''_{C_y} := next desired state of F_{C_y};
9: **else**
10:    once $S'_{C_y} \neq S$ send "ACK, α, TS" to F_{C_x} that allows C_x to perform the action α;
11:    **if** no "UNBLOCK, α, $ts2$" from F_{C_x} has been received **then**
12:       WAIT for it;
13:    **end if**
14:    $UpTS(ts2)$;
15: **end if**

The *"WAIT"* instructions of code lines 6 and 12 block the current local adaptor in order to not allow the corresponding component to enter in a forbidden state. Note that, while the *"UNBLOCK"* message has a one-to-one correspondence, that is, for each message there is a receiver waiting for it, the *"ACK"* message can be sometimes useless. In fact a local adaptor needs just one *"ACK"* message in order to allow the corresponding component to proceed with the enquired critical action. All the other possible *"ACK"* messages are ignored.

3.3 Correctness

We now provide the correctness of our method by proving that assuming K and P_{LTS}, the method synthesizes local adaptors that (i) allow the composed system to be free from deadlocks and (ii) allow P_{LTS} to be exhibited.

We prove (i) by focussing on the last chance nodes. Note that, since the synthesis of K is correct as proved in [2], we can assume that the last chance nodes are correctly discovered by means of the procedure *KVisit* that performs a standard depth-first visit. Thus, our proof can be reduced to show that the local adaptors disallow the system to reach a forbidden path. Note that, by construction, such a path can be undertaken only through a last chance node by performing an action that labels one of its outgoing dead edges. Let us assume by contradiction that the component z can perform the critical action c from the last chance state S, and that S has an outgoing dead edge labeled by c_z (see for instance Figure 2). Since, as already noticed, the last chance nodes are correctly discovered, when procedure *KVisit* is visiting S, it stores in F_z^{LC} the tuple $< S, c >$. At runtime, whenever the component z would perform action c, F_z checks if c is a critical action by means of code line 4 of its *Ask* procedure. It then starts to ask the permission (at least an "ACK" message) to all the other components by means of the "while" cycle of code line 8 of the same procedure. Each enquired local adaptor F_{C_i}, by the *Ack* procedure, checks if the current state of the corresponding component C_i is in S. If it is, it does not reply to z till it does not change status (code line 10 of the *Ack* procedure). In doing so, until the system state remains S, no local adaptor will reply to F_z. Since F_z is blocked on code line 13 of the *Ask* procedure till no "ACK" message is received, a contradiction follows by observing that action c can be performed by z at code line 29 of the same procedure.

To prove (ii), let us assume by contradiction that the component x performs the action a when this is not allowed by P_{LTS}, that is, the current state S_P of P_{LTS} has no outgoing edge labeled by a_x. First of all, in order for a component to be active, either its local adaptor has received an "UNBLOCK" message from some other local adaptor (by means of code line 31 of the *Ask* procedure) or the system is just started and F_x^{UA} has some entry with $S0$ (the initial state of K) as first element. In both cases each time a component is active, its local adaptor knows exactly which is the current P_{LTS} state. By construction, x can perform action a if there exists an entry in F_x^{UA} whose first element matches with the current state of P_{LTS} and whose second element matches with a (see code line 3 of the *Ask* procedure). The contradiction follows by observing that such an entry was obtained by visiting P_{LTS} hence, by construction, there must exist an outgoing edge whose label matches with a_x from the node labeled by S_P.

4 Running Example

In this section we show our approach at work by means of a running example. This example concerns the semi-automatic assembly of a distributed client-server system made of four components, two servers (denoted by *C1* and *C2*) and two clients (denoted by *C3* and *C4*). The behavioral specification of *C1*, *C2*, *C3* and *C4* (shown in Figure 3 in form of LTSs) has been borrowed from an industrial case study described in [6]. *C1* (resp., *C2*) provides two methods p and *FreeP* (resp., *p1* and *FreeP1*). Moreover, *C2* provides also a method *Connect*. By referring to the method described in Figure 1, by taking into account the LTSs of *C1*, *C2*, *C3* and *C4*, we automatically synthesize a model of the centralized glue adaptor K. This is done by using SYNTHESIS and performing the approach described in [2]. Finally, by taking into account the LTS specification of the desired behavior that the composed system must exhibit, we mechanically distribute the correct behavior of K in a set of local adaptors *f1*, *f2*, *f3* and *f4*.

4.1 Our Approach at Work

Figure 3 shows the LTSs of *C1*, *C2*, *C3* and *C4*. Within the LTS of a component, a message $?m$ ($!m$) denotes a received (sent) request or notification labeled with m. The state with an incoming arrow denotes the initial state. For instance, *C1* (*C2*), from its initial state, receives a request of p (*p1*) followed by a request of *FreeP* (*FreeP1*).

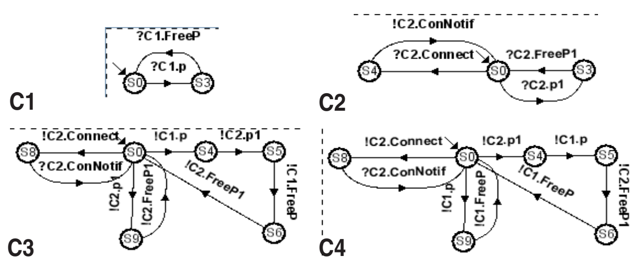

Fig. 3. Components' LTSs

Moreover, *C2* from the initial state can receive a request of *Connect* and, subsequently, replies to it by means of the notification *ConNotif.* The interaction behavior of the clients $C3$ and $C4$ can be easily understood by simply looking at Figure 3.

Figure 4 shows part of the LTS of K. Within K, a message $?m\_j$ ($!m\_j$) denotes a request or notification labeled with m and received (sent) from (to) Cj. The state $S0$ with an incoming arrow denotes the initial state. K contains filled nodes, which denote deadlocks. Deadlocks might occur, e.g., because of a "race condition" among $C3$ and $C4$. In fact, one client (e.g., *C3*) performs a request of *p1* (see the sequence of transitions from the state $S0$ to $S2$ in K) and waits for performing the request p while the other client (i.e., *C4*) performs p (see the sequence of transitions from the state $S2$ to $S16$ in K) and waits *p1*. In this scenario $C3$ and $C4$ are in the state $S4$ of their LTSs (see Figure 3). Since none of the clients performs the corresponding *FreeP* method before having performed both p and *p1*, a deadlock occurs (see the filled state $S16$ in K). Note that, following the sequence of transitions from the state $S0$ to $S12$ leads to the symmetric scenario. By referring to Figure 4, the states $S15$ and $S29$ are filled since they can only lead to deadlocks. $S2$ and $S12$ are last chance states. The paths from $S2$ to $S16$ and from $S12$ to $S16$ are forbidden paths and $FS = \{S15, S16, S29\}$. Following we show the tables of last chance nodes used by each local adaptor as generated by procedure *KVisit*. For both $S2$ and $S12$ there is just one critical action that leads to a deadlock. This is translated by procedure

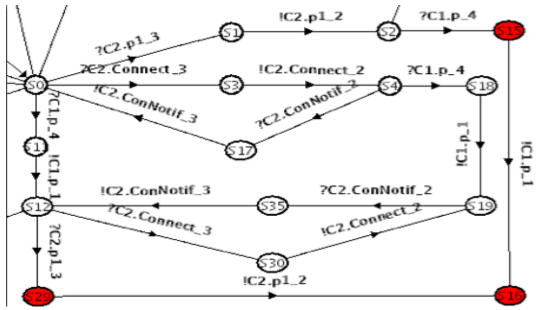

Fig. 4. Part of the LTS of the centralized glue adaptor K. The filled nodes belong to deadlock paths.

$KVisit$ in storing the entries $< S12 \equiv < S3_{C1}, S0_{C2}, S0_{C3}, S9_{C4} >, !C2.p1 >$ in F^{LC}_{C3} and $< S2 \equiv < S0_{C1}, S3_{C2}, S9_{C3}, S0_{C4} >, !C1.p >$ in F^{LC}_{C4}, respectively. In this way, each time the component $C3$ ($C4$) performs the action $!C2.p1$ ($!C1.p$), the corresponding local adaptor has to check that the global status is not $S12$ ($S2$). After performing $KVisit$ to derive the last chance nodes table for each local adaptor, SYNTHESIS performs $PVisit$ to derive the updating and allowed actions table for each local adaptor. This is done by taking into account the LTS specification P_{LTS} (see Figure 5). We recall that these tables are needed to distribute P_{LTS} among the local adaptors.

Fig. 5. The system's desired behavior specified by P_{LTS}

In our context, P_{LTS} describes (at an high-level) a desired behavior for the composed system. Each node is a state of the system. The node with the incoming arrow is the initial state. The syntax and semantics of the transition labels is the same of the LTS of K except for two kinds of action: i) a universal action (i.e., $?true\_$) which represents any possible action, and ii) a negative action (e.g., $! - C2.Connect\_4$ in Figure 5) which represents any possible action different from the negative action itself. P_{LTS} specifies that it is mandatory for $C3$ to perform a *Connect* before performing *p1* (see the self-transition on the state $S1$ and the transition from $S1$ to $S2$ showed in Figure 5). The self-transition on $S1$ is the logical AND of the actions in the action list delimited by '{' and '}'. The semantics of this self transition is that the current state of P_{LTS} (i.e., $S1$) remains unchanged until an action different from $!C2.Connect\_3$, $!C2.FreeP1\_3$ and $!C2.p1\_3$ is performed. When $C3$ performs $!C2.Connect\_3$ the current state of P_{LTS} becomes $S2$. Then, while being in the state $S2$ all the components but

$C4$ simultaneously execute unconstrained (see the negative self-transition on the state $S2$ in Figure 5). Finally, *FreeP1* will be performed by $C3$ to allow another client to perform *p1* (see the transition from $S2$ to $S1$ showed in Figure 5). Following we show the tables of updating and allowed actions used by each local adaptor as generated by the procedure *PVisit*. Denoting by "$*$" any possible value of a specified scope, P_{LTS} is translated by procedure *PVisit* in storing the entries $< *, *, *, *, * >$ in F^{UA}_{C1}, F^{UA}_{C2}; while $< S0, !C2.Connect, S1, *, * >$, $< S1, !C1.p, S1, *, * >$, $< S1, !C2.p1, S1, *, * >$, $< S1, !C1.FreeP, S1, *, * >$, $< S1, !C2.Connect, S1, *, * >$, $< S1, !C2.FreeP1, S0, *, * >$ in F^{UA}_{C3} and $< S0, *, S0, *, * >$, $< S1, !C1.p, S1, *, * >$, $< S1, !C2.p1, S1, *, * >$, $< S1, !C1.FreeP, S1, *, * >$, $< S1, !C2.FreeP1, S1, *, * >$ in F^{UA}_{C4}. Note that, when during the runtime, the state of P_{LTS} changes from $S0$ to $S1$ by means of the action $!C2.Connect$ performed by $C3$, F_{C3} informs F_{C4} of the new state of P_{LTS} by means of the "UNBLOCK" message of code line 31 of its *Ask* procedure. Consequently F_{C4} knows that in such a state $C4$ cannot perform $!C2.Connect$ since the entries $< S1, !C2.Connect, *, *, * >$ are not present in F^{UA}_{C4}. Once the LC and UA tables are filled, the interactions among local adaptors can start by means of procedures *Ask* and *Ack*. In order to better understand such an interaction, let us consider the sequence of messages that according to the glue coordinator of Figure 4 leads the global state from $S0$ to $S15$. Note that a forbidden path starts from $S15$. The first message is sent by F_{C3} to F_{C2} in order to ask the resource $p1$. This is allowed by the entry $< *, *, *, *, * >$ contained in F^{UA}_{C2}. It means, in fact, that $C2$ can perform any action from any global state according to P_{LTS}. When from $S1$, F_{C4} would perform $?C3.p\_4$, according to the entry $< S2 \equiv < S0_{C1}, S3_{C2}, S9_{C3}, S0_{C4} >, !C1.p >$ contained in F^{LC}_{C4}, it has to check if the current global state is $S2$ in order to not incur in the forbidden path that starts from $S15$. According to procedure *Ask*, it starts to ask the permission to all the other local adaptors (see code lines 8 of procedure *Ask*). Since the current state is exactly $S2$ it will not receive any answer from the other local adaptors (this is accomplished by code line 3 of procedure *Ack* since if the enquired local adaptor is in the enquired state, the if condition is not satisfied and the ACK message cannot be sent). Such a situation changes as soon as some component changes its status hence unblocking F_{C4} (see code line 12 of procedure *Ack*). Note that since such interaction concern just an action of $C4$, by construction, this allow all the other local adaptors to continue their interaction according to P_{LTS} hence maintaining the eventual parallelism.

5 Related Work

The approach presented in this paper is related to a number of other approaches that have been considered by researchers. For space reasons, we discuss only the ones closest to our approach.

In [7] a game theoretic approach is used for checking whether incompatible component interfaces can be made compatible by inserting a converter between them. This approach is able to automatically synthesize the converter. Contrarily to what we have presented in this paper, the synthesized converter is a centralized adaptor.

Our research is also related to [8] in the area of protocol adaptor synthesis. The main idea is to modify the interaction mechanisms that are used to glue components together so that compatibility is achieved. This is done by integrating the interaction protocol into components. However, they are limited to only consider syntactic incompatibilities between the interfaces of components and they do not allow to automatically derive a distributed implementation of the adaptor. Note that our approach can be easily extended to address syntactic incompatibilities between component interfaces. We refer to [2] for details concerning such an extension.

In another work by some of the authors [6], it is showed how to generate a distributed adaptor by exploiting an approach to the definition of distributed Intrusion Detection Systems (IDS). Analogously to the approach described in this paper, the distributed adaptor is derived by *splitting* a pre-synthesized centralized one in a set of local adaptors (each of them local to each component). The work in [6] represents a first attempt for distributing centralized adaptors and it has two main disadvantages with respect to the approach described here: (a) the method requires a more complex (in time and space) process for pre-synthesizing the centralized adaptor. In fact, it does not simply model all the possible component interactions (like our centralized glue adaptor), but it has to model the component' interactions that are deadlock-free and that satisfies the specified desired behavior (P_{LTS}). In that approach, in fact, the glue adaptor is generated and, afterwards, a suitable synchronous product with P_{LTS} is performed. This longer process with respect to the current approach might also lead to a final bigger centralized adaptor. (b) The adopted solution realize distribution but not parallelism. The distributed local adaptors realize, in fact, the strict distribution of the obtained centralized adaptor by means of the pre-synthesizing step. This means that, since the centralized coordinator cannot parallelize its contained traces, the interactions of the local adaptors maintain this behavior.

In [9], the authors show how to monitor safety properties locally specified (to each component). They observe the system behavior simply raising a *warning message* when a violation of the specified property is detected. Our approach goes beyond simply detecting properties by also allowing their enforcement. In [9] the best thing that they can do is to reason about the global state that each component *is aware of*. Note that, such a global state might not be the actual current one and, hence, the property could be considered guaranteed in an *"expired"* state. Furthermore, they cannot automatically detect deadlocks.

6 Conclusion and Future Work

In this paper we have presented an approach to automatically assemble concurrent and distributed component-based systems by synthesizing distributed adaptors. Our method extends our previous work described in [2] that permitted to automatically synthesize centralized adaptors for component-based systems. The method described in this paper allows us to derive a distributed implementation of the centralized adaptor and, hence, it enhances *scalability*, *fault-tolerance*, *efficiency*, *parallelism* and *deployment*. We successfully validated the approach on a running example. We have also implemented it as an extension of our SYNTHESIS tool [2]. The state explosion phenomenon suffered by the centralized glue adaptor K still remains an open problem. K is required to detect the last chance nodes that are needed to automatically avoid deadlocks. Indeed when the deadlocks can be solved in some other ways (e.g., using timeouts) or P_{LTS} ensures their avoidance, generating K is not needed. Local adaptors may add some overhead in terms of messages exchanged. In practical cases, where usually many parallel computations are allowed, the overhead is negligible since additional communications are much less then standard ones. As future work, whenever K is required, an interesting research direction is to investigate the possibility of directly synthesizing the implementation of the distributed adaptor without producing the model of the centralized one. Further validation by means of a real-scale case study would be interesting.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (2004)
2. M.Tivoli, M.Autili: Synthesis: a tool for synthesizing “correct” and protocol-enhanced adaptors. RSTI L’Objet journal **12** (2006) 77–103
3. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
4. Ben-Ari, M.: Principles of concurrent and distributed programming. Prentice Hall (1990)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
6. P.Inverardi, L.Mostarda, M.Tivoli, M.Autili: Synthesis of correct and distributed adaptors for component-based systems: an automatic approach. In: Proc. of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)-Long Beach, CA, USA. (2005)
7. Passerone, R., de Alfaro, L., Heinzinger, T., Sangiovanni-Vincentelli, A.L.: Convertibility verification and converter synthesis: Two faces of the same coin. In: Proc. of International Conference on Computer Aided Design (ICCAD) - San Jose, CA, USA. (2002)
8. Yellin, D., Strom, R.: Protocol specifications and component adaptors. ACM Trans. on Programming Languages and Systems **19**(2) (1997) 292–333
9. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proc. of International Conference on Software Engineering (ICSE) - Edinburgh - UK. (2004).