Automatic Failures-Free Connector Synthesis: An Example

Paola Inverardi and Massimo Tivoli

University of L'Aquila
(Dip. Informatica)
via Vetoio 1, 67100 L'Aquila, Italy
{inverard,tivoli}@di.univaq.it

Abstract. Many software projects are based on the integration of independently designed software components that are acquired on the market rather than developed within the project itself. This type of components is well known as COTS (Commercial-Off-The-Shelf) components. Nowadays component based technologies provide interoperability and composition mechanisms that cannot solve the COTS components assembly problem in an automatic way. One of the main problems in components assembly is related to the ability to establish properties on the assembly code by only assuming a limited knowledge of the single components properties. Our answer to this problem is a software architecture based approach in which the software architecture imposed on the assembly, allows for detection and recovery of COTS integration anomalies. We build applications by assuming a defined architectural style. Then, we compose a system in such a way that it is possible to check whether and why the system presents some software anomalies (e.g.: deadlock, livelock). Depending on the kind of failures a recovery policy which can avoid the anomalies and obtain a correct assembly can be performed. A tool can then synthesize the assembly code (as a failures-free connector component) to glue together a set of COTS components. This glue code must be synthesized in such a way that (a well defined set of) functional properties required for the composed system are automatically guaranteed. In the paper we briefly describe our approach and then we present its application to an example.

1 Introduction

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. This type of components is well known as COTS (Commercial-Off-The-Shelf) components. One of the main problems in assembling COTS components is related to the ability to establish properties on the assembly code by only assuming a limited knowledge of the single components properties. Our answer to this problem is a software architecture based approach in which the software architecture imposed on the assembly, allows for detection and recovery of COTS integration anomalies. Notably, in the context of

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 184–198, 2004.

component based concurrent systems, COTS components integration may cause deadlocks or other software anomalies within the system [18,1,12,13]. The use of COTS components in system construction presents new challenges to system architects and designers [14]. Building a system from a set of COTS components introduces a set of problems. Many of these problems arise because of the nature of COTS components. They are truly black-box and developers have no method of looking inside the box. This limit is coupled with an insufficient behavioral specification of the component which does not allow to understand the component interaction behavior. Component assembling can result in architectural mismatches when trying to integrate components with incompatible interaction behavior [3]. Thus if we want to assure that a component based system validates specified behavioral properties, we must take into account the component interaction behavior. In this context, the notion of software architecture assumes a key role since it represents the reference skeleton used to compose components and let them interact. In the software architecture domain, the interaction among the components is represented by the notion of software connector.

Our aim is to analyze and fix dynamic behavioral problems that can arise from components composition. We propose an architectural connector-based approach [8,10,9]. The idea is to build applications by assuming a defined architectural style, namely a modified version of the C2 architectural style [15]. We compose a system in such a way that it is possible to check whether and why the system presents some software anomalies (e.g.: deadlock). We can then derive, in an automatic way, directly from the COTS (black-box) components, the code that implements a new component to insert in the composed system. This new component implements an explicit software connector. This code is derived in such a way that the functional properties of the composed system are satisfied. We assume that a behavioral specification of the composed system is available. This specification is provided through *Message Sequence Chart* (MSC) and *High Level MSC* (HMSC) specifications [21,20,22]. Moreover we also assume that a precise definition of the properties to satisfy exists through LTL (*Linear-time Temporal Logic*) formulas definition [2,6,4,5].

The paper is organized as follows. In Section 2 we introduce the problem we want to address and some background. Section 3 presents the technique to allow failures-free connectors synthesis [11] which is then applied in Section 4 on an example. Section 5 concludes.

2 Problem Description

We consider COTS components which are truly black-box components. The properties we want to check are functional properties as deadlock-freeness or general safety and liveness properties which describe expected behaviors of the system. The assembly A depends on the constraints induced by the architectural model the system is based on. The present architectural model, which defines the rules used to build the composed, is a modified version of the C2 architectural style. This modified version of C2 architectural style is called CBA (i.e. *Connector Based Architecture*) style and it is described in detail in [11].

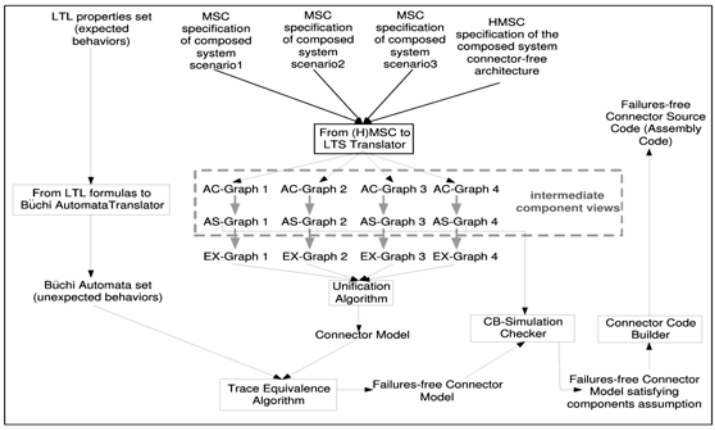

Fig. 1. Tool Architecture for the Automatic Failures-free Connector Synthesis.

It is important to notice that, besides assuming that the system architecture must reflect the rules of a well defined architectural style, we also assume that a system behavioral specification is provided through message sequence chart (MSC) and High level MSC (HMSC) specifications. Then we can derive, in an automatic way, from the MSC and HMSC specification, the behavioral description of each component in the composed system. This behavioral description can be derived by suitable applying the algorithm described in [21,20,22]. Each component behavioral description take a form of graph. Informally our approach is the following. 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 property analysis is performed. If the synthesized connector contains property violating behaviors, a recovery policy is applied. Depending on the kind of property, the analysis of only the connector is enough to obtain a property-satisfying version of the system. Otherwise, the property is due to some component internal behavior and cannot be fixed without directly operating on the component code. In a component based setting in which we are assuming black-boxes components this is the best we can expect to do.

3 Connector Synthesis

Our goal is to develop a tool that performs an automatic software connector synthesis. The aim of this synthesis process is to derive the glue code for the components that constitute the composed system. The glue code is automatically synthesized directly from the partial specification of the composed system (MSCs and HMSC). The composed system automatically synthesized must satisfy the behavioral properties expected from the system designers and architects. In Figure 1 we illustrate the automatic synthesis tool architecture.

We represent with labelled boxes the components of the synthesis tool and with the labels the input and output data for each functional component of the tool. We have represented with the labels followed by a gray bold arrow intermediate data that are necessary to perform some transformations on the output data of the *"From (H)MSC to LTS Translator"* component. We perform these transformations in order to obtain the input data for the *"Unification Algorithm"* component. The gray bold arrows represent these transformations.

We model components as labelled transition systems (LTS) where labels represent messages that the components can input and output on the communication channel. We consider the system as a parallel composition of all components. In literature many approaches to build, in an automatic way, an LTS from an MSC specification exist; we are entirely based on the approach described in [21,20,22]. We adapt these algorithms to build the actual behavior graph (AC-Graph) [8,10,9] of the system components directly from the MSC specifications. A formal definition of AC-Graph is in [8]. Informally we can say that an AC-Graph for a component C describes the interaction behavior of C with its external environment. This environment is modelled as the parallel composition of all the others components in the system. We wish to derive from a component behavior the requirements on its environment that guarantee deadlock-freedom. From the AC-Graphs we can automatically derive the corresponding AS (*ASsumption*) graphs. The AS-Graph is different from the corresponding AC-Graph only in the arcs labels. Actually these labels are symmetric since they model the deadlock-free environment as each component expects it. This is true because we assume synchronous communication between components of the system. Given the CBA style [8], the component environment can only be represented by one or more connectors, thus we refine the definition of AS-Graph into a new graph, the EX-Graph, that represents the behavior that the component expects from the connector. 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. Actually in the EX-Graph of a component C we have actions on a communication channel that is unknown for C and actions on a communication channel that links C with the connector. The global connector behavior will be derived by taking into account all the EX-graphs. This will be done through a sort of unification algorithm [8]. The role of the connector is to route every component request to the request receiver component. Then it returns the request response to the component which fired the request. We automatically synthesize a model of the behavior of the connector which contains all the possible request routing policies. Then we perform analysis of properties and recovery. This means that we could allow a designer to assign a precise scheduling policy to the connector. Referring to the usual model checking approach [2,7], we can think of defining the properties that the system must satisfy by using *Linear-time Temporal Logic* (LTL) formulas [6,4,5]. We can specify the set P of properties that describe the expected behaviors of the system. The synthesis tool uses the set of properties P to identify connector graph portions that do not satisfy the requested routing policy. Therefore every property in P is used to check if the connector contains an unexpected behavior. For each property

$p_i \in P$ the synthesis tool derives the Büchi Automaton [2,6,4] BA_i corresponding to the LTL formula $!p_i$, where the symbol ! is the logical negation operator. Then the tool verifies if in the connector graph a (possibly infinite) arc labels sequence t (an execution trace) exists in such a way that an *accepting* execution of BA_i on the word corresponding to t exists. We have explained formally this test in [11]. Informally, an execution trace t on the connector graph represents a connector behavior that satisfies the negation of the property p_i, thus it represents an unexpected behavior of the connector. At this point the tool could apply two possible recovery strategies to guarantee the desired behavior: i) It does not modify the connector semantics or ii) it modifies the connector semantics. In the first case the tool, in the code derivation step, considers the connector graph portions marked as unexpected behavior, as exceptional running traces. Thus it derives for them a code that implements an exception handling block. In the second case the tool simply cuts the connector graph portions marked as unexpected behavior. Thus the code derivation step does not implement these unexpected running traces. Finally the synthesis tool verifies if the connector ensures the expected behavior for all components connected to it. In this last step the tool compares any AS-Graph with a corresponding connector graph portion by using a sort of state-based equivalence (CB-Simulation) [8]. After we have obtained the connector graph that satisfies a particular routing policy, the tool automatically derives the code of a new component, the connector component, to insert in the composed system. This new component routes the requests of the components connected to it in such a way that, by construction, the composed system behaves as required. It is important to notice, in Figure 1, that every LTS corresponding to a component is expressed by using a data structure that takes the form of automata (the AC-Graph). We can give a textual representation for each AC-Graph by using a process algebra specification (e.g. *Calculus of Communicating Systems* (CCS) [16]). The following are the steps of the algorithm used to build the failures-free connector graph and to derive the failures-free assembly code:

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 all the EX-Graphs EX_i THEN $exit(FAILURE)$ ELSE unify all the EX-Graphs EX_i and put in K the EX-Graphs unification result;
4. FOR EACH property p_i in the set P of properties to be validated, build the Büchi Automaton $BA_{!p_i}$ for the logical negation of p_i;
5. IF a *Connector Graph Execution Trace* t_j exists (in K) in such a way that an accepting execution of $BA_{!p_i}$ on t_j exists, THEN mark the trace t_j for all j;
6. remove from K all the paths that contain a marked *Connector Graph Execution Trace*;
7. FOR EACH component C_i IF $CBSimulation(AS_i, CB_i)$ does not successfully terminate THEN $exit(FAILURE)$ ELSE derive from K the assembly code implementation;
8. $exit(SUCCESS)$;

where:

AS_i is the AS-Graph of the component C_i which is connected to the connector; CB_i is the CB-Graph [11] for C_i. This graph represents the portion of the connector graph that communicates with the component 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 [11] from the portion of the connector behavior regarding the communication with a given component (C_i). $CBSimulation(AS_i, CB_i)$ is performed by not considering the expected environment behaviors (paths of AS_i) corresponding to execution paths in $BA_{!p_i}$. Informally *CB-Simulation* is a notion of simulation based on *stuttering* equivalence [17].

4 Example: The Dining Philosophers

This example is an instance of the well known *Dining Philosophers* problem [19] in which we consider two philosophers and two forks. We present the component structure of the dining philosophers problem in Figure 2.

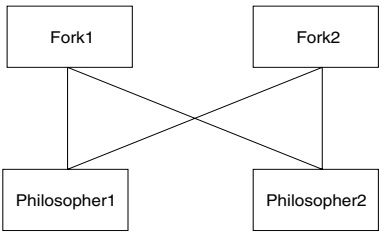

Fig. 2. Architectural View of the Dining Philosophers Problem.

There are 4 components: i) the first fork (**Fork1**), ii) the second fork (**Fork2**), iii) the first philosopher (**Philosopher1**), and iv) the second philosopher (**Philosopher2**). The forks components can iteratively wait for a request, give the fork, and then wait for the fork to be released. The philosophers can non-deterministically choose to ask for a fork, get it, then ask for the other, eat and then release the forks. Since a philosopher to eat needs both the forks it is obvious that in the following scenario a deadlock could arise: 1) component Philosopher1 requests and gets the resource of component Fork1; 2) component Philosopher2 requests and gets the resource of component Fork2; 3) component Philosopher1 requests and waits for the resource of component Fork2; 4) component Philosopher2 requests and waits for the resource of component Fork1. In this scenario Philosopher1 is waiting for Fork2 release. Since Philosopher2 gets the resource of Fork2, this event can be caused only by Philosopher2 who is waiting for Fork1 release. Since Philosopher1 gets the resource of Fork1, this event can be caused only by Philosopher1. Thus each system component is waiting for an event that only another system component can cause. It means a deadlock.

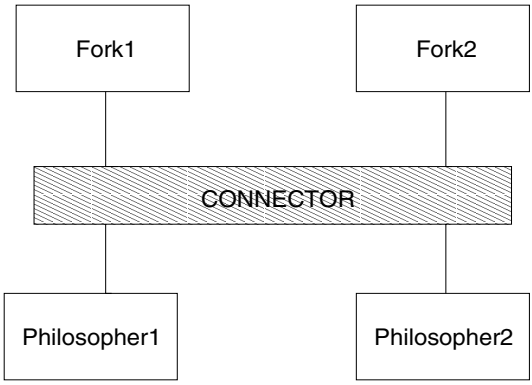

Fig. 3. Architectural View of the Connector-based Dining Philosophers Problem.

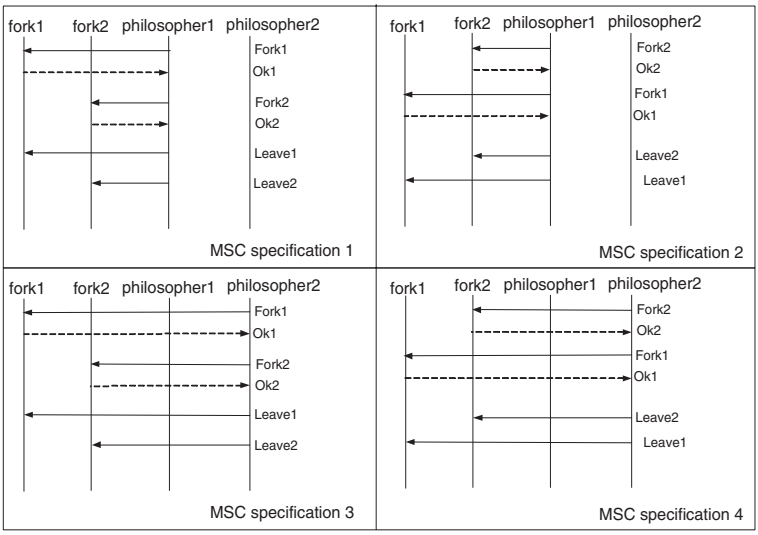

Fig. 4. Basic MSC specification of the composed system.

Now we present the connector based component structure of the dining philosophers problem in Figure 3. The role of the connector is to route every component request to the request receiver component. Then it returns the request response to the component which fired the request. Through the routing policy it implements, the connector can decide to accept or to reject a specific request. Suppose that we can benefit of the behavioral specification of the composed system given in Figures 4 and 5 as MSC and HMSC specification respectively.

By applying the MSC to LTS translation algorithm described in [11], and based on an our purpose adapted version of the algorithm described in [21,20,22], we can obtain the AC-Graphs for each component in the composed system

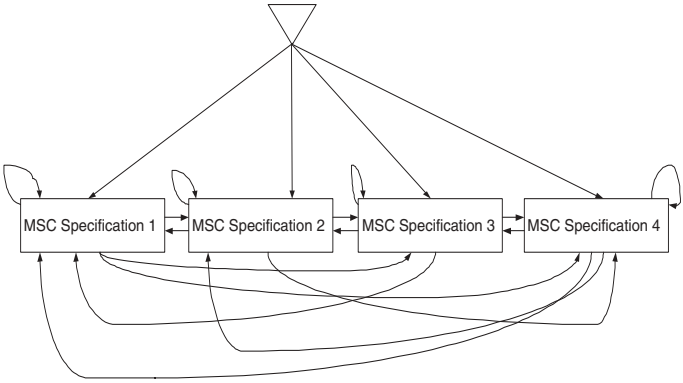

Fig. 5. High Level MSC specification of the composed system.

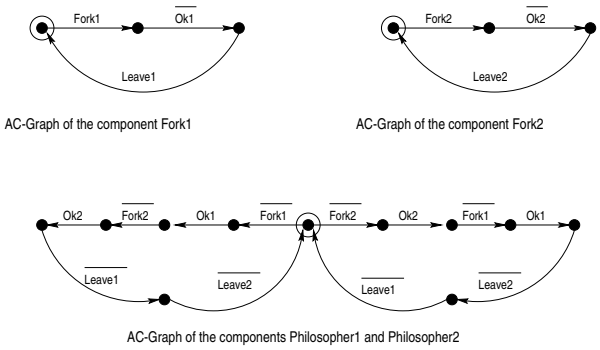

Fig. 6. AC-Graphs of the Dining Philosophers components.

($Fork1$, $Fork2$, $Philosopher1$ and $Philosopher2$). We show these AC-Graphs in Figure 6.

From these graphs we derive the AS-Graphs showed in Figure 7 and from the AS-Graphs we derive the EX-graphs in Figure 8. From the EX-Graphs by applying the unification algorithm described in Section 3 we can obtain the connector graph illustrated in Figure 9.

As showed in Figure 9, we automatically synthesize a model of the behavior of the connector which contains all the possible request routing policies. Then we perform analysis of deadlocks and recovery. The deadlocks analysis step consists of searching for stop nodes in the connector behavioral graph. These nodes represent states in which the system does not perform any action. Thus stop nodes represent deadlock states. The deadlocks recovery step consists of cutting the connector graph branches that lead to stop nodes. In Figure 11 we have showed the deadlock-free connector graph.

It is worthwhile noticing that before the possible deadlocks are fixed the connector contains all possible composed system behaviors. This means that it

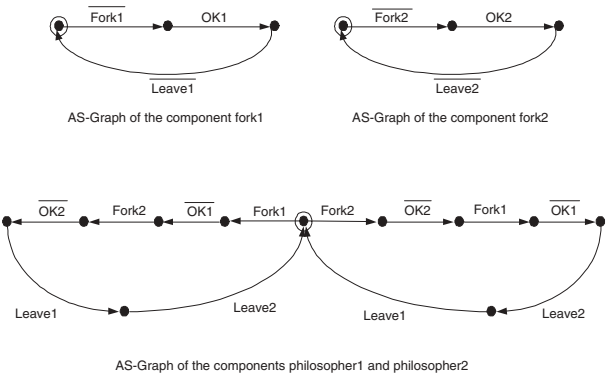

Fig. 7. AS-Graphs of the Dining Philosophers Components.

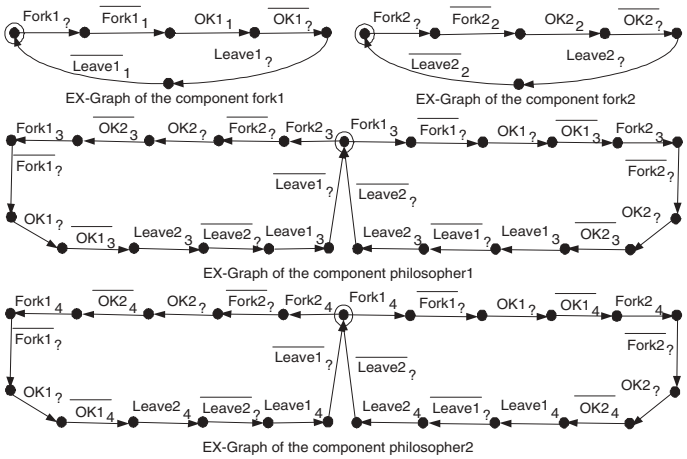

Fig. 8. EX-Graphs of the Dining Philosophers Components.

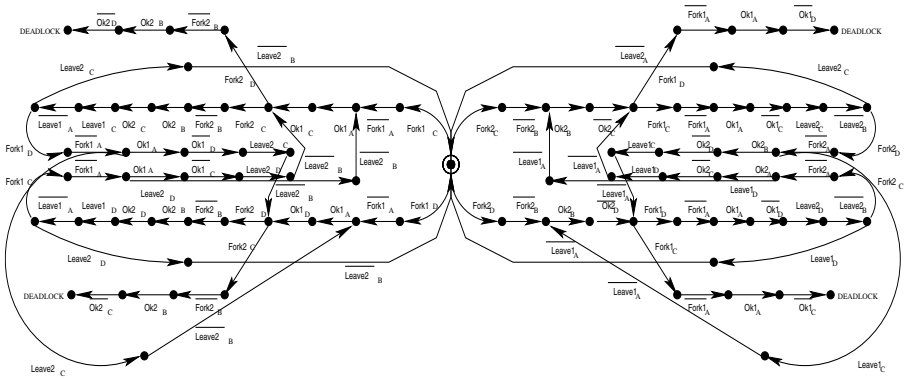

Fig. 9. Automatically Synthesized Connector.

contains all possible routing policies. A designer can now think not only of a deadlock-free routing policy but of a precise scheduling one. For instance he might want the philosophers to eat in turn or that the Philosopher1 always eats twice before Philosopher2. This means that we could allow a designer to assign a precise scheduling policy to the connector. Suppose that the composed system designer has specified the following properties:

- **PROPERTY 1**:
  $LP_1 \equiv \ [](([\overline{Ok1_C} \wedge \overline{Ok2_C}) \longrightarrow \ X([](!([\overline{Ok1_C} \wedge \overline{Ok2_C}))));$

- **PROPERTY 2**:
  $LP_2 \equiv \ [](([\overline{Ok1_D} \wedge \overline{Ok2_D}) \longrightarrow \ X([](!([\overline{Ok1_D} \wedge \overline{Ok2_D}))))$.

With these two properties, the system designer specifies two expected system behaviors that have been specified to avoid this two possible scenarios: i) the first philosopher eats and the second philosopher waits for the forks forever and ii) the second philosopher eats and the first philosopher waits for the forks forever. These two conditions are important for the progress of the system because they implies that both the two philosophers eat an equal number of times. More exactly for equal number of times we means the same number of times in quantity order. By satisfying LP_1 and LP_2 the connector can avoid the starvation. For example with LP_1 the user specifies that for all executions (in the connector model that we are considering), in which the first philosopher has requested and obtained both the two forks then it will have to be always true that the first philosopher does not obtain both the two forks again. Analogously for LP_2.

To limit the size of the paper, we describe the behavioral properties analysis step only for the property LP_1. The following approach is completely equivalent for LP_2. We derive, in an automatic way, a suitable form of the Büchi Automaton $BA_{!LP_1}$ corresponding to the property $!LP_1$ in order to search, in the graph of Figure 11, *Connector Graph Execution Traces* t_j in such a way that an *accepting* execution of $BA_{!LP_1}$ on t_j exists. In Figure 10 we have illustrated the Büchi Automaton, corresponding to $!LP_1$.

The reader, by looking the Figure 11, can easily see that there are infinite *Connector Graph Execution Traces* t_j, corresponding to the two paths ***LP1_Failure1*** and ***LP1_Failure2*** in Figure 11 respectively, in such a way that an *accepting* execution of $BA_{!LP_1}$ on t_j exists. This means that the deadlock-free connector model satisfies the LTL formula $!LP_1$ since there is at least one execution trace in which only the first philosopher requests and obtains the two forks and the second one waits for the forks forever. Thus the derived deadlock-free connector model is a really deadlock-free model of the connector but it does not guarantee the progress of the system.

4.1 Behavioral Failures Recovery

After we have performed the behavioral failures analysis step, we can have possible traces or paths in the connector graph in which the properties are not

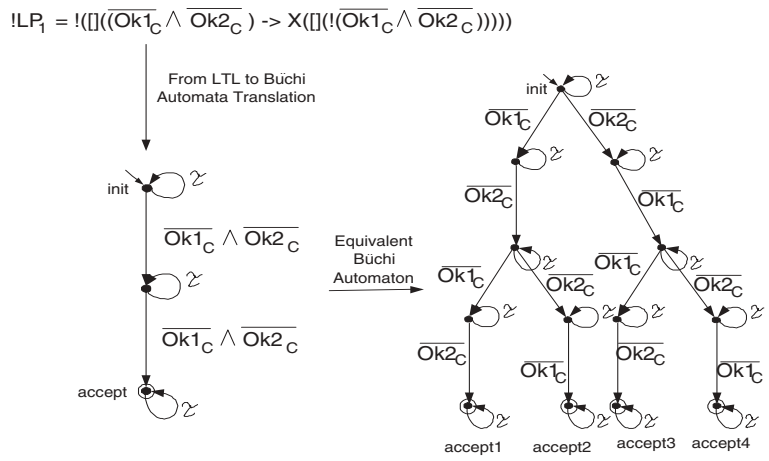

Fig. 10. Equivalent Büchi Automata corresponding to LTL property $!LP_1$.

satisfied (behavioral failures). We propose a strategy based on the elimination of all the paths, in the connector graph, in which we have found a *Connector Graph Execution Trace* accepted by the Büchi Automaton $BA_{!LP_1}$. Then, for every component C_i, we checked if its AS-Graph AS_i is simulated by the CB-Graph CB_i of C_i under the notion of CB-Simulation. We have seen, by performing the analysis step on properties LP_1 and LP_2, that there are two paths, in the deadlock-free connector graph, in which the property LP_1 is not satisfied and there are others two paths in which LP_2 is not satisfied. In Figure 11 we have represented this four paths by coloring gray the nodes in the paths.

We have called the two paths that not satisfy the property LP_1 as ***LP1_Failure1*** and ***LP1_Failure2*** respectively and the two paths that not satisfy the property LP_2 as ***LP2_Failure1*** and ***LP2_Failure2*** respectively. By cutting those paths from the connector graph we obtain the *Deadlock-Free and Progress-Satisfying Connector Graph* of Figure 12.

By the Figure 12, we can see that this model of the connector forces the two philosophers to eat in an alternate way. This is true because every time the first philosopher eats then necessarily the second philosophers will eat too and viceversa. This implies that this model of the connector satisfies the properties LP_1 and LP_2. The reader can easy verifies that for every component C_i the CB-Graph CB_i (of the connector graph of Figure 12) for C_i simulates the AS-Graph AS_i under the notion of CB-Simulation[1].

5 Conclusions and Future Works

In this paper we have presented an example of application of our approach to compose a system out of a set of black-box components in a behavioral failures-free way. A key feature of the approach is that the system architecture follows

[1] Except for the paths of AS_i that are also execution paths of $BA_{!LP_1}$.

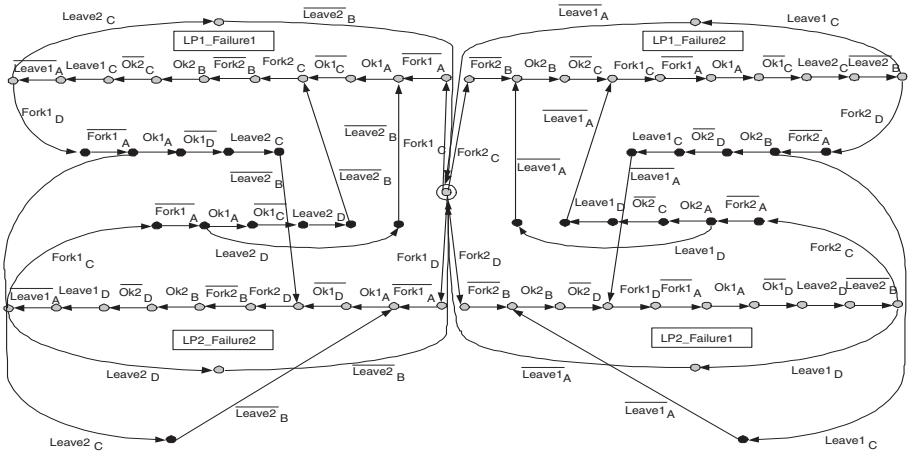

Fig. 11. Deadlock-Free Connector Graph not-satisfying the properties LP_1 and LP_2.

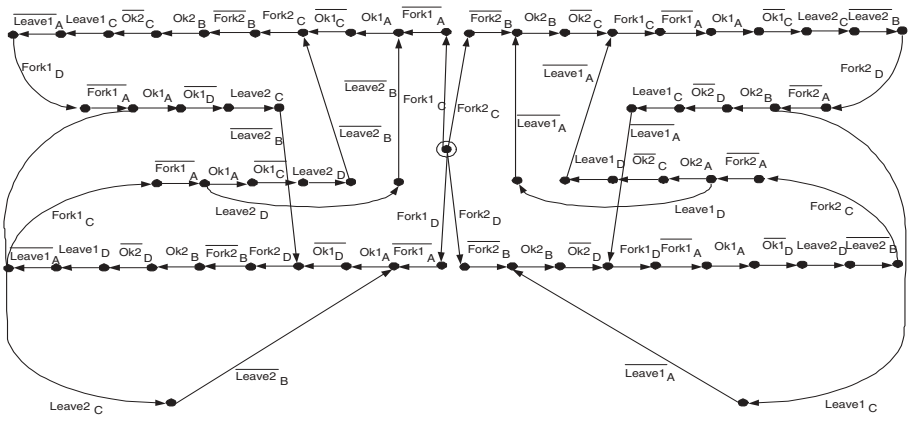

Fig. 12. Deadlock-Free and Progress-Satisfying Connector Graph.

a precise architectural style; this makes the automatic synthesis of connectors possible. Furthermore, the fact that it is known how the components will interact through the connectors makes behavioral failures analysis and/or recovery possible. The approach thus exploits the knowledge of the system architecture in order to improve the quality of the resulting system with respect to architectural mismatches.

As far as complexity is concerned, our approach, contrary to [12], does not improve standard analysis techniques. From this point of view our method shares the same problems of the techniques based on analysis performed on the global system behavioral model. The added-value of our method is the ability to generate a failures-free system, by automatically synthesizing a *safe* connector.

At present we have applied the approach in a real scale context, namely in the context of COM/DCOM applications [9]. At a very high level of description, what has been done is to recast the notions and techniques introduced in the paper in the COM/DCOM context by suitably extending the IDL in order to accommodate our notion of AC-graph in the component interface description. Then the application is build according to the CBA style by letting a COM/DCOM server to act as the synthesized connector. Behavioral failures freedom is then checked by following our definitions. To our respect, it showed the feasibility of our approach and its applicability in commercial component based contexts. As far as components are concerned we only assumed to have a description of the composed system behavior by means of MSCs, which is, in our view, an acceptable hypothesis.

In [10] we mentioned that our method can be applied to multi-layered systems as well. When more than two layers are considered we have to split AS-Graphs according to the two component domains top and bottom. From these we can generate the corresponding pairs of expected graphs which should then be unified. The unification algorithm must be slightly modified to cope with this extension. Our method can have better state complexity results in a layered system, since the connector corresponding to each layer must only consider the subset of components that interact with it.

References

1. B. Boehm and C. Abts. Cots integration: Plug and pray? *IEEE Computer*, 32(1), Jan. 1999.
2. O. G. Edmund M. Clarke, Jr. and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 2001.
3. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), Nov. 1995.
4. P. Gastin and D. Oddoux. Fast ltl to buchi automata translation. *in Proceedings of CAV'01*, 2001.
5. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of liner temporal logic. *in Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95)*, 1995.
6. D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. *RIACS Technical Report 01.21*, 2001.
7. D. Giannakopoulou, J. Kramer, and S. Cheung. Behaviour analysis of distributed systems using the tracta approach. *Journal of Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1):7–35, January 1999.
8. P. Inverardi and S. Scriboni. Connectors syntesis for deadlock-free component based architectures. *16th ASE, Coronado Island, California*, November 2001.
9. P. Inverardi and M. Tivoli. Automatic synthesis of deadlock free connectors for com/dcom applications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE, ACM Press*, Vienna, Sep 2001.
10. P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *to appear on Elsevier Journal of Systems and Software Special Issue on Component-based Software Engineering*, Nov. 2001.

11. P. Inverardi and M. Tivoli. Connectors synthesis for failures-free component based architectures. *Technical Report, University of L'Aquila, Department of Computer Science, http://www.di.univaq.it/tivoli/ffsynthesis.ps*, ITALY, August 2002.
12. P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. *Proceed. FASE 2001, LNCS 2029 pp. 60-75*, April 2001.
13. N. Kaveh and W. Emmerich. Deadlock detection in distributed object system. *8th FSE/ESEC, Vienna*, September 2001.
14. D. Mark, R. Vigder, and J. Dean. An architectural approach to building systems from cots software components. *National Research Council Report Number 40221.*
15. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *In Proceedings of the 1997 Symposium on Software Reusability and Proceedings of the 1997 International Conference on Software Engineering*, May 1997.
16. R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.
17. R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.
18. C. Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow, England, 1998.
19. A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Inc., 1992.
20. S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *in proceeding of the 23rd IEEE International Conference on Software Engineering (ICSE'01)*, Toronto, Canada. May 2001.
21. S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE, ACM Press*, Vienna, Sep 2001.
22. S. Uchitel, J. Kramer, and J. Magee. From sequence diagrams to behaviour models. In *In WTUML: Workshop on Transformations in UML. Satellite event of the European Joint Conferences on Theory and and Practice of Software (ETAPS'01)*, Genova, Italy. April 2001.