Correct and automatic assembly of COTS components: an architectural approach

Paola Inverardi
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila, Italy
inverard@univaq.it

Massimo Tivoli
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila, Italy
tivoli@univaq.it

ABSTRACT

Many software projects are based on the integration of independently designed software components that are acquired on the market rather than developed within the project itself. This type of components is well known as COTS (Commercial-Off-The-Shelf) components. Nowadays component based technologies (COM/ DCOM, Sun's JavaBeans, CORBA) provide interoperability and composition mechanisms that cannot solve the COTS component assembling problem in an automatic way. Notably, in the context of component based concurrent systems, the COTS component integration may cause deadlocks or other software anomalies within the system. In this position paper, we present our approach to contribute to the research in components assembly. Our long term, goal is to develop a tool that synthesize the assembling code 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. We propose an architectural connector-based approach for the assembly problem. The basic idea is to 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). Based on the analysis results a recovery policy which can avoid the anomalies and obtain a correct assembly can be performed.

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. Nowadays component based technologies (COM/ DCOM, Sun's JavaBeans, CORBA) provide interoperability and composition mechanisms that cannot solve the COTS component assembling problem in an automatic way. Notably, in the context of component based concurrent systems, COTS components integration may cause deadlocks or other software anomalies within the system [15, 2, 11, 12]. The use of COTS software components in system construction presents new challenges to system architects and designers [4]. Building a system from a set of COTS components introduces a specified 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 in a multi-component system. Vendors do not provide, with a COTS component, a behavioral specification that is correct and complete with respect to the interaction behavior; they provide an informal specification that is often expressed in natural language. Even if the vendor provides a formal behavioral specification, it refers to the component in a stand-alone context and it specifies nothing about the component interaction behavior. Component assembling can result in architectural mismatches when trying to integrate components with incompatible interaction behavior [5]. Thus if we want to assure that a component based system validates specified dynamic properties, we must refer to 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. Analyzing the actual commercial component based architectural models, connectors often are explicit entities at an architectural level of the system. Even if connectors are intangible entities in a system implementation, this does not prevent us from implementing a component which provides the services of a specific connector (e.g.: coordination connector, communication connector, persistence connector, load balancing connector) to the other components of the system.

Since some time we have been working in the field of component assembly. Our aim is to analyze and fix dynamic behavioral problems that can arise from component composition. So far, for dynamic properties we concentrated on deadlock analysis while for actual component setting we considered COM/DCOM. In this position paper we briefly sketch our current and future research directions which attempt to contribute to the research in predictable assembly. Our research is characterized by 4 general guidelines:

- extend the analysis of component based systems to general safety and liveness properties;
- provide recovery strategies besides property analysis;
- make the whole approach as automatic as possible;
- validate the approach on several concrete component based architectural models.

We propose an architectural connector-based approach to the assembly problem. This approach is based on the theoretical framework introduced in [8]. The idea there is to build applications by assuming a defined architectural style, namely a modified version of the C2 architectural style [13]. We compose a system in such a way that it is possible to check whether and why the system presents some software anomalies, namely deadlock. At present we have developed a limited portion of the proposed approach [9, 10]. To be more explicit in describing our approach, we can recast what summarized above in the context of the COM/DCOM component based architectural model. We want to 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. Since we are interested in behavioral properties of the assembled system we require this code to be automatically derived in such a way that the functional properties of the composed system are satisfied. At present we assume that there is someone which give us the behavioral specification of the components and we limit ourselves to only one behavioral property of the assembled system namely deadlock freeness. With these two assumptions we are able to develop an automatic tool which derives the assembling code for a set of COM/DCOM components to obtain a deadlock-free system. 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 code. This technique avoids the deadlock by using COM composition mechanisms to insert the synthesized connector within the system while letting the system COM components unmodified. In Section 2 we mention how we intend to deal with other behavioral properties. We also discuss a possible way to avoid the strong assumption about the fact that behavioral specification of the components is someway provided.

The position paper is organized as follows. In Section 2 we introduce the problem in the more general setting of generic behavioral properties. We also address the issue of automatically derive the behavioral specification for a component directly from the component itself. Some notions that are important to understand the paper are also presented. Section 3 presents the technique to allow connectors synthesis [8].

2. PROBLEM DESCRIPTION AND BACKGROUND

Building on our experience as described in Section 1, the problem we want to treat is similar to the predictable assembly problem. In [7], the predictable assembly problem is described as follows: *Given a set of components C, predict property P of an assembly A of these components.* In our setting this problem can be rephrased as follows: *Given a set of components C and a set of properties P automatically derive an assembly A of these components which satisfies every property in P.*

The basic ingredients of this problem are: i) the type of components we refer to, ii) the type of properties we want to check and iii) the type of systems we want to build. We consider COTS components which are truly black-box components. The properties we want to check are functional properties which describe unexpected behaviors of the system. The assembly A depends on the constraints induced by the architectural model the system is based on. The composed systems that we consider are component-based distributed systems. 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 [8]. Here we briefly summarize the main differences between C2 style and CBA style:

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

The precedent items represent the CBA style distinguished characteristics that we do not find in C2. Rationale behind these choices can be found in [8].

2.1 Beyond deadlock

In [9] we only address a specific behavioral property that a composed system must hold to work correctly, namely Deadlock Freeness. In [9] we have explained the approach by providing an example of his application. This example is an instance of the well known *Dining Philosophers* problem [16] in which we consider two philosophers and two forks. We now build on that example in order to discuss other possible behavioral property that a designer might wish to guarantee. We present the component structure of the dining philosophers problem in Figure 1.

There are 4 components:

- the first fork (**Fork1**);

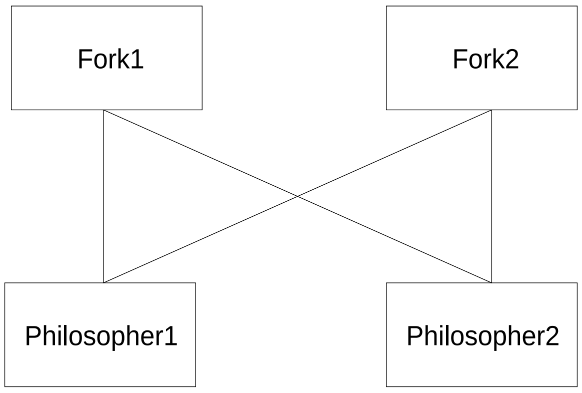

Figure 1: Architectural View of the Dining Philosophers Problem

- the second fork (**Fork2**);
- the first philosopher (**Philosopher1**);
- 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.

Now we present the connector based component structure of the dining philosophers problem in Figure 2. 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. In our approach 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.

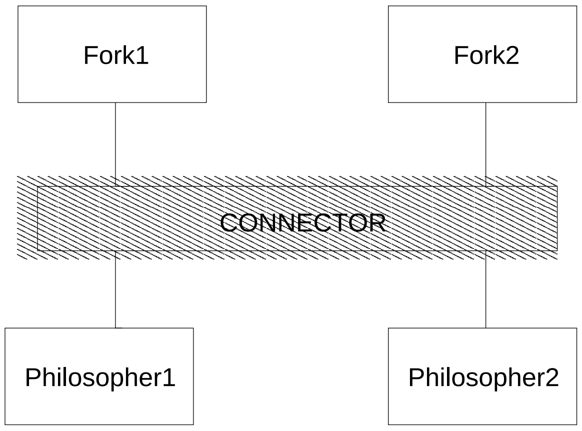

Figure 2: Architectural View of the Connector-based Dining Philosophers Problem

It is worthwhile noticing that before the possible deadlocks are fixed the connector contains all possible composed system behaviors. This means that it 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. At present we are working in this direction in order to extend our approach [9] so that it can be synthesized a connector that implements a particular (deadlock-free) routing policy. This means that we could allow a designer to assign a precise scheduling policy to the connector. The approaches we are experimenting are based on two existent line of research in the program verification area:

- referring to the usual model checking approach [3], we can think of defining the properties which the system must not hold (i.e. unexpected behaviors of the system) by using some kind of temporal logic. In this way we can specify the set P of properties that describe the unexpected behaviors of the system. Then referring to the automata theoretic approach to program verification [3], the idea is to implement a translation function from a property (i.e. temporal logic formula) to an automata, namely a Generalized B$\ddot{u}$chi Automata [3]. By automatically applying this translation function, for any property $p_i \in P$, we can obtain a set of property graphs. For any property graph if it suitably matches with connector graph portions then this means that in those portions the connector does not satisfy the requested routing policy. Therefore, the connector contains a specified unexpected behavior. The recovery policy simply consists of cutting the subgraphs that match with the property graph. In this way we can guarantee the desired behavior;
- other model checking verification approaches can be

derived referring to the supervisory control theory [1]. The theory provides algorithms for the automatic synthesis of supervisory controllers from their specifications. We think that these algorithms can be adapted to the automatic synthesis of software connectors.

2.2 Synthesizing dynamic information off black-box components

In [9] we assume that someone provides the components behavioral specification. This is a strong assumption. In order to treat our version of the predictable assembly problem in a real scale context we must try to avoid it. Black-box inspection techniques could be developed to automatically derive the component behavioral specification from its binary code. In literature there are many techniques to inspect a black-box component [17, 6]. The main problem of these techniques is that they are not completely automatic. Our goal is to device methods to increase the automation of these techniques in our component based setting. Moreover we must extend them in order to derive a behavioral specification that reflects the constraints of the architectural model we refer to. This means that the inspection technique can benefit of the knowledge of the type of components and architectural interaction model we are dealing with. In any case we do not expect to derive a complete specification but only a partial one. However, this partial specification together with the component documentation could be a significant help to derive the complete dynamic behavior specification of the component. It is worthwhile noticing that in many cases component documentation can also be quite expressive taking the form of message sequence charts [15]. In this way an analysis and synthesis tool which derives the components integration code (the connector) directly from components acquired on the market could be derived. Although specifically directed to the discovery of components architectural behavior, this line of research has a large intersection with research in component testing from which it can greatly benefit.

3. CONNECTOR SYNTHESIS

In this section we informally describe how we could extend our current approach towards the definition of a more general synthesis environment. We describe the environment using Figure 3.

As we can see from Figure 3, the tool first inspects the components (intended as Black-box component) in order to derive the behavioral specification code. Then it gets in input the behavioral specification code for all the components of the system. This specification code is expressed by using a language to describe the communication between concurrent processes. We have chosen the CSS (Calculus of Communicating Systems) language because it allows an easy translation from the behavioral specification code to a data structure that takes the form of automata. This automata is the AC-Graph of a component. A formal definition of AC-Graph is in [8]. Informally we can say that an AC-Graph for a component C describes the actual behavior of the component. 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. From the AC graphs the tool 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 specified properties. The AS-Graph is different from the corresponding AC-Graph only in the arcs labels. In fact these labels are symmetric since they model the environment as each component expects it. Given the CBA style, the component environment can only be represented by one or more connectors, thus 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. 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]. At this point the tool builds the property graph for any specified property. Then it verifies, for any property graph, if there are structural mismatches between some portion of connector graph and the considered property graph. Finally the 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 observational equivalence [8].

The following are the steps of the algorithm used to build the connector graph:

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. for each property p_i in the set P of properties to validate, build the PR-Graph PR_i for p_i;
5. if there are some structural mismatches between the PR_i for each property p_i and a portion (or some portions) of the transition graph for K then delete the branches that form these portions of the transition graph of K;
6. for each component C_i if $CBSimulation(AS_i, CB_i)$ does not successfully terminate then $exit(FAILURE)$;
7. $exit(SUCCESS)$;

where:

AS_i is the AS-Graph of the component C_i which is connected to the connector; PR_i is the PR-Graph for the property p_i. We can informally imagine the PR-Graph of p_i as a graph with the same structure of the connector graph and that models the property which represents an unexpected behavior of the system; CB_i is the CB-Graph [8] for C_i.

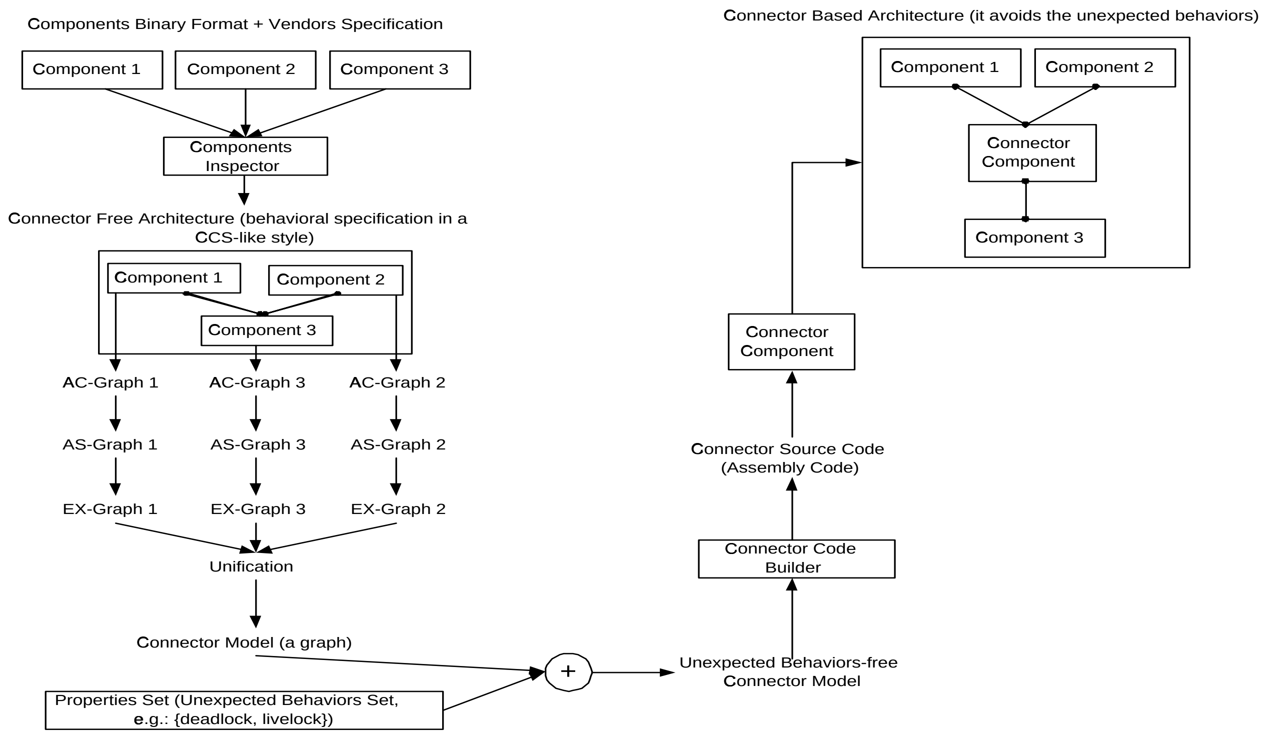
Components Binary Format + Vendors Specification
Component 1
Component 2
Component 3
Components Inspector
Connector Free Architecture (behavioral specification in a CCS-like style)
Component 1
Component 2
Component 3
AC-Graph 1
AC-Graph 3
AC-Graph 2
AS-Graph 1
AS-Graph 3
AS-Graph 2
EX-Graph 1
EX-Graph 3
EX-Graph 2
Unification
Connector Model (a graph)
Properties Set (Unexpected Behaviors Set, e.g.: {deadlock, livelock})
+
Unexpected Behaviors-free Connector Model
Connector Code Builder
Connector Source Code (Assembly Code)
Connector Component
Connector Based Architecture (it avoids the unexpected behaviors)
Component 1
Component 2
Connector Component
Component 3

Figure 3: Automatic Synthesis Tool

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 [8] from the portion of the connector behavior regarding the communication with a given component (C_i). Informally *CB-Simulation* is a notion of simulation based on observational equivalence [14]. It is needed to perform step 6 of the algorithm because independently of the type of the unexpected behavior that we want to avoid, the connector component must ensure the expected behavior for all components connected to it. In this way we are guaranteed that the system will not exhibit unexpected behaviors.

At this point we have obtained the connector graph that models the behavior of the composed system. Its behavior avoids all the unexpected behaviors specified by the set P of behavioral properties. Thus the tool ends its activity automatically deriving the code of the services provided by the connector component. Obviously these services implement the requests routing policy in order to avoid every unexpected behavior specified in P. Informally in this step the tool, by visiting the connector graph, should automatically derive when and how the connector delegates requests from a connected component to another one. These request delegations are conditioned in order to implement the correct routing policy.

4. CONCLUSION

In this position paper we have briefly described our experience in treating a restricted instance of the predictable assembly problem. We have described an architectural approach to connector synthesis for a deadlock-free component based architecture which we validated in the concrete setting of COM/DCOM. Based on that experience, we propose how to extend the analysis of component based systems to general safety and liveness properties. Our approach is oriented to recovery strategies besides property analysis in such a way that the whole approach can be as automatic as possible. However, the scenario we have depicted depends on a number of results and problems solutions that we have not mentioned. We have not specified the class of safety and liveness property we intend to deal with. This obviously is a preliminary step since it will not be possible to successfully deal with any dynamic property. We have not analyzed the relationship between property and interaction architecture. We simply stuck to the CBA one. We have not considered real component based contexts, which can greatly influence/constrain many of the above mentioned choices. All these dimensions have to be properly taken into consideration in order to propose feasible instances of our synthesis environment.

5. REFERENCES

[1] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

[2] B. Boehm and C. Abts. Cots integration: Plug and pray? *IEEE Computer*, 32(1), Jan. 1999.

[3] J. K. D. Giannakopoulou 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.

[4] R. V. Dr. Mark and J. Dean. An architectural approach to building systems from cots software components. *National Research Council Report Number 40221.*

[5] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), Nov. 1995.

[6] G. C. Hunt and M. L. Scott. Intercepting and instrumenting com applications. *5th USENIX Conference on Object-Oriented Technologies and Systems*, May 1999.

[7] J. S. I. Crnkovic, H. Schmidt and K. Wallnau. Anatomy of a research project in predictable assembly. *Fifth ICSE Workshop on Component-Based Software Engineering White paper.*

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

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

[13] 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.

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

[15] C. Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow, England, 1998.

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

[17] J. Voas. An application-specific approach for assessing the impact of cots components. *Reliable Software Technologies.*