The role of architecture in components assembly

Paola Inverardi and Massimo Tivoli
University of L'Aquila
(Dip. Informatica)
via Vetoio 1, 67100 L'Aquila, Italy
{inverard, tivoli}@univaq.it

Abstract

One of the main problem in component assembly is related to the ability to establish properties on the assembly code by only assuming a relative 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. In this position paper, by means of an explanatory example we illustrate our approach and discuss on possible recovery strategies.

1 Introduction

One of the main problem in component assembly is related to the ability to establish properties on the assembly code by only assuming a relative 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 component based concurrent systems, COTS components integration may cause deadlocks or other software anomalies within the system [12, 1, 7, 8]. The use of COTS software components in system construction presents new challenges to system architects and designers [9]. 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. Component assembling can result in architectural mismatches when trying to integrate components with incompatible interaction behavior [2]. 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.

In this position paper, by means of an explanatory example we illustrate our approach to the assembly problem. Our aim is to analyze and fix dynamic behavioral problems that can arise from component composition. We propose an architectural connector-based approach [4, 5, 6]. The idea is to build applications by assuming a defined architectural style, namely a modified version of the C2 architectural style [10]. 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 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. We assume that some description of the behavioral specification of the components is available and that a precise definition of the properties to satisfy exist. With these two assumptions we are able to develop an automatic tool which derives the assembling code for a set of components to obtain a properties-satisfying system. In previous works we have assumed to have explicit formal specifications of the components behavior and we limited ourselves to only one behavioral property of the assembled system namely deadlock freeness. In this position paper we go a step forward and we show how our technique and tool can actually be used to deal with components whose behavior is only partially specified and with properties of the assembly code others than deadlock.

The position paper is organized as follows. In Section 2 we introduce the problem we want to address and some background. Some notions that are important to understand the paper are also presented. Section 2.1 presents the technique to allow connectors synthesis [4] which is then applied in Section 2.2 on an example. Section 3 concludes.

2 Problem description and Background

Building on our experience as described in Section 1, the problem we want to treat can be phrased 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.*

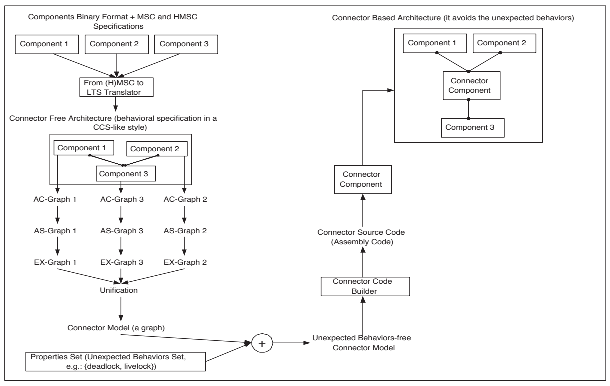
The basic ingredients of this problem are: i) the type of components we refer to, ii) the type of properties we want to check and iii) the type of systems we want to build. We consider COTS components which are truly black-box components. The properties we want to check are functional properties which 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 [4].

It is important to notice that, besides assuming that the system architecture must reflect the rules of a well defined architectural style (namely CBA style), we also assume that some kind of COTS components behavioral specification is provided (e.g.: through message sequence chart (MSC) and high level MSC (HMSC) specifications [14, 13, 15] of the system components). Thus when we say: *Given a set of components C* in the problem definition we mean that we consider a set of component behavioral specifications C. 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 the following section we detail the approach in order to provide enough background to follow the explanatory example.

2.1 Connector Synthesis

Our goal is to develop a tool that performs an automatic software connector synthesis to derive directly from the COTS components specification (MSCs and HMSC) the assembly code to build the composed system. The composed system automatically synthesized must satisfy the behavioral properties expected from the system designers and architects. In our synthesis approach 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 [14, 13, 15] exist. We can adapt these algorithms to build the actual behavior graph (AC-Graph) [4, 5, 6] of the system components directly from the MSC specifications. These specifications are then used to the automatic synthesis of software connectors. A formal definition of AC-Graph is in [4]. 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. These graphs describe the interaction behavior of each component with the external environment. We wish to derive from a component behavior the requirements on its environment that guarantee specified properties. 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. In fact these labels are symmetric since they model the environment as each component expects it. Given the CBA style [4], 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. An action with the symbol (?) is an action on an unknown communication channel for the component associated with the EX-Graph. An action with the symbol (l) is an action on the communication channel that links the component identified by the label l 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 [4]. 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. 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 properties and recovery. So far we have mainly concentrated on deadlock analysis. In this case the deadlocks analysis step consists of searching for stop nodes in the connector behavioral graph. The deadlocks recovery step consists of cutting the connector graph branches that lead to stop nodes. 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 observational equivalence [4]. It is worthwhile noticing once all the possible deadlocks are fixed the connector still contains all possible composed system deadlock-free 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.

At present we are working to extend our approach [5] 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. 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. 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. At this point the tool could apply two possible recovery strategies to guarantee the desired behavior: i) It does not modify the (deadlock-free) connector semantics or ii) it modifies the (deadlock-free) 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, after that 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 by the system designers and architects. In Figure 1 we describe the architecture of the automatic connector synthesis tool.



Components Binary Format + MSC and HMSC Specifications
Component 1
Component 2
Component 3
From (H)MSC to LTS Translator
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 1: Automatic Synthesis Tool

It is important to notice that every LTS corresponding to a component is expressed by using a language to describe the communication between concurrent processes. We have chosen the CCS [11] (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 (the AC-Graph).

2.2 Directory Service Problem

Now we present an application of the synthesis to a COTS [1, 9, 16] components based version of an example presented in [12]. This example describes the *Directory Service* problem. It considers a system that provides a basic directory service as part of a simple file system. The directory service supports callbacks to notify clients of changes in the managed directory. Such a notification service is useful, for instance, to keep a visual representation of the directory up to date. Using a *Microsoft IDL* notation we can represent the interface of the directory service component as follows:

```
interface IDirectoryService : IDispatch {
    [id(1), helpstring("method ThisFile")]
    HRESULT ThisFile([in] char *fileName, [out, retval] HANDLE *fileHandle);
    [id(2), helpstring("method AddEntry")]
    HRESULT AddEntry([in] char *fileName, [in] HANDLE fileHandle);
    [id(3), helpstring("method RemoveEntry")]
    HRESULT RemoveEntry([in] char *fileName);
    [id(4), helpstring("method RegisterNotifier")]
    HRESULT RegisterNotifier([in] LP_CALLBACK_FUNCTION pNotifier);
    [id(5), helpstring("method UnregisterNotifier")]
    HRESULT UnregisterNotifier([in] LP_CALLBACK_FUNCTION pNotifier);
};
```

The first three methods support file lookup, file addition and file removal respectively. The last two support registration and unregistration of callbacks respectively. Registered callbacks are invoked on addition or removal of a file. The directory service system is composed by the following components: i) a client that invokes the addition or removal of a file, ii) a directory service component that exports the above interface, iii) a directory display component that, by using a callback function registered on the directory service component to notify the changes of the directory content, provides a visual representation of the directory up to date.

Now we suppose that the vendor of the above COTS components provides the MSCs and the HMSC for the directory service system. Thus we can benefit of the specifications described in Figure 2 and in Figure 3.

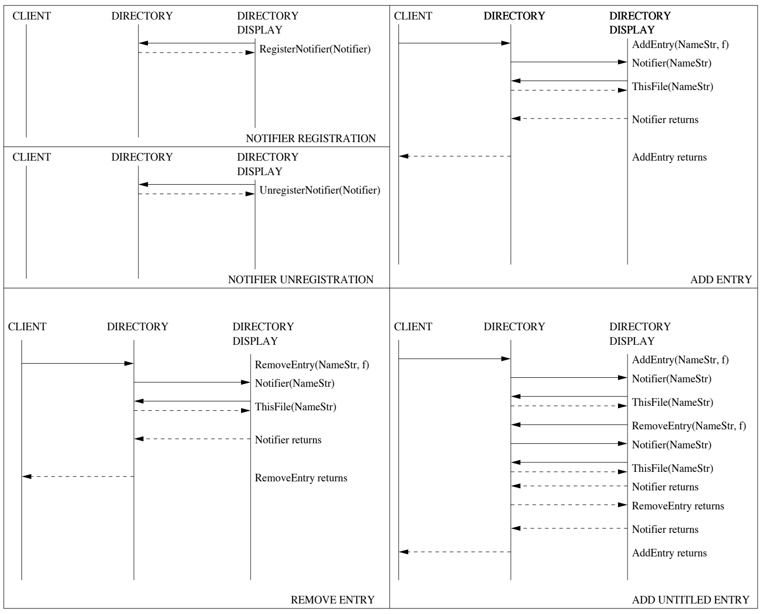


Figure 2: Basic MSCs for the Directory Service System

The directory service system consists of five phases:

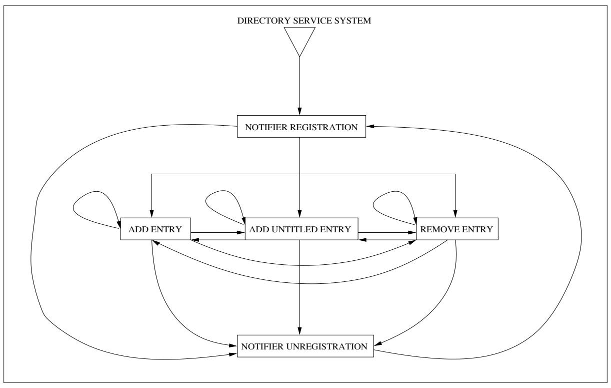


Figure 3: High level MSC for the Directory Service System

- **NOTIFIER REGISTRATION**: the directory display component invokes the method $RegisterNotifier$ of the directory service component to register a callback for the addition or removal of a file;
- **NOTIFIER UNREGISTRATION**: the directory display component invokes the method $UnregisterNotifier$ of the directory service component to unregister a callback for the addition or removal of a file;
- **ADD ENTRY**: the client invokes the method *AddEntry* of the directory service component to add a named file in the directory. This call will enable the callback function;
- **REMOVE ENTRY**: the client invokes the method *RemoveEntry* of the directory service component to remove a named file in the directory. This call will enable the callback function;
- **ADD UNTITLED ENTRY**: the client invokes the method *AddEntry* of the directory service component to add an unnamed file in the directory. This call will enable the the callback function. Since the users would be confused if the directory ever contained an entry for an unnamed file, in this case the callback function is implemented to remove the unnamed entry as soon as the notification call arrives.

As appears in Figure 3, the directory service system combines these five phases first performing the **NOTIFIER REGISTRATION** phase, then performing either of the three phases: **ADD ENTRY**, **REMOVE ENTRY** or **ADD UNTITLED ENTRY** and finally performing the **NOTIFIER UNREGISTRATION** phase. From the MSC and the HMSC specifications we can automatically derive the AC-Graphs for the components of the directory service system. In Figure 4 we report these AC-Graphs.

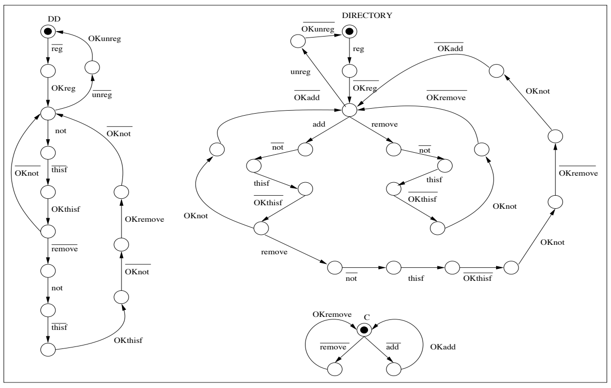


Figure 4: AC-Graphs for the components of the Directory Service System

The components C, DD, and $DIRECTORY$ represent the client, the directory display and the directory service component respectively. The actions $thisf$, *add*, *remove*, *reg* and *unreg* correspond to the $ThisFile$, $AddEntry$, $RemoveEntry$, $RegisterNotifier$ and $UnregisterNotifier$ method respectively. The actions $OKthisf$, $OKadd$, $OKremove$, $OKreg$ and $OKunreg$ correspond to the usual returns for the above methods respectively. The action *not* corresponds to the callback function activation and the action $OKnot$ corresponds to the callback function return. From the AC-Graphs the synthesis tool derives the corresponding AS-Graphs and from these it derives the corresponding EX-Graphs. In Figure 5 we represent the AS-Graphs for the components of the directory service system.

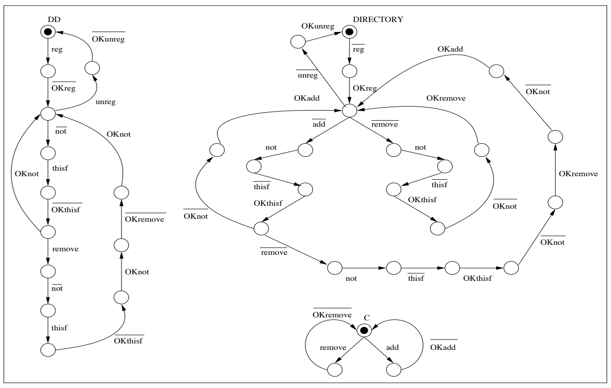


Figure 5: AS-Graphs for the components of the Directory Service System

In Figure 6 and in Figure 7 we represent the EX-Graphs for the components of the directory service.

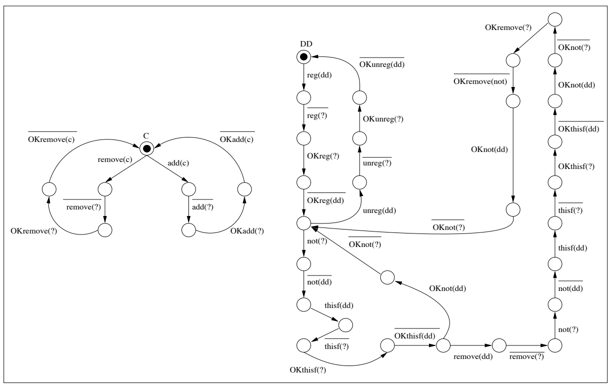


Figure 6: EX-Graphs for the Client and Directory Display component

The communication channels c, dd and d are the channels that link the C, DD and $DIRECTORY$ component to the connector respectively. We can apply the unification algorithm to build the behavioral connector graph. In Figure 8 we represent the graph automatically synthesized for the directory service system.

At this point we have obtained the connector graph that models the behavior of the composed system. After the tool has fixed all possible deadlocks and has verified if the deadlock-free connector ensures the expected behavior of all components connected to it, we can assign to the deadlock-free connector a precise scheduling policy in order to satisfy the wanted behavior. For instance we might want the *AddEntry* method of the directory service component to satisfy the following pre and post-conditions:

```
[id(2), helpstring("method AddEntry")]
HRESULT AddEntry([in] char *fileName, [in] HANDLE fileHandle);
// precondition:    ((fileName != "") and (fileHandle != NULL))
// postcondition:   (ThisFile(fileName) == fileHandle)
```

but, referring to [12], the implementation of the *AddEntry* method with the callback function implementation breaks the above contract. Looking at the MSC of Figure 2, if

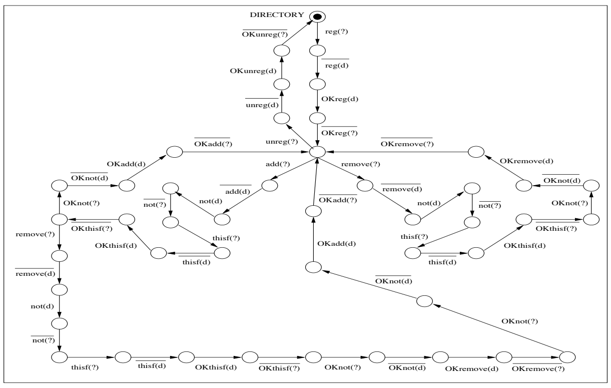


Figure 7: EX-Graph for the Directory Service component

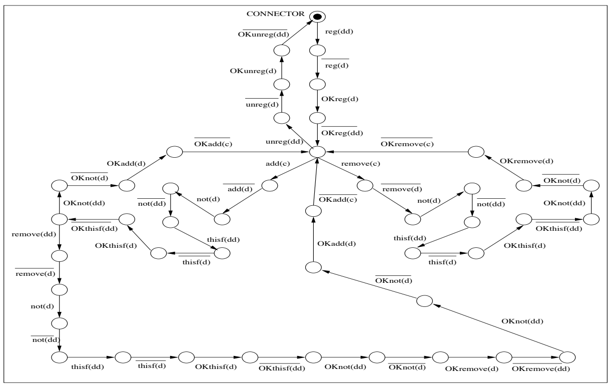


Figure 8: Behavioral graph for the connector of the Directory Service System

$AddEntry(''Untitled'', someFile)$ is called, the newly added entry is immediately removed by the callback function. Thus, the post-condition $ThisFile(''Untitled'') == someFile$ does not hold. In order to fix this problem we can think of operating on the connector in order to satisfy the *AddEntry* post-condition without touching the *AddEntry* method implementation. We can operate as follows. First, we define property $P1$ that represents the unexpected behavior by using a classic linear temporal logic: $P1 = \Box\ ((add(c) \Longrightarrow \Diamond\ remove(dd)) \Longrightarrow \Diamond\ \overline{OKadd(c)})$ where the symbol $\Box$ is the *henceforth* operator and the symbol $\Diamond$ is the *eventually* operator. Then we model check this property on the connector graph and we discover that it matches with a portion of the connector graph. At this point we can think of simply removing the unwanted behavior thus cutting away the connector graph portion. Thus we decide to change the semantics of the glue code (the connector)in order to guarantee the desired behavior. Figure 9 reports the connector behavior reduced in order to avoid the unexpected behavior defined by the property $P1$.

Property $P1$ is defined in such a way that it represents all the connector graph portions in which there is a request of *RemoveEntry* within a request of *AddEntry*. It means that the property identifies all the possible scenarios that do not satisfy the *AddEntry* post-condition. The synthesis tool can now perform the derivation code step for the connector component in such a way that the composed system satisfies the wanted behavior.

As discussed in Section 2.1 a designer could also have adopted a different policy which does not change the connector semantics but simply synthesize the connector code in such a way that the unexpected behaviors are treated as exceptional running traces.

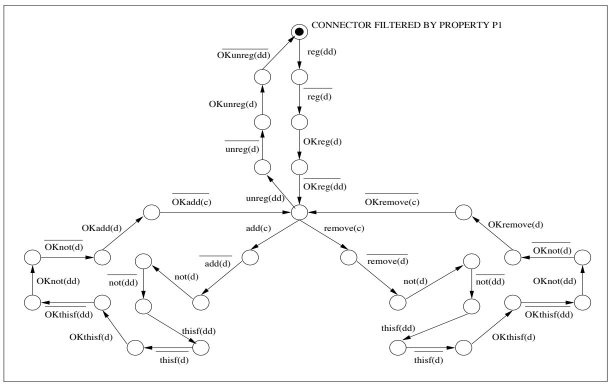

CONNECTOR FILTERED BY PROPERTY P1
$\overline{\text{OKunreg(dd)}}$
reg(dd)
OKunreg(d)
$\overline{\text{reg(d)}}$
$\overline{\text{unreg(d)}}$
OKreg(d)
unreg(dd)
$\overline{\text{OKreg(dd)}}$
$\overline{\text{OKadd(c)}}$
$\overline{\text{OKremove(c)}}$
OKadd(d)
add(c)
remove(c)
OKremove(d)
$\overline{\text{OKnot(d)}}$
$\overline{\text{add(d)}}$
$\overline{\text{remove(d)}}$
$\overline{\text{OKnot(d)}}$
OKnot(dd)
$\overline{\text{not(dd)}}$
not(d)
not(d)
$\overline{\text{not(dd)}}$
OKnot(dd)
OKthisf(dd)
thisf(dd)
thisf(dd)
OKthisf(dd)
OKthisf(d)
$\overline{\text{thisf(d)}}$
$\overline{\text{thisf(d)}}$
OKthisf(d)

Figure 9: Connector graph of the Directory Service System filtered by property P1

3 Conclusion

In this position paper we have described a connector-based architectural approach to component assembly. Our approach focusses on detection and recovery of the assembly unexpected behaviors. A key role is played by the software architecture structure since it allows all the interactions among components to be explicitly routed through the connector. We have applied our approach to an example known in the literature and we have discussed its implications on the actual nature of COTS components and on the designer choices at the connector synthesis level. As far as components are concerned we only assumed to have a description of the components behavior by means of MSCs, which is, in our view, an acceptable hypothesis. For the connector synthesis, we have shown in the example that different policies can be adopted in order to guarantee the wanted properties. In one case we can actually think of changing the connector semantics in order to prevent unwanted behaviors, in the other we can be more conservative but still implement a recovery strategy.

References

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

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

[3] 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.

[4] P. Inverardi and S. Scriboni. Connectors syntesis for deadlock-free component based architectures. *16th ASE, Coronado Island, California*, November 2001.

[5] 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.

[6] 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.

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

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

[9] D. Mark, R. Vigder, and J. Dean. An architectural approach to building systems from cots software components. *National Research Council Report Number 40221.*

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

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