Software Architecture for Correct Components Assembly

Paola Inverardi and Massimo Tivoli

University of L'Aquila
Dip. Informatica
fax: +390862433057
via Vetoio 1, 67100 L'Aquila
{inverard, tivoli}@di.univaq.it

Abstract. Correct automatic assembly in software components is an important issue in CBSE (Commercial-Off-The-Shelf). Building a system from reusable software components or from COTS (*Commercial-Off-The-Shelf*) components introduces a set of problems. One of the main problems in components assembly is related to the ability to properly manage the dynamic interactions of the components. Component assembling can result in architectural mismatches when trying to integrate components with incompatible interaction behavior like deadlock and other software anomalies. This problem represents a new challenge for system developers. The issue is not only in specifying and analyzing a set of properties rather in being able to enforce them out of a set of already implemented (local) behaviors. 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. Starting from the specification of the system to be assembled and of its properties we develop a framework which automatically derives the glue code for the set of components in order to obtain a properties-satisfying system (i.e. the failure-free version of the system).

1 Introduction

Nowadays there is the need to built high quality software systems in short time. This moves developers toward reuse-based development methodologies. CBSE (*Component Based Software Engineering*) is a process focussed on the software systems design and developing by assembling reusable software components. Clemens (1995) describes the CBSE process as follows: *the CBSE is changing the methods to develop huge-size software systems. It adopts the philosophy "Buy! No Build!" followed by Fred Brooks et al. The CBSE moves the attention of the software developers to the software systems assembly (i.e. component assembly). The implementation has been replaced from the integration.*

Thus in CBSE the integration is the real challenge. Building a system from a set of COTS(*Commercial-Off-The-Shelf*) [26] components introduces a set of

M. Bernardo and P. Inverardi (Eds.): SFM 2003, LNCS 2804, pp. 92–121, 2003.
© Springer-Verlag Berlin Heidelberg 2003

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 [9] when trying to integrate components with incompatible interaction behavior like deadlock and other software anomalies. Thus if we want to assure that a component based system validates specified dynamic 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 [2, 10].

Our approach to the assembly problem is to compose systems by assuming a well defined architectural style [15, 14] in such a way that it is possible to detect and to fix software anomalies. Moreover we assume that a specification of the desired assembled system is available and that a precise definition of the properties to satisfy exists. With these assumptions we are able to develop a framework that automatically derives the assembly code for a set of components so that, if possible, a properties-satisfying system is obtained (i.e. the failure-free version of the system). The assembly code implements an explicit software connector which mediates all interactions among the system components as a new component to be inserted in the composed system. The connector can then be analyzed and modified in such a way that the behavioral (i.e. functional) properties of the composed system are satisfied. Depending on the kind of property, the analysis of 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. We assume that components behavior is only partially and indirectly specified by using bMSC (*basic Message Sequence Charts*) and HMSC (*High level MSC*) specifications [1] of the desired assembled system and we address behavioral properties of the assembly code together with different recovery strategies. The behavioral properties we deal with are the deadlock freeness property [15, 14] and generic coordination policies of the components interaction behavior [16].

The paper is organized as follows. Section 2 introduces background notions and theoretical foundations in order to understand our approach. Section 3 formalizes the method concerning the behavioral failures-free connectors synthesis which is also applied to an explanatory example. Section 4 describes a realistic application example of our approach. Section 5 presents related works and Section 6 discusses future work and concludes.

2 Background

In this section we provide the background needed to understand the approach formalized in Section 3.

2.1 The Reference Architectural Style

The architectural style we use, called *Connector Based Architecture* (CBA), is derived from C2 architectural style [22] and consists of components and connectors which define a notion of top and bottom. The top (bottom) of a component may be connected to the bottom (top) of a single connector. There is no bound on the number of components that may be attached to a single connector. Components can only communicate via connectors. It is disallowed the direct connection between connectors. Each component is connected to the connector through a synchronous communication channel. Components communicate synchronously by passing two type of messages: notifications and requests. A notification is sent downward, while a request is sent upward. Requests are service or data demands, while notifications are reply to requests, and they announce state changes or return data. Both components and connectors have a top-domain and a bottom-domain. A top-domain of a component or of a connector is the set of requests sent upward and of received notifications. A bottom-domain of a component or of a connector is the set of received requests received and of notifications sent downward. Connectors are responsible for the routing of messages and they exhibit a strictly sequential input-output behavior[1]. CBA is a generic layered style. Since it is always possible to decompose a n-layered CBA system in n single-layered CBA systems (see the right side of Figure 1), in the following of this paper we will only deal with single layered systems. This decomposition is done by considering for each intermediate component (i.e. a component of an intermediate layer) two behavioral views: i) the component's behavior with respect to the messages exchanged on the component's top-domain and ii) the component's behavior with respect to the messages exchanged on the component's bottom-domain [15]. This decomposition is possible because each layer of a multi-layered CBA system is independent from the other ones. Thus to cope with multi-layered systems we apply the formalized approach for each single-layered sub-system.

In Figure 1 we show an instance of the CBA style made of two components and one connector.

2.2 CCS

For our purpose we need to summarize the most relevant definitions regarding CCS (*Calculus of Communicating Systems*), we refer to [23] for more details.

[1] Each input action is strictly followed by the corresponding output action.

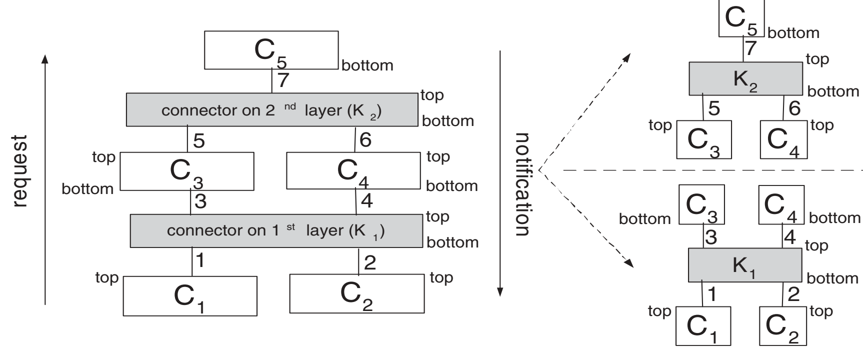

Fig. 1. An instance of the 2-layered CBA style and its decomposition

The CCS syntax is the following:

$$p ::= \mu.p \mid nil \mid p + p \mid p|p \mid p\backslash A \mid x \mid p[f]$$

Terms generated by p ($Terms$) are called *process terms* (called also *processes* or *terms*); x ranges over a set $\{X, Y, ..\}$, of process variables. A process variable is defined by a process definition $x \stackrel{def}{=} p$, (p is called the expansion of x). As usual, there is a finite set of visible actions $Vis = \{a, \overline{a}, b, \overline{b}, ...\}$ over which α ranges, while μ, ν range over $Act = Vis \cup \{\tau\}$, where τ denotes the so-called *internal action.* We denote by $\overline{\alpha}$ the action complement: if $\alpha = a$, then $\overline{\alpha} = \overline{a}$, while if $\alpha = \overline{a}$, then $\overline{\alpha} = a$. By nil we denote the empty process. The operators to build process terms are prefixing ($\mu.p$), summation ($p + p$), parallel composition ($p|p$), restriction ($p\backslash A$) and relabelling ($p[f]$), where $A \subseteq Vis$ and $f : Vis \rightarrow Vis$.

An operational semantics OP is a set of inference rules defining a relation $D \subseteq Terms \times Act \times Terms$. The relation is the least relation satisfying the rules. If $(p, \mu, q) \in D$, we write $p \stackrel{\mu}{\rightarrow}_{OP} q$. The rules defining the semantics of CCS [23], from now on referred to as SOS, are here recalled:

$$Act \quad \frac{}{\alpha.P \stackrel{\alpha}{\rightarrow} P} \qquad Synch \quad \frac{P \stackrel{\alpha}{\rightarrow} P', Q \stackrel{\overline{\alpha}}{\rightarrow} Q'}{P|Q \stackrel{\tau}{\rightarrow} P'|Q'}$$

$$Sum \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P+Q \stackrel{\alpha}{\rightarrow} P'} \qquad Rel \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P[f] \stackrel{f(\alpha)}{\rightarrow} P'[f]}$$

$$Comp \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P|Q \stackrel{\alpha}{\rightarrow} P'|Q} \qquad Res \quad \frac{P \stackrel{\alpha}{\rightarrow} P', \alpha \notin L \cup \overline{L}}{P\backslash L \stackrel{\alpha}{\rightarrow} P'\backslash L}$$

$$Con \quad \frac{P \stackrel{\alpha}{\rightarrow} P', A \stackrel{def}{=} P}{A \stackrel{\alpha}{\rightarrow} P'}$$

The rules Sum and $Comp$ have a symmetric version which is omitted.

A *labelled transition system (LTS)* (or simply *transition system*) TS is a quadruple (S, T, D, s_0), where S is a set of states, T is a set of transition labels, $s_0 \in S$ is the initial state, and $D \subseteq S \times T \times S$. A transition system is finite if D is finite.

A finite computation of a transition system is a sequence $\mu_1\mu_2..\mu_n$ of labels such that:
$s_0 \xrightarrow{\mu_1}_{OP} .. \xrightarrow{\mu_n}_{OP} s_n$.

Given a term p (and a set of process variable definitions), and an operational semantics OP, $OP(p)$ is the transition system $(Terms, Act, D, p)$, where D is the relation defined by OP. For example, $SOS(p)$ is the transition system defined by the SOS semantics for the term p. CCS can be used to define a wide class of systems, that ranges from Turing machines to finite systems [27]; therefore, in general, CCS terms cannot be represented as finite state systems. For our purposes we will in the following assume that all the systems we will deal with are finite state. In general a correspondence between CCS terms and LTSs can be always defined. A CCS term may be encoded in LTS as follows:

- LTS states are CCS terms;
- transitions given by $\rightarrow_{OP}$, i. e. by operational semantics;
- the LTS start state is the one corresponding to the encoded CCS term;

and any finite-state LTS can be encoded in CCS as follows:

- associate a process S_i to each LTS state s_i;
- in declaration of S_i, sum (summation operator $+$) together terms of form $\alpha.S_j$ for each transition $s_i \xrightarrow{\alpha} s_j$ in LTS;
- the CCS term is the one corresponding to the encoded LTS start state.

2.3 Configuration Formalization

In order to describe components and system behaviors we use the CCS [23] notation. For the purpose of this paper this is a fair assumption. Actually our framework allows to automatically derive these CCS descriptions from the HMSC and bMSC specifications [1] of the system [29, 25]. These kinds of specifications are common in practice thus CCS can merely be regarded as an internal to the framework specification language. In Section 4 we show a complete treatment of a case study starting from a HMSC and bMSC specification of the system. Since HMSC and bMSC specifications model finite-state behaviors of a system we will use finite-state CCS. To our purposes we need to formalize two different ways to compose a system. The first one is called *Connector Free Architecture* (CFA) and is defined as *a set of components directly connected in a synchronous way* (i.e. without a connector). The second one is called *Connector Based Architecture* (CBA) and is defined as *a set of components directly connected in a synchronous way to one or more connectors*:

Definition 1 (Connector Free Architecture (CFA)).
$CFA \equiv (C_1 \mid C_2 \mid \ldots \mid C_n)\backslash \bigcup_{i=1}^{n} Act_i$ *where for all* $i = 1,..,n$, Act_i *is the actions set of the CCS process* C_i.

Definition 2 (Connector Based Architecture (CBA)).
$CBA \equiv (C_1[f_1] \mid C_2[f_2] \mid \ldots \mid C_n[f_n] \mid K)\backslash \bigcup_{i=1}^{n} Act_i[f_i]$ *where for all* $i = 1,..,n$, Act_i *is the actions set of the CCS process* C_i *and* f_i *is a relabelling functions such that* $f_i(\alpha) = \alpha_i$ *for all* $\alpha \in Act_i$ *and* K *is the CSS process representing the connector.*

α in a CFA denotes a visible action (i.e. an input action or an output action). α_i in the corresponding CBA denotes the same visible action performed on the communication channel i. The channel i connects the component has performed α_i with the connector. In Figure 2 we show an example of a CFA system and of the corresponding CBA system. The double circled states represent initial states.

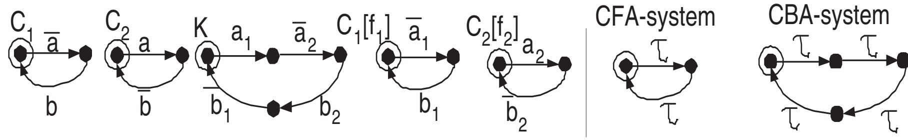

Fig. 2. CFA and corresponding CBA

In Figure 3, we show a graphic representation of the CFA and CBA systems showed in Figure 2.

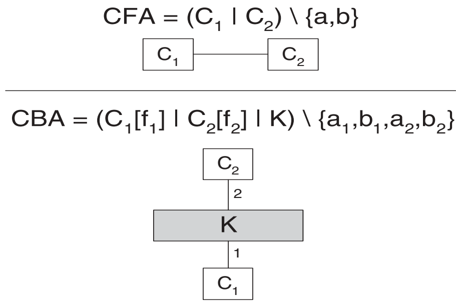

Fig. 3. A graphic representation of the CFA and CBA systems of Figure 2

3 Approach Description

The problem we want to treat can be informally phrased as follows: *given a CFA system T for a set of black-box interacting components and a set of coordination properties P automatically derive the corresponding CBA system V which satisfies every property in P.*

We are assuming that a specification of the system to be assembled is provided in terms of bMSCs and HMSCs. Referring to Definition 1, we also assume that for each component a description of its behavior as finite-state CCS term (i.e. LTS *Labelled Transitions System*) has been automatically derived from the bMSCs and HMSCs specification [29, 25]. Moreover we assume that a specification of the coordination properties to be checked exists. In the following, by means of a working example, we discuss our method proceeding in three steps as illustrated in Figure 4.

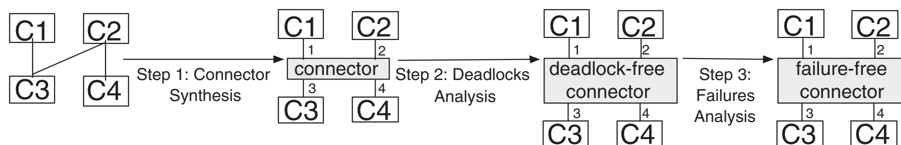

Fig. 4. 3 step method

The first step builds a connector following the CBA style constraints. The second step performs the deadlocks detection and recovery process. Finally, the third step performs the check of the specified coordination properties against the deadlock-free connector model and then synthesizes a properties-satisfying connector model.

Note that although in principle we could carry on the second and third step together, for the sake of clarity we decided to keep them separate. Actually, the current framework implementation follows this schema.

3.1 First Step: Connector Synthesis

The first step of our method (see Figure 4) starts with a CFA system and produces the equivalent CBA system. It is worthwhile noticing that this can always be done [15]. As we will see in Section 3.2 the two systems behave equivalently, under a suitable notion of equivalence. We proceed as follows:

i) By taking into account the bMSCs and the HMSCs specification of the composed system and the CFA architectural style, we first derive the CCS specification for each component and then for each finite-state CCS component specification (in the CFA system) we derive the corresponding AC-Graph. AC-Graphs

model components behavior in terms of interactions with the external environment. The external environment for a component C_i is represented by the parallel composition of the components $C_j, j \neq i$. 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. AC-Graph carry on information on both labels and states:

Definition 3 (AC-Graph). *Let $\langle S_i, L_i, \rightarrow_i, s_i \rangle$ be a labelled transition system of a component C_i. The corresponding Actual Behavior (AC) Graph AC_i is a tuple of the form*
$\langle N_{AC_i}, LN_{AC_i}, A_{AC_i}, LA_{AC_i}, s_i \rangle$ where $N_{AC_i} = S_i$ is a set of nodes, LN_{AC_i} is a set of state labels, LA_{AC_i} is a set of arc labels with τ ($LA_{AC_i} = L_i \bigcup \tau$), $A_{AC_i} \subseteq N_{AC_i} \times LA_{AC_i} \times N_{AC_i}$ is a set of arcs and s_i is the root node.

- *We shall write $g \xrightarrow{l} h$, if there is an arc (g, l, h) $\in A_{AC_i}$. We shall also write $g \rightarrow h$ meaning that $g \xrightarrow{l} h$ for some $l \in LA_{AC_i}$.*
- *If $t = l_1 \cdots l_n \in LA^*_{AC_i}$, then we write $g \overset{t}{\longrightarrow}^* h$, if $g \xrightarrow{l_1} \cdots \xrightarrow{l_n} h$. We shall also write $g \longrightarrow^* h$, meaning that $g \overset{t}{\longrightarrow}^* h$ for some $t \in LA^*_{AC_i}$.*
- *We shall write $g \overset{l}{\Rightarrow} h$, if $g \overset{t}{\longrightarrow}^* h$ for some $t \in \tau^*.l.\tau^*$.*

In Figure 5 we show the AC-Graphs of the CFA system of our working example.

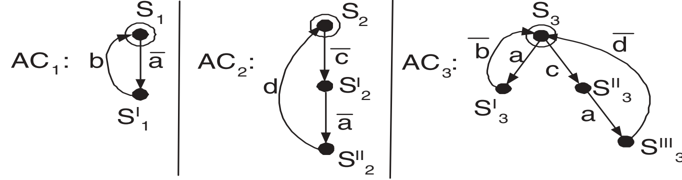

Fig. 5. AC-Graphs of the example

ii) By taking into account the deadlock freeness property and the CFA architectural style, we derive from AC-Graph the requirements on its environment that guarantee deadlock freedom. Referring to Definition 1, we recall that the environment of a component C_i is represented by the set of components C_j $(j \neq i)$ in parallel. A component will not block if its environment can always provide the actions it requires for changing state. This is represented as AS-Graphs (Figure 6):

Definition 4 (AS-Graph). *Let* $(N_{AC_i}, LN_{AC_i}, A_{AC_i}, LA_{AC_i}, s_i)$ *be the AC-Graph* AC_i *of a component* C_i*, then the corresponding ASsumption (AS) Graph* AS_i *is* $(N_{AS_i}, LN_{AS_i}, A_{AS_i}, LA_{AS_i}, s_i)$ *where* $N_{AS_i} = N_{AC_i}$, $LN_{AS_i} = LN_{AC_i}$, $LA_{AS_i} = LA_{AC_i}$ *and*
$A_{AS} = \{(\nu,\overline{a},\nu') \mid (\nu,a,\nu') \in A_{AC}\} \bigcup \{(\nu,b,\nu') \mid (\nu,\overline{b},\nu') \in A_{AC}\}$.

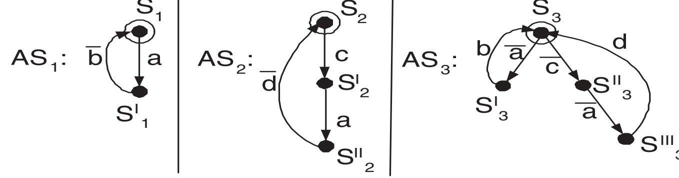

Fig. 6. AS-Graphs of the example

Analogously to AC-Graphs we have one graph for each component. The only difference from AC-graphs is in the arcs labels, which are symmetric since they model the environment as each component expects it.

iii) Now if we consider Definition 2 (i.e. by taking into account CBA architectural style), the environment of a component can only be represented by connectors. Thus we can refine the definition of AS-Graph in a new graph, the EX-Graph, that represents the behavior that the component expects from the connector (Figure 7). We know that the connector performs strictly sequential input-output operations only, thus if it receives an input from a component it will then output the received input message to the destination component. Analogously, if the connector outputs a message, this means that immediately before it inputs that message. Intuitively, for each transition labelled with a visible action α ($\overline{\alpha}$) in the AS graph, in the corresponding EX graph there are two strictly sequential transition labelled α_i and $\overline{\alpha_?}$ ($\alpha_?$ and $\overline{\alpha_i}$), respectively. Let C_i the component for which we are deriving from the AS-Graph the corresponding EX-Graph; referring to CBA in Section 2.3, action α_i ($\overline{\alpha_i}$) denotes an input (output) action α towards the connector on the communication channel that connects C_i to the connector (i. e. the communication channel i). Action $\alpha_?$ ($\overline{\alpha_?}$) denote an input (output) action α towards the connector on a communication channel that connects the connector to a component different than C_i; thus this communication channel is unknown for C_i (we denotes this unknown channel by using the question mark):

Definition 5 (EX-Graph). *Let* $(N_{AS_i}, LN_{AS_i}, A_{AS_i}, LA_{AS_i}, s_i)$ *be the AS-Graph* AS_i *of a component* C_i*; we define the connector EXpected (EX) Graph* EX_i *from the component* C_i *the graph* $(N_{EX_i}, LN_{EX_i}, A_{EX_i}, LA_{EX_i}, s_i)$*, where:*

– $N_{EX_i} = N_{AS_i}$ *and* $LN_{EX_i} = LN_{AS_i}$
– A_{EX_i} *and* LA_{EX_i} *are empty*
– $\forall$ $(\mu, \alpha, \mu') \in A_{AS_i}$, *with* $\alpha \neq \tau$
  - *Create a new node* μ_{new} *with a new unique label, add the node to* N_{EX_i} *and the unique label to* LN_{EX_i}
  - *if* (μ, α, μ') *is such that* α *is an input action (i.e.* $\alpha = a$*, for some* a *)*
    * *add the labels* a_i *and* $\overline{a_?}$ *to* LA_{EX_i}
    * *add* (μ, a_i, μ_{new}) *and* $(\mu_{new}, \overline{a_?}, \mu')$ *to* A_{EX_i}
  - *if* (μ, α, μ') *is such that* α *is an output action (i.e.* $\alpha = \overline{a}$*, for some* a *)*
    * *add the labels* $\overline{a_i}$ *and* $a_?$ *to* LA_{EX_i}
    * *add* $(\mu, a_?, \mu_{new})$ *and* $(\mu_{new}, \overline{a_i}, \mu')$ *to* A_{EX_i}

– $\forall$ $(\mu, \tau, \mu') \in A_{AS_i}$ *add* τ *to* LA_{EX_i} *and* (μ, τ, μ') *to* A_{EX_i}

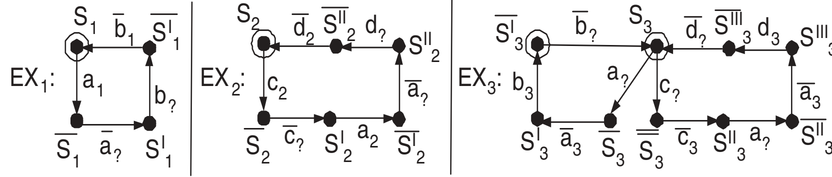

Fig. 7. EX-Graphs of the example

iv) Each EX-Graph represents a partial view of the connector behavior. It is partial since it only reflects the expectations of a single component. We derive the connector global behavior through an EX-Graphs unification algorithm. We refer to [13, 15] for a formal definition of the EX-Graphs unification algorithm. Informally this unification algorithm is based on syntactic unification. For each step the unification procedure attempts to match actions on known communication channels (i.e. terms) in a EX-Graph with actions on unknown communication channels (i.e. variables) in another EX-Graph. Each match represents a new transition (in the connector graph) from the current node to the next new (i.e. not yet considered) adjacent node. Then the algorithm proceeds in the unification procedure from each adjacent node. It is worthwhile noticing that in a generic step it could be possible to not find matches representing new transitions from the current node. In this case we obtain a stop node (i.e. a node without outgoing arcs) in the connector graph. In Figure 8 we show the effect of the first step of the above unification procedure on the EX-Graphs of our working example.

In Figure 9 we show the connector graph K for the example illustrated in this section. The $i-th$ generated node of the connector graph is annotated as K_i and its label is reported in the figure. The resulting CBA system is built as defined in Definition 2.

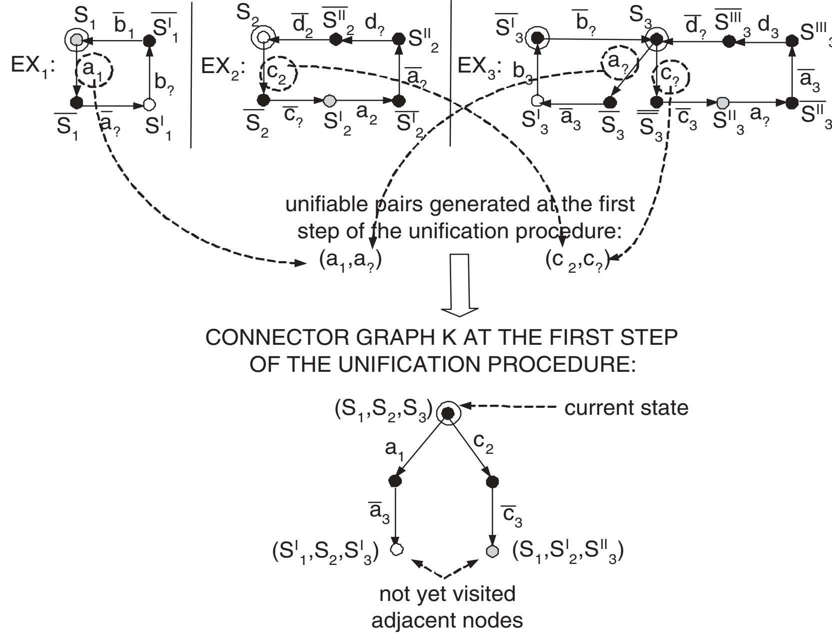

Fig. 8. An example of execution of the first step of the unification procedure

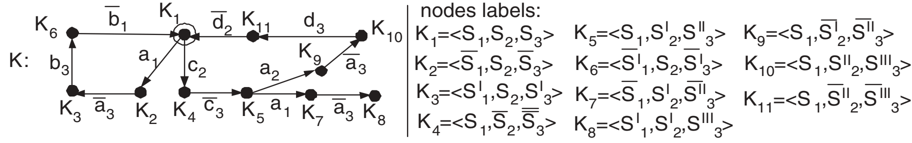

Fig. 9. Connector graph K of the example

3.2 Correctness and Completeness of the Approach

In [15] we have proved that the CBA-system obtained by the connector synthesis process is equivalent to the corresponding CFA-system. To do this we have proved the following proposition:

Proposition 1. *Let T be a CFA-system, and let V be the corresponding CBA-system; then T can be CB-simulated from V.*

Where we define the notion of CB-Simulation as follows:

Definition 6 (CB-Simulation). *Let S and T be two systems and s and t two generic states of the systems;*

– *a relation* $\leq_{CB}$ *is called* CB-simulation *if* $s \leq_{CB} t$ *then:*
  - *the label of s is equal to the label of t;*
  - *if* $s \rightarrow s'$ *then there exists* $n > 0$, $t_0, ..., t_n$ *such that* $t = t_0$ *and for all* $i < n$ *:*
    $t_i \rightarrow t_{i+1}$, $s' \leq_{CB} t_n$.
– *A state t CB-simulates a state s* ($s \leq_{CB} t$) *if it exists a relation of CB-simulation between s and t*
– *A path* σ *CB-simulates a path* ρ, ($\rho \leq_{CB} \sigma$)*, if* ρ *can be partitioned as* $\rho_1\rho_2...$ *and* σ *can be partitioned as* $\sigma_1\sigma_2...$ *in such a way that, for all j, the sequences* ρ_j *and* σ_j *are not empty and* $\rho_j \leq_{CB} \sigma_j$

In other words we have proved the correctness of the synthesis by proving that the CFA-system can be simulated by the synthesized CBA-system under a suitable notion of *"state based"*[2] equivalence called CB-Simulation. The starting point of CB-Simulation is the stuttering equivalence [24]. In [15], we have also proved that the connector does not introduce in the system any new logic (completeness of the synthesis).

3.3 Second Step: Deadlocks Analysis and Recovery

The second step concerns the deadlock freeness analysis, which is performed on the CBA system. Depending on the deadlock type we can operate on the connector in order to obtain a deadlock-free equivalent system. We distinguish between: i) deadlocks due to wrong coordination among components and ii) deadlocks due to wrong components assumptions.

The former are deadlocks due to a bad coordination of components interactions, while the latter are due to components incorrect internal behaviors (e.g. buffer size). It is worthwhile recalling we are dealing with black-box components, whose only known behavior concerns the interactions with the others components into the system. This means that we can only operate on the components interaction behavior and we cannot operate on the components internal behavior. Thus we can only deal with wrong coordination deadlocks. In a black-box setting this is the best we can expect to do in terms of deadlocks prevention. In terms of deadlocks detection we could detect not only wrong coordination deadlocks but also wrong components assumption deadlocks [15]. To do this we need a more complete specification than the bMSCs and HMSCs specification. Actually we need to know not only the observable components interactions but also the hidden interactions of a component. An hidden interaction represents the situation in which the component changes (in an autonomous way) its internal state because of some internal event (e.g. buffer overflow). An hidden interaction is represented on the component's AC-Graph through a τ transition. Refer to [15] for a detailed description of the detection process of wrong components assumptions deadlocks.

[2] By definition, both CFA and CBA systems exhibit only τ transitions.

We can formally define wrong coordination deadlocks as follows:

Definition 7 (Wrong coordination deadlock). *Let K be the connector graph synthesized by the unification of EX-Graphs $EX_1, .., EX_n$ of the CFA-system components $C_1, .., C_n$ respectively. We define a* wrong coordination deadlock *of K as a stop node of K (i.e. a node without outgoing arcs).*

If a wrong coordination deadlock is possible, then this results in a precise connector behavior that is detectable by observing the connector graph. To fix this problem it is enough to prune all the finite branches of the connector transition graph. The pruned connector preserves all the correct (with respect to deadlock freeness) behaviors of CFA-system (Proposition 2). Refer to [15] for a proof of Proposition 2. In Figure 10 we show the wrong coordination deadlock-free connector graph.

Proposition 2. *Let T be a CFA-system, let V_{df} be the corresponding CBA-system based on wrong coordination deadlock-free connector and let Π_T^{inf} be T without the finite paths; then Π_T^{inf} can be CB-Simulated from V_{df}.*

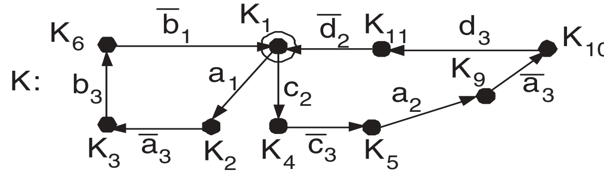

Fig. 10. Deadlock-free connector graph of the example

3.4 Third Step: Behavioral Failures Analysis and Recovery

In this section we formalize the third step of the method of Figure 4. This step concerns properties enforcing on the connector graph.

General Behavioral Properties Specification:

The behavioral properties we want to enforce are related to behaviors of the CFA system that concern coordination policies of the interaction behavior of the components in the CFA system. The CFA behaviors that do not comply to the specified properties represent behavioral failures. A behavior of the CFA system is given in terms of sequences of actions performed by components in the CFA system. In specifying properties we have to distinguish an action α performed by a component C_i with the same action α performed by a component C_j ($i \neq j$). Thus, referring to Definition 1, the behavioral properties (i.e.

coordination properties) can only be specified in terms of visible actions of the components $C_1[f_1], C_2[f_2], .., C_n[f_n]$ where for each $i = 1, .., n$, f_i is a relabelling function such that $f_i(\alpha) = \alpha_i$ for all $\alpha \in Act_i$ and Act_i is the actions set for C_i. By referring to the usual model checking approach [7] we specify every property through a temporal logic formalism. We choose LTL [7] (*Linerar-time Temporal Logic*) as specification language. We define $AP = \{\gamma : \gamma = l_i \vee \gamma = \overline{l}_i \textit{ with } l \in LA_{AC_i}, l \neq \tau, i = 1, .., n\}$ as the set of atomic proposition on which we define the LTL formulas corresponding to the coordination policies.

LTL Syntax:

Referring to [7], we give the standard syntax for the LTL. Given a set of atomic propositions AP, a LTL formula is either:

- if $p \in AP$, then p is a LTL formula;
- if f and g are LTL formulas, then:
  - $!f$ (logical not),
  - $f \vee g$ (logical or),
  - $f \wedge g$ (logical and),
  - $f \longrightarrow g$ (logical implication $\equiv\ !f \vee g$),
  - $\mathbf{X}f$ ("next time" temporal operator),
  - $\mathbf{F}f$ ("eventually" or "in the future" temporal operator),
  - $\mathbf{G}f$ ("always" or "globally" temporal operator),
  - $f\mathbf{U}g$ ("until" temporal operator),
  - and $f\mathbf{R}g$ ("release" temporal operator) are LTL formulas.

LTL Semantics:

Referring to [7], we give the standard semantics for the LTL. We define the semantics of LTL with respect to a Kripke structure. Recall that a Kripke structure M is a quadruple (S, R, S_0, L), where S is the set of states; $R \subseteq S \times S$ is the total transition relation; $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic proposition true in that state; and $S_0 \subseteq S$ is the set of initial states. A *path* in M is an infinite sequence of states, $\pi = s_0, s_1, ..$ such that for every $i \geq 0$ $(s_i, s_{i+1}) \in R$ and $s_0 \in S_0$. We use π^i to denote the *suffix* of π starting at s_i.

If f is a LTL formula the notion $M, \pi^i \models f$ means that f holds along every path starting from the state s_i of the path π in the Kripke structure M. The relation $\models$ is defined inductively as follows (assuming that f and g are LTL formulas):

- for all $f \in AP$, $M, \pi^i \models f$ iff $f \in L(s_i)$;
- $M, \pi^i \models\ !f$ iff not $M, \pi^i \models f$;
- $M, \pi^i \models f \vee g$ iff $M, \pi^i \models f$ or $M, \pi^i \models g$;

- $M, \pi^i \models f \wedge g$ iff $M, \pi^i \models f$ and $M, \pi^i \models g$;
- $M, \pi^i \models f \longrightarrow g$ iff not $M, \pi^i \models f$ or $M, \pi^i \models g$;
- $M, \pi^i \models \mathbf{X}f$ iff $M, \pi^{i+1} \models f$;
- $M, \pi^i \models \mathbf{F}f$ iff there exists a $k \geq i$ such that $M, \pi^k \models f$;
- $M, \pi^i \models \mathbf{G}f$ iff for all $k \geq i$, $M, \pi^k \models f$;
- $M, \pi^i \models f\mathbf{U}g$ iff there exists a $k \geq i$ such that $M, \pi^k \models g$ and for all $i \leq j < k$, $M, \pi^j \models f$;
- $M, \pi^i \models f\mathbf{R}g$ iff for all $k \geq i$, if for every $j < k$, not $M, \pi^j \models f$ then $M, \pi^k \models g$.

Enforcing a Behavioral Property:

The semantics of a LTL formula is defined with respect to a model represented by a Kripke structure. We consider as Kripke structure corresponding to the connector graph K a connector model KS_K that represents the Kripke structure of K. KS_K is defined as follows:

Definition 8 (Kripke structure of a connector graph K). *Let (N, LN, LA, A, k_1) be the connector graph K. We define the Kripke Structure of K, the Kripke structure $KS_K = (V, T, \{k_1\}, LV)$ where $V = N$, $T = A$, $LV = 2^{LA}$ with $LV(k_1) = \{\alpha_i : LA((\overline{k}, k_1)) = \alpha_i,\ (\overline{k}, k_1) \in A\}$. For each $v \in V$ then $LV(v)$ is interpreted as the set of atomic propositions true in state v.*

In Figure 11, we show the Kripke structure of K. The node with an incoming little-arrow is the initial state (i.e. k_1).

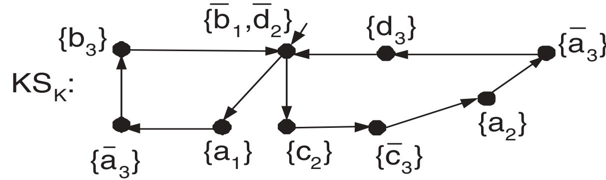

Fig. 11. Kripke structure of K

In Section 3.4 we have described how we can specify a property in terms of desired CFA behaviors. We have also said that all the undesired behaviors represent CFA failures. Analogously to deadlocks analysis, we can solve behavioral failures of the CFA system that are identifiable in the corresponding CBA system with precise behaviors of the synthesized connector. A connector behavior is simply an execution path into the connector graph. An execution path is a sequence of state's transition labels. It is worthwhile noticing that the behavioral properties (i.e. coordination properties) that we specify for the CFA

system are corresponding to behavioral properties of the connector in the CBA system. In fact every action $\gamma = \alpha_i \in AP$ can be seen as the action $\overline{\alpha}$ (into the connector graph) performed on the communication channel that connects C_i to the connector. This is true for construction (see Section 3.1). Thus let P be a behavioral property specification (i.e. LTL formula) for the CFA system, we can translate P in another behavioral property: P_{cba}. P_{cba} is automatically obtained by applying the CCS complement operator to the atomic propositions in P. P_{cba} is the property specification for the CBA system corresponding to P. Then we translate P_{cba} in the corresponding Büchi Automaton [7] $B_{P_{cba}}$:

Definition 9 (Büchi Automaton). *A Büchi Automaton B is a 5-tuple $< S, A, \triangle, q_0, F >$, where S is a finite set of states, A is a set of actions, $\triangle \subseteq S \times A \times S$ is a set of transitions , $q_0 \in S$ is the initial state, and $F \subseteq S$ is a set of accepting states. An* execution *of B on an infinite word $w = a_0 a_1 ...$ over A is an infinite sequence $\sigma = q_0 q_1 ...$ of elements of S, where $(q_i, a_i, q_{i+1}) \in \triangle$, $\forall\ i \geq 0$. An execution of B is* accepting *if it contains some accepting state of B an infinite number of times. B accepts a word w if there exists an accepting execution of B on w.*

Referring to our example we consider the following behavioral property: $P = F((\overline{a}_1 \wedge X(!\overline{a}_1 U \overline{a}_2)) \vee (\overline{a}_2 \wedge X(!\overline{a}_2 U \overline{a}_1)))$. This property specifies all CFA system behaviors that guarantee the evolution of all components in the system. It specifies that the components C_1 and C_2 can perform the action a by necessarily using an alternating coordination policy. In other words it means that if the component C_1 performs an action a then C_1 cannot perform a again if C_2 has not performed a and viceversa. The connector to be synthesized will avoid starvation by satisfying this property. In Figure 12 we show $B_{P_{cba}}$. We recall that $P_{cba} = F((a_1 \wedge X(!a_1 U a_2)) \vee (a_2 \wedge X(!a_2 U a_1)))$; p_0 and p_2 are the initial and accepting state respectively.

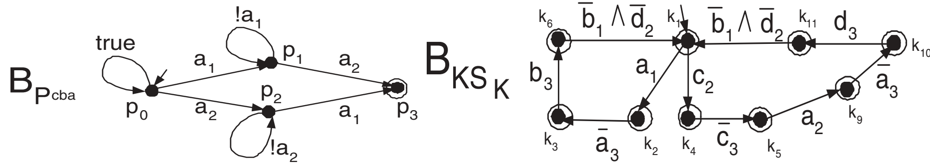

Fig. 12. Büchi Automata $B_{P_{cba}}$ and B_{KS_K} of P_{cba} and KS_K respectively

Given a Büchi Automaton A, $L(A)$ is the *language* consisting of all words accepted by A. Moreover to a Kripke structure T corresponds a Büchi Automaton B_T [7]. We can derive B_{KS_K} as the Büchi Automaton corresponding to KS_K (see Figure 12). The double-circled states are accepting states.

Given $B_{KS_K} = (N, \Delta, \{s\}, N)$ and $B_P = (S, \Gamma, \{v\}, F)$ the method performs the following enforcing procedure in order to synthesize a deadlock-free connector graph that satisfies the property P:

1. *build the automaton that accepts* $L(B_{KS_K}) \cap L(B_{P_{cba}})$*; this automaton is defined as* $B^{K,P}_{intersection} = (S \times N, \Delta', \{<v, s>\}, F \times N)$ *where* $(<r_i, q_j>, a, <r_m, q_n>) \in \Delta'$ *if and only if* $(r_i, a, r_m) \in \Gamma$ *and* $(q_j, a, q_n) \in \Delta$*;*
2. *if* $B^{K,P_{cba}}_{intersection}$ *is not empty return* $B^{K,P_{cba}}_{intersection}$ *as the Büchi Automaton corresponding to the* P*-satisfying execution paths of* K*.*

In Figure 13, we show $B^{K,P_{cba}}_{intersection}$.

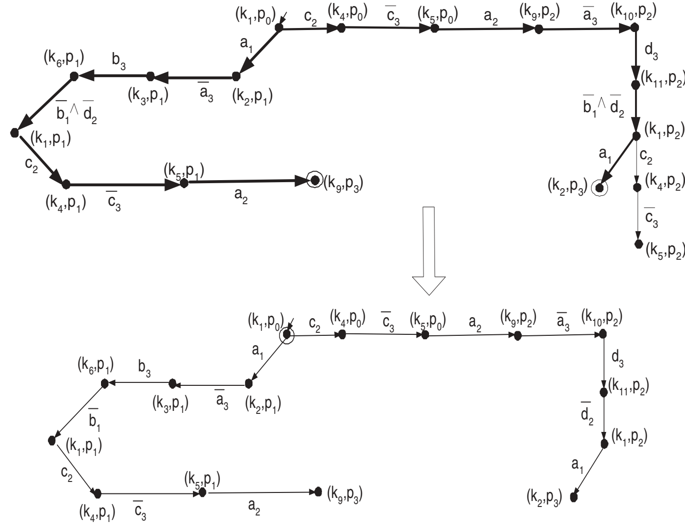

Fig. 13. $B^{K,P_{cba}}_{intersection}$ and deadlock-free property-satisfying connector graph of the explanatory example

Finally our method derives from $B^{K,P_{cba}}_{intersection}$ the corresponding connector graph. This graph is constructed by considering the execution paths of $B^{K,P_{cba}}_{intersection}$ that are only *accepting* (see the path made of bold arrows in Figure 13); we define an *accepting execution path* of $B^{K,P_{cba}}_{intersection}$ as follows:

Definition 10 (Accepting execution path of $B^{K,P_{cba}}_{intersection}$). *Let* $B^{K,P_{cba}}_{intersection} = (S \times N, \Delta', \{<v, s>\}, F \times N)$ *be the automaton that accepts* $L(B_{KS_K}) \cap L(B_{P_{cba}})$*. We define an accepting execution path of* $B^{K,P_{cba}}_{intersection}$ *a sequence of states* $\gamma = s_1, s_2, .., s_n$ *such that* $\forall\ i = 1, .., n\ :\ s_i \in S \times N$*; for* $1 \leq i \leq n-1$*,* $(s_i, s_{i+1}) \in \Delta'$ *and* $(s_n, s_1) \in \Delta'$ *or* $(s_n, s_1) \notin \Delta'$*; and* $\exists\ k = 1, .., n\ :\ k \in F \times N$*.*

It is worthwhile noticing that (depending on the property) an accepting execution path of $B^{K,P_{cba}}_{intersection}$ could be also cyclic (for example if we consider a property using the *always* temporal operator). In this case (in order to build the property-satisfying connector graph) we do not consider the cyclic execution paths without accepting states (refer to [13] for an example in which we find cyclic accepting execution paths in $B^{K,P_{cba}}_{intersection}$).

In Figure 13, we show the deadlock-free property-satisfying connector graph for our explanatory example. Depending on the property, this graph could contain finite paths (i.e. paths terminating with a stop node). Note that at this stage the stop nodes representing accepting states. In fact we have obtained the deadlock-free property-satisfying connector graph by considering only the accepting execution paths of $B^{K,P_{cba}}_{intersection}$, thus stop nodes represent connector states satisfying the property. Once the connector has reached an accepting stop node it will return to its initial state waiting for a new request from an its client. Returning to the initial state is not explicitly represented in the deadlock-free property-satisfying connector graph but it will be implicitly considered in the automatic derivation of the code implementing the deadlock-free property-satisfying connector.

By visiting this graph and by exploiting the information stored in its states and transitions we can automatically derive the code that implements the P-satisfying deadlock-free connector (i.e. the coordinator component) analogously to what done for deadlock-free connectors [14]. The implementation refers to Microsoft COM (*Component Object Model*) components and uses C++ wiht ATL (*Active Template Library*) as programming environment. A single-layered CFA system can be considered a client-server COM system. Into the CFA of our example we have two COM clients components (C_1 and C_2) and one COM server component (C_3). C_3 exports to its clients an interface $IC3$ declaring two methods: input actions a and c on AC_3 (see Figure 5). All the other actions ($\overline{b}$ and $\overline{d}$ on AC_3) are responses to the requests of a and c. The connector component K implements the COM interface $IC3$ of the component C_3 by defining a COM class K and by implementing a wrapping mechanism in order to wrap the requests that C_1 and C_2 perform on component C_3 (actions $\overline{a}$ and $\overline{c}$ on AC_1 and AC_2 of Figure 5). In the following we show fragments of the IDL (*Interface Definition Language*) definition for K, of the K COM library and of the K COM class respectively. *c3Obj* is an instance of the inner COM server corresponding to C_3 and encapsulated into connector component K.

```
import ic3.idl; ... library K_Lib {
   ...
   coclass K {
      [default] interface IC3;
   }
}
```

```
...

class K : public IC3 {
   // stores the current state of the connector
   private static int sLbl;

   // stores the current state of the
   // property automaton
   private static int pState;

   // stores the number of clients
   private static int clientsCounter = 0;

   // channel's number of a client
   private int chId;

   // COM smart pointer; is a reference to
   // the C3 server object
   private static C3* c3Obj;

   ...

   // the constructor
   K() {
      sLbl = 1;
      pState = 0;
      clientsCounter++;
      chId = clientsCounter;
      c3Obj = new C3();
      ...
   }

   // implemented methods
   ...
}
```

In the following we show the deadlock-free property-satisfying code implementing the methods ***a*** and ***c*** of the connector component K. Even if the property P of our example considers a coordination policy only for action a, we have to coordinate also the requests of c in order to satisfy P. Actually, as we can see in Figure 13, the deadlock-free property-satisfying connector has execution paths in which transitions labelled with c there exist.

```
HRESULT a(/* params list of a */) {
  if(sLbl == 1) {
     if((chId == 1) && (pState == 0)) {
        return c3Obj->a(/* params list of a */);
        pState = 1; sLbl = 1; //it goes on the state preceding the next
                              //request of a method from a client
     }
```

```
     else if((chId == 1) && (pState == 2)) {
        return c3Obj->a(/* params list of a */);
        pState = 0; sLbl = 1; //since it has found an accepting stop node,
                              //it returns to its initial state
     }
  }
  else if(sLbl == 5) {
     if((chId == 2) && (pState == 1)) {
        return c3Obj->a(/* params list of a */);
        pState = 0; sLbl = 1; //since it has found an accepting stop node,
                              //it returns to its initial state
     }
     else if((chId == 2) && (pState == 0)) {
        return c3Obj->a(/* params list of a */);
        pState = 2; sLbl = 1; //it goes on the state preceding the next
                              //request of a method from a client
     }
  }

  return E_HANDLE;
}

HRESULT c(/* params list of c */) {
    if(sLbl == 1) {
      if((chId == 2) && (pState == 1)) {
         return c3Obj->a(/* params list of a */);
         pState = 1; sLbl = 5; //it goes on the state preceding the next
                               //request of a method from a client
      }
      else if((chId == 2) && (pState == 0)) {
         return c3Obj->a(/* params list of a */);
         pState = 0; sLbl = 5; //it goes on the state preceding the next
                               //request of a method from a client
      }
   }

   return E_HANDLE;
}
```

3.5 Correctness of the Approach

The following proposition states the correctness of the property enforcing procedure. For the complete proof, please refer to [18]. We prove that the CBA-system based on the property-satisfying deadlock-free connector preserves all the property-satisfying behaviors of the corresponding deadlock-free CFA-system.

Proposition 3. *Let T be a CFA-system, let V_P be the corresponding CBA-system based on a P-satisfying deadlock-free connector and let Π_T^P be T without the finite paths then Π_T^P can be CB-Simulated from V_P except for the execution*

paths ρ of Π_T^P that are also execution paths of the automaton $B_{!P}$ (! is the logical not).

4 An Application Example

In this section we present a real-scale application example of the approach formalized in Section 3. We use our approach to build from a set of suitable COTS components a collaborative writing (CW) system [20, 19, 8, 21]. Refer to [17, 16] for a detailed description of the CW system and of the complete application of the approach to it. In this paper we just reuse a sub-system of the CW system described in [17, 16]. We apply our approach to this sub-system in order to derive an assembly satisfying a property different than the property enforced for the whole system in [17, 16].

Based on a detailed analysis [17] of many CW systems [20, 19, 8, 21] we can identify the COTS computational components that provide the main features of a CW system. These four types of COTS components are showed in Figure 14.

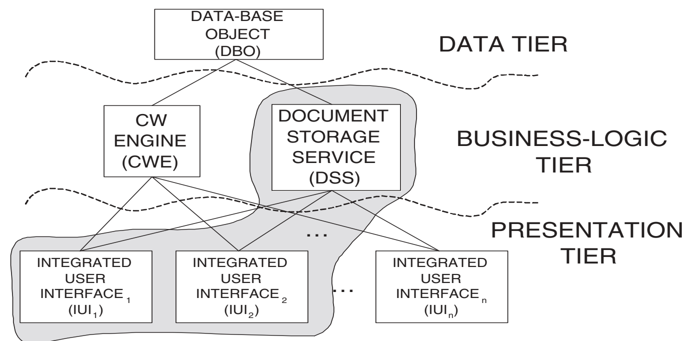

Fig. 14. Architecture of the CW system

Our CW system is a three-tier application. According to our approach, it is composed through coordinators components automatically synthesized in order to satisfy a specified coordination policy. In the following, we briefly describe our CW system and apply our approach to a sub-system of it (namely the grey area in Figure 14).

In order to build our CW system we have identified the following four COTS components. 1) **DBO**: This component is a data-base. The data-base stores all group awareness information useful to support a group activity. 2) **CWE**: This is a CW engine; it provides all services useful to perform a group activity in the CW context. It is an handler of all group awareness information stored in

DBO and of the typical CW activities [17]. 3) **DSS**: A document is a set of document's partitions. This component is an abstraction of the physical container of the shared documents that are logically partitioned according to their structure. In an asynchronous working mode we use version-controlled documents. In a synchronous working mode it is shared among the users and we have to use pessimistic concurrency control. Referring to the version-controlled hierarchical documents [21], a local copy of a document is an alternative and the globally shared document is the last document's version. When a user wants to work in asynchronous mode, DSS expects that all the other users work in asynchronous mode too. In this way DSS can maintain a consistent version of the globally shared document and it evolves in a new consistent version only after the merging of all users alternatives. 4) **IUI**: This component is an integrated environment of tools for editing, navigation, display of awareness communication among the group members and import and export of data from external applications. It is composed of a CW user interface supporting all CW operations, editors for many data types, communication tools such as e-mail and chat.

Let us suppose that the designer of the composed CW system provides a behavioral specification in terms of bMSCs and HMSCs see Figures 15 and 16. The continued lines in the bMSCs are method calls; the hatched lines are the corresponding responses. In our example we consider only two instances of IUI_i: IUI_1 and IUI_2. Moreover we provide the system's behavioral specification only for the part of the CW system identified in the grey area of Figure14. This sub-system is composed by the components IUI_1, IUI_2 and DSS.

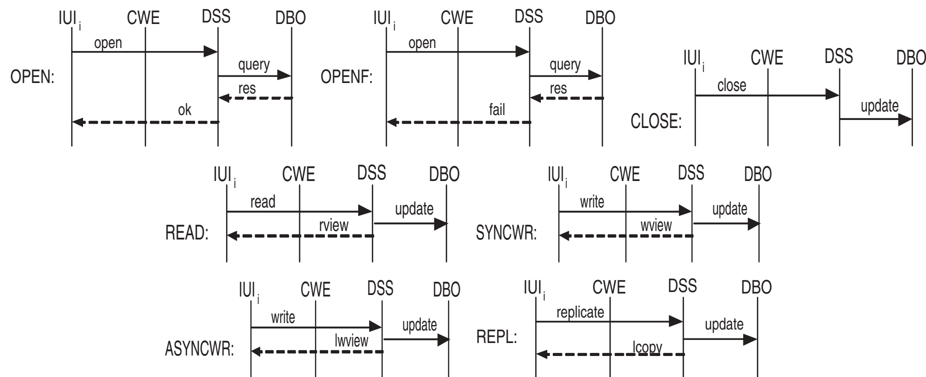

Fig. 15. bMSCs of OPEN, OPENF, CLOSE, READ, SYNCWR, ASYNCWR and REPL scenarios

In Figure 15 we show the bMSCs representing the *'open work session'*, the *'close work session'*, the *'data displaying'*, the *'data synchronous updating'*, the

'data asynchronous updating' and the *'data replication for asynchronous writing'* scenarios. DSS is a server for the two clients IUI_1 and IUI_2. It exports an interface $IDSS$ declaring five methods (**open**, **close**, **read**, **write** and **replicate**) whose behavior is described in the above five scenarios.

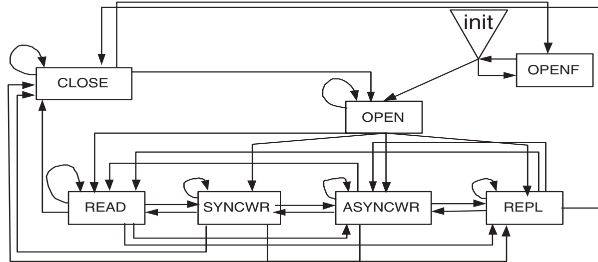

Fig. 16. HMSC of the CW sub-system

In Figure 16 the HMSC specification for the composed CW sub-system is reported in. Let us suppose that the designer of the CW system wants that the composed system satisfies a particular coordination policy. The policy is specified in form of the following behavioral LTL property:
$P = F(\overline{write}_1 \wedge X(!\overline{write}_1 U \overline{write}_2))$. P specifies that when the client IUI_1 wants to update the document it necessarily has to use an alternating coordination policy with IUI_2. Once IUI_1 has performed a **write** request it cannot perform it again before IUI_2 has not performed a **write** request. From the HMSC and bMSCs specification we can automatically derive the AC-Graphs for each component in our CW sub-system (see Figure 17).

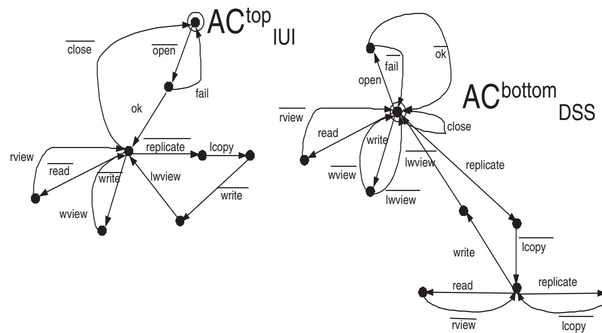

Fig. 17. AC-Graphs of components IUI_1, IUI_2 and DSS

According to Section 3, from the AC-Graphs of IUI_1, IUI_2 and DSS we derive the corresponding AS-Graphs and then we derive the corresponding EX-Graphs (see Figure 18).

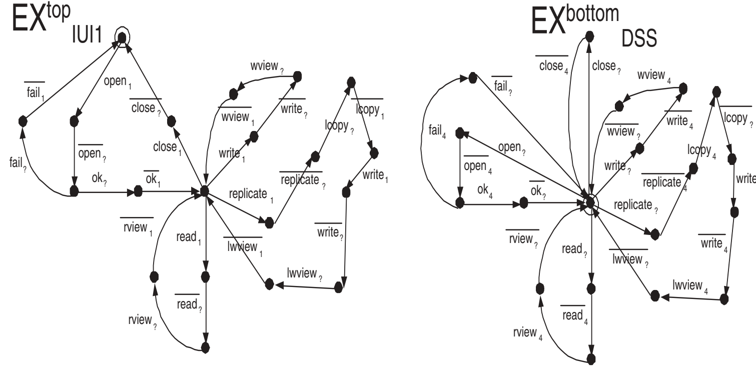

Fig. 18. EX-Graphs of components IUI_1, IUI_2 and DSS

Referring to Figure 18, the EX-Graph of IUI_2 is different from the EX-Graph of IUI_1 only in the identifier of the channel specified in known actions labels (2 instead of 1). We derive the connector global behavior through the EX-Graphs unification algorithm described in Section 3.1. In this paper, for the sake of presentation, we only show a sub-graph of the connector global behavior graph (see Figure 19) and we reduce the analysis of the whole connector to the sub-graph $K_{1.1}$ of the connector global behavior graph. Refer to [17] for a complete visualization of the connector graph.

The sub-connector $K_{1.1}$ has two deadlocks represented by two finite branches. These deadlocks are related to the consistency maintenance in an asynchronous writing scenario. We recall that in order to maintain a consistent version of the shared document, the DSS expects that all users work in asynchronous mode every time another user chooses to work in asynchronous mode. The third-party components IUI_1 and IUI_2 do not respect this DSS assumption. Thus the composed system has concurrency conflicts. This puts in evidence a typical problem in COTS components assembling. In order to synthesize the deadlock-free version of $K_{1.1}$ we simply prune the two finite branches. The deadlock-free $K_{1.1}$ forces IUI_1 and IUI_2 to respect the DSS assumption. According to our approach, once obtained the deadlock-free version of the connector our framework performs the coordination policy enforcing step (see Section 3.4).
In Figure 20 we have shown the P-satisfying and deadlock-free connector model for $K_{1.1}$.

As said in Section 3.4, this behavioral model is enough to derive the deadlock-free property-satisfying connector code that implements the connector methods

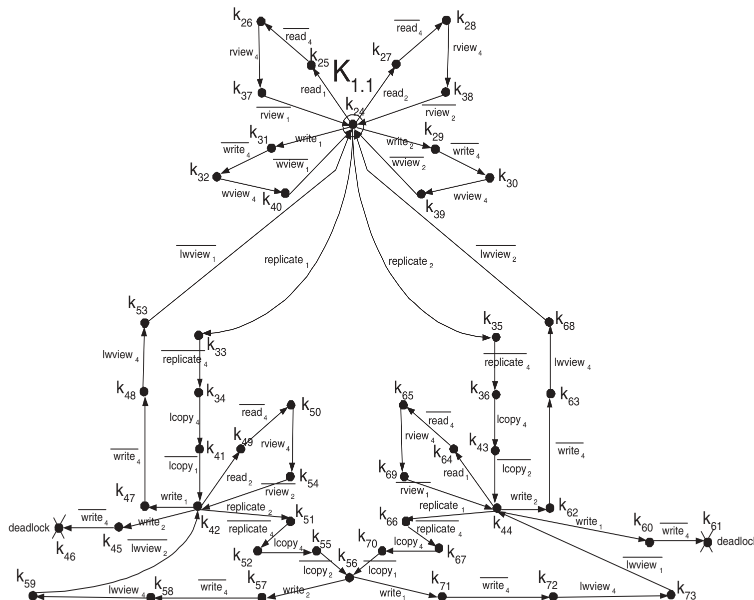

Fig. 19. Sub-graph $K_{1.1}$ of the global connector graph

related to the LTL property specification. The only methods relating to property P are the methods **write** and **replicate** (see Figure 20). For all the others methods (i.e. **open**, **close** and **read**) the connector is a simple delegator since property P has not influence on them. In the following we show the deadlock-free property-satisfying code implementing the method ***write*** of the connector component $K_{1.1}$. We refer to [17], for the complete implementation of $K_{1.1}$. The implementation refers to Microsoft COM (*Component Object Model*) components and uses C++ wiht ATL (*Active Template Library*) as programming environment. The method ***write*** of the inner DSS object gets a parameter of type ***S_DA***. ***S_DA*** is a document alternative *"struct"*. It contains information about the document update to be realized.

```
HRESULT write(S_DA da) {
   if(sLbl == 24) {
      if((chId == 1) && (pState == 0)) {
         return dssObj->write(da);
         pState = 1; sLbl = 24;
      }
      else if((chId == 2) && (pState == 1)) {
         return dssObj->write(da);
         pState = 0; sLbl = 24;
      }
   }
```

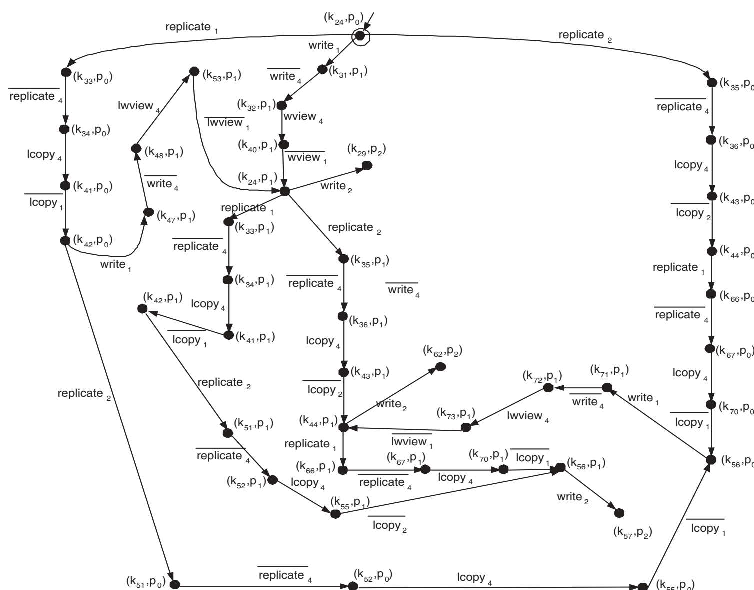

Fig. 20. P-satisfying connector model for $K_{1.1}$

```
   else if(sLbl == 56) {
      if((chId == 1) && (pState == 0)) {
         return dssObj->write(da);
         pState = 1; sLbl = 44;
      }
      else if((chId == 2) && (pState == 1)) {
         return dssObj->write(da);
         pState = 0; sLbl = 42;
      }
   }
   else if(sLbl == 42) {
      if((chId == 1) && (pState == 0)) {
         return dssObj->write(da);
         pState = 1; sLbl = 24;
      }
   }
   else if(sLbl == 44) {
      if((chId == 2) && (pState == 1)) {
         return dssObj->write(da);
         pState = 0; sLbl = 24;
      }
   }

   return E_HANDLE;
}
```

This code is automatically synthesized by visiting the sub-automaton of Figure 20 and by exploiting the information stored in its states and transitions labels. The connector component $K_{1.1}$ is an aggregated server component that encapsulates an instance of the inner DSS component.

5 Related Works

The architectural approach to correct and automatic connector synthesis presented in this paper is related to a large number of other problems that have been considered by researchers over the past two decades. For the sake of brevity we mention below only the works closest to our approach. The most strictly related approaches are in the "*scheduler synthesis*" research area. In the discrete event domain they appear as "*supervisory control*" problem [3, 4, 28]. In very general terms, these works can be seen as an instance of a problem similar to the problem treated in our approach. However the application domain of these approaches is sensibly different from the software component domain. Dealing with software components introduces a number of problematic dimensions to the original synthesis problem: i) the computational complexity and the state-space explosion and ii) in general the approach is not compositional. The first problem can be avoided by using a logical encoding of the system specification in order to use a more efficient data structure (i. e. BDD (Binary Decision Diagram)) to perform the supervisor synthesis; however the second problem cannot be avoided and only under particular conditions it is possible to synthesize the global complete supervisor by composing modular supervisors. While the state-space explosion is a problem also present in our approach, on the other side we have proved in [15] that our approach is compositional to some extents. It means that if we build the connector for a given set of components and later we add a new component in the resulting system we can extend the already available connector and we must not perform again the entire synthesis process.

Other works that are related to our approach, appear in the *model checking of software components* context in which CRA (*Compositional Reachability Analysis*) techniques are largely used [12, 11]. Also these works can be seen as an instance of the general problem formulated in Section 3. They provide an optimistic approach to software components model checking. These approaches suffer the state-space explosion problem too. However this problem is raised only in the worst case that may not be the case often in practice. In these approaches the assumptions that represent the *weakest* environment in which the components satisfy the specified properties are automatically synthesized. However the synthesized environment does not provide a model for the properties satisfying glue code. The synthesized environment may be rather used for runtime monitoring or for components retrieval.

Recently promising formal techniques for the compositional analysis of component based design have been developed [5, 6]. The key of these works is the

modular-based reasoning that provides a support for the modular checking of behavioral properties. The goal of these works is quite different from our in fact they are related only to software components interfaces compatibility check. Thus they provide only a check on component-based design level.

6 Conclusion and Future Work

In this paper we have described a connector-based architectural approach to component assembly. Our approach focusses on detection and recovery of the assembly concurrency conflicts and on enforcing of coordination policies on the interaction behavior of the components constituting the system to be assembled.

A key role is played by the software architecture structure since it allows all the interactions among components to be explicitly routed through a synthesized connector. By imposing this software architecture structure on the composed system we isolate the components interaction behavior in a new component (i.e. the synthesized connector) to be inserted into the composed system. By acting on the connector we have two effects: i) the components interaction behavior can satisfies the properties specified for the composed system and ii) the global system becomes flexible with respect to specified coordination policies.

Our approach requires to have a bMSC and HMSC specification of the system to be assembled. Since these kinds of specifications are common practice in real-scale contexts, this is an acceptable assumption. Moreover we assumed to have a LTL specification of the coordination policies to be enforced.

The complexity of the synthesis and analysis algorithm is exponential either in space and time. This value of complexity is obtained by considering the unification process complexity and the size of the data structure used to build the connector graph. At present we are studying better data structures for the connector model in order to reduce their size. By referring to the automata based model checking [7], we are also working to perform on the fly analysis during the connector model building process. Other possible limits of the approach are: i) we completely centralize the connector logic and we provide a strategy for the connector source code derivation step that derives a centralized implementation of the connector component. We do not think this is a real limit because even if we centralize the connector logic we can actually think of deriving a distributed implementation of the connector component; ii) we assume that an HMSC and bMSC specification for the system to be assembled is provided. Although this is reasonable to be expected, it is interesting to investigate testing and inspection techniques to directly derive from a COTS (black-box) component some kind (possibly partial) behavioral specification; iii) we assume also an LTL specification for the coordination policy to be enforced. It is interesting to find a more user-friendly coordination policy specification; for example by extending the HMSC and bMSC notations to express more complex system's components interaction behaviors.

Acknowledgements

This work has been partially supported by Progetto MIUR SAHARA and by Progetto MURST CNR-SP4.

References

[1] Itu telecommunication standardisation sector, itu-t reccomendation z.120. message sequence charts. (msc'96). Geneva 1996.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions On Software Engineering and Methodology, Vol. 6, No. 3, pp. 213-249*, 6(3):213–249, July 1997.

[3] 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.

[4] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39(2), February 1994.

[5] L. de Alfaro and T. Heinzinger. Interface automata. In *ACM Proc. of the joint 8th ESEC and 9th FSE, ACM Press*, Sep 2001.

[6] L. de Alfaro and T. Heinzinger. Interface theories for component-based design. In *In Proc. of EMSOFT'01: Embedded Software, LNCS 2211, pp. 148-165. Springer-Verlang*, 2001.

[7] O. G. Edmund M. Clarke, Jr. and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, London, England, 2001.

[8] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407, 1989.

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

[10] D. Garlan and D. E. Perry. *Introduction to the Special Issue on Software Architecture, Vol. 21. Num. 4. pp. 269-274*, April 1995.

[11] 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.

[12] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. *Proc. 17th IEEE Int. Conf. Automated Software Engineering 2002*, September 2002.

[13] P. Inverardi and M. Tivoli. Failure-free connector synthesis for correct components assembly. *Specification and Verification of Component-Based Systems (SAVCBS'03) - Workshop at ESEC/FSE 2003. September 1-2, 2003. Helsinki, Finland.*

[14] 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.

[15] P. Inverardi and M. Tivoli. Connectors synthesis for failures-free component based architectures. *Technical Report, University of L'Aquila, Department of Computer Science, http://sahara.di.univaq.it/tech.php?id_tech=7 or http://www.di.univaq.it/~tivoli/ffsynthesis.pdf*, ITALY, January 2003.

[16] P. Inverardi, M. Tivoli, and A. Bucchiarone. Automatic synthesis of coordinators of cots group-ware applications: an example. In *International Workshop on Distributed and Mobile Collaboration (DMC 2003). To be published by the IEEE Computer Society Press in the post-proceedings of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003)*, 9-11 June, Linz, Austria 2003.

[17] P. Inverardi, M. Tivoli, and A. Bucchiarone. Coordinators synthesis for cots group-ware systems: an example. *Technical Report, University of L'Aquila, Department of Computer Science, http://www.di.univaq.it/tivoli/cscw_techrep.pdf*, ITALY, March 2003.

[18] P. Inverardi, M. Tivoli, and A. Bucchiarone. Failures-free connector synthesis for correct components assembly. *Technical Report, University of L'Aquila, Department of Computer Science, http://www.di.univaq.it/tivoli/ffs_techrep.pdf*, ITALY, March 2003.

[19] M. Koch. Design issues and model for a distributed multi-user editor. *Computer Supported Cooperative Work, International Journal*, 5(1), 1996.

[20] M. Koch and J. Kock. Using component technology for group editors - the iris group editor environment. In *In Proc. Workshop on Object Oriented Groupware Platforms*, pages 44–49, Sep 1997.

[21] B. G. Lee, K. H. Chang, and N. H. Narayanan. A model for semi-(a)synchronous collaborative editing. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work, ECSCW 93*, pages 219–231, September 1993.

[22] 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.

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

[24] R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

[25] P.Inverardi and M.Tivoli. Automatic failures-free connector synthesis: An example. *Technical Report, published on the Monterey 2002 Workshop Proceedings: Radical Innovations of Software and Systems Engineering in the Future, Universita' Ca' Foscari di Venezia, Dip. di Informatica, Technical Report CS-2002-10*, September 2002.

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

[27] D. Taubner. Finite representations of ccs and tcsp programs by automata and petri nets. *LNCS 369*, 1989.

[28] E. Tronci. Automatic synthesis of controllers from formal specifications. *Proc. of 2nd IEEE Int. Conf. on Formal Engineering Methods*, December 1998.

[29] 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*, Vienna, Sep 2001.