Failure-free Coordinator Synthesis for Correct Components Assembly

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

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

ABSTRACT

One of the main challenges in components assembly is related to the ability to predict possible coordination policies of the components interaction behavior by only assuming a limited knowledge of the single components computational behavior. Our answer to this problem is a software architecture based approach in which the software architecture imposed on the coordinating part of the system, allows for detection and recovery of COTS (*Commercial-Off-The-Shelf*) concurrency conflicts and for the enforcing of coordination policies on the interaction behavior of the components into composed system. Starting from the specification of the system to be assembled and of the coordination policies for the components interaction behavior, we develop a framework which automatically derives the glue code for the set of components in order to obtain a conflict-free and coordination policy-satisfying system.

1. INTRODUCTION

One of the main challenges in components assembly is related to the ability to predict possible coordination policies of the components interaction behavior by only assuming a limited knowledge of the single components computational behavior. Our answer to this problem is a software architecture based approach [11, 10] in which the software architecture imposed on the coordinating part of the system, allows for detection and recovery of COTS (*Commercial-Off-The-Shelf*) [17] concurrency conflicts and for the enforcing of coordination policies on the interaction behavior of the components into composed system. Building a system from a set of COTS components introduces problems related to their truly black-box nature. Since system developers have no method of looking inside the box, they can only operate on components interaction behavior to enforce coordination policies of the components into assembled system. 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]. We recall that a software architecture is defined as *"the structure of the components of a system, their interrelationships, principles and guidelines governing their design and evolution over time, plus a set of connectors that mediate communication, coordination or cooperation among components."* [7].

Our approach is to compose systems by assuming a well defined architectural style [11] in such a way that it is possible to detect and to fix software anomalies. An architectural style is defined as *"a set of constraints on a software architecture that identify a class of architectures with similar features"* [4]. Moreover we assume that a specification of the desired assembled system is available and that a precise definition of the coordination policies to enforce 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 conflict-free and coordination policy-satisfying system is obtained. The assembly code implements an explicit software connector (i.e. a coordinator) 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 concurrency conflicts can be avoided and the specified coordination policies can be enforced on the interaction behavior of the others components into assembled system. Moreover the software architecture imposed on the composed system allows for easy replacement of a connector with another one in order to make the whole system flexible with respect to different coordination policies.

In previous works [11, 10] we limited ourselves to only concurrency conflict avoidance by enforcing only one type of coordination policy namely deadlock-free policy. In [9] we have applied the deadlock-free approach in a real scale context, namely the context of COM/DCOM applications. In this paper we generalize the framework by addressing generic coordination policies of the components into assembled system. In an other work [12] we have applied the framework we show in this paper to an instance of a typical CSCW (*Computer Supported Cooperative Work*) application, that is a collaborative writing (CW) system we have designed.

The paper is organized as follows. Sections 2 and 3 introduce background notions and, by using an explanatory example,

summarize the method concerning the synthesis of coordinators that are only deadlock-free, already developed in [11, 10]. Section 3.3 contains the main contribution of the paper and, by continuing the explanatory example, formalizes the conflict-free coordination policy-satisfying connectors synthesis. Section 4 presents related works and Section 5 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), 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. Components can only communicate via connectors. Direct connection between connectors is disallowed. Components communicate synchronously by passing two type of messages: notifications and requests. A notification is sent downward, while a request is sent upward. A top-domain (bottom-domain) of a component or of a connector is the set of requests sent upward and of received notifications (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]. The CBA style is a generic layered style. For the sake of presentation, in this paper we describe our approach for single-layer systems. In [11] we show how to cope with multi-layered systems.

2.2 Configuration formalization

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*. In order to describe components and system behaviors we use CCS [14] (*Calculus of Communicating Systems*) notation. For the purpose of this paper this is an acceptable assumption. Actually our framework allows to automatically derive these CCS descriptions from *"HMSC (High level Message Sequence Charts)"* and *"bMSC (basic Message Sequence Charts)"* [1] specifications of the system to be assembled [16, 12]. This derivation step is performed by applying a suitable version of a translation algorithm from bMSCs and HMSCs to LTS (*Labelled Transition Systems*) [19]. HMSC and bMSC specifications are common practice in real-scale contexts thus CCS can merely be regarded as an internal to the framework specification language. Since these specifications model finite-state behaviors of a system we will use finite-state CCS:

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

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

Definition 2. Connector Based Architecture (CBA): $CBA \equiv (C_1[f_1] \mid C_2[f_2] \mid ... \mid C_n[f_n] \mid K) \backslash \bigcup_{i=1}^{n} Act_i[f_i]$ 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 CCS process representing the connector.

In Figure 1 we show an example of CFA system and of the corresponding CBA system. The double circled states represent initial states.

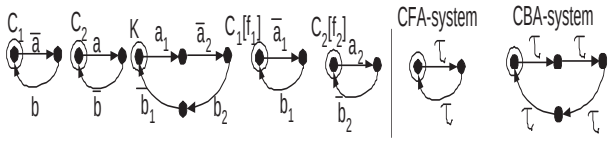

Figure 1: CFA and corresponding CBA

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, C_i, and a set of coordination policies P automatically derive the corresponding CBA system V which implements every policy in P.*

We are assuming that a specification of the system to be assembled is provided. Referring to Definition 1, we assume that for each component a description of its behavior as finite-state CCS term is provided (i.e. LTS *Labelled Transitions System*). Moreover we assume that a specification of the coordination policies to be enforced exists. In the following, by means of a working example, we discuss our method proceeding in three steps as illustrated in Figure 2.

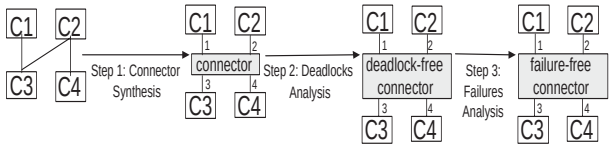

Figure 2: 3 step method

The first step builds a connector (i.e. the coordinator) following the CBA style constraints. The second step performs the concurrency conflicts (i.e. deadlocks) detection and recovery process. Finally, the third step performs the enforcing of the specified coordination policies against the conflict-free connector and then synthesizes a coordination policy-satisfying connector. The first two steps concern the approach already developed in our precedent works [11, 10]. Instead the third step concerns the extension of the approach to deal with generic coordination policy. From the latter we can derive the code implementing the coordinator component which is by construction correct with respect to the coordination policies specification.

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

3.1 First step: Coordinator Synthesis

The first step of our method (see Figure 2) starts with a CFA system and produces the equivalent CBA system. It is worthwhile noticing that this can always be done [11]. We proceed as follows:

i) 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. 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 3 we show the AC-Graphs of the CFA system of our explanatory example. The double-circled states are the initial states. For the transition labels we use a CCS notation (α is an input action and $\overline{\alpha}$ is the corresponding output action).

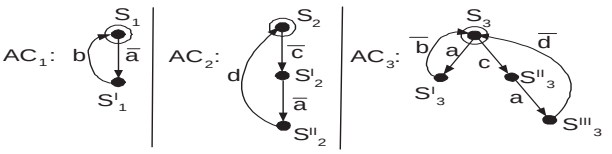

Figure 3: AC-Graphs of the example

We are assuming a client-server components setting. AC_1 and AC_2 are the AC-Graphs of the two client components (i.e. C_1 and C_2). AC_3 is the AC-Graph of the server component (i.e. C_3). C_3 exports two services, namely a and c. c has not a return value. a has either b or d as return values. The input actions on AC_3 represent requests of service from the clients while the output actions represent return values towards the clients. The input actions on AC_1 and AC_2 represent return values from the server while the output actions represent requests of service towards the server.

ii) We derive from AC-Graph the requirements on its environment that guarantee concurrency conflict (i.e. deadlock) freedom. Referring to Definition 1, 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 be in conflict with its environment if the environment can always provide the actions it requires for changing state. This is represented as AS-Graphs (Figure 4):

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}= $\{(\nu,\overline{a},\nu') \mid (\nu,a,\nu') \in A_{AC}\} \bigcup \{(\nu,b,\nu') \mid (\nu,\overline{b},\nu') \in A_{AC}\}$.

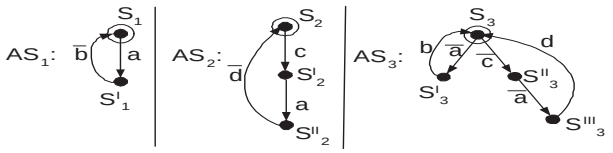

Figure 4: AS-Graphs of the example

Now if we consider Definition 2, the environment of a component can only be represented by connectors, EX-Graph represents the behavior that the component expects from the connectors (Figure 5):

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. α = 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}

iii) Each EX-Graph represents a partial view (i.e. the single component's view) of the connector behavior. The EX-Graph for component C_i (i.e. EX_i) is the behavior that C_i expects from the connector. Thus EX_i has either transitions

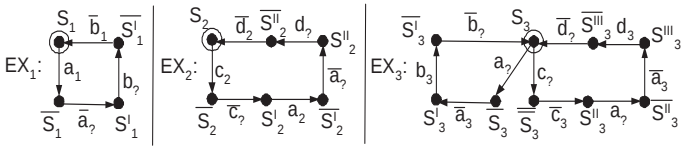

Figure 5: EX-Graphs of the example

labelled with known actions or with unknown actions for C_i. Known actions are performed on the channel connecting C_i to the connector. This channel is known to C_i and identified by a number. Unknown actions are performed on channels connecting other components C_j $(j \neq i)$ to the connector, therefore unknown from the C_i perspective. These channels are identified by the question mark. We derive the connector global behavior through the following EX-Graphs unification algorithm.

Definition 6. EX-Graphs Unification:

- Let $C_1, .., C_n$ be the components in CFA-version of the composed system in such a way that $\{C_1, .., C_h\}$ is the set of null bottom domain components and $\{C_{h+1}, .., C_n\}$ is the set of null top domain components;
- Let $EX_1, .., EX_h, EX_{h+1}, .., EX_n$ be their corresponding EX-Graphs;
- Let $1, .., h, h+1, .., n$ be their corresponding communication channels;
- Let $S_1, .., S_h, S_h+1, .., S_n$ be their corresponding current states.

At the beginning the current states are the initial states.

1. Create the actual behavior graph of the connector, with one node (initial state) and no arcs.
2. Set as current states of the components $EX-Graphs$ the respective initial states.
3. Label the connector initial state with an ordered tuple composed of the initial states of all components (null bottom domain and null top domain). For simplicity of presentation we assume to order them so that the $j-th$ element of the state label corresponds to the current state of the component C_j where $j \in [1, .., h, h+1, .., n]$. This state is set as the connector current state.
4. Perform the following unification procedure:
   (a) Let g be the connector current state. Mark g as visited.
   (b) Let $< S_1, .., S_h, S_{h+1}, .., S_n >$ be the state label of g.
   (c) Generate the set TER of action_terms and the set VAR of action_variables so that $t_i \in TER$, if in $EX_i\ S_i \stackrel{t_i}{\Rightarrow} \overline{S}_i$. Similarly $v_? \in VAR$, if $\exists\ j$ in such a way that in $EX_j\ S_j \stackrel{v_?}{\Rightarrow} \overline{S}_j$.
   (d) For all unifiable pairs $(t_i,\ v_?)$, with $i \neq j$ do:
      i. if $i \in \{1, .., h\}$, $j \in \{h+1, .., n\}$ and they do not already exist then create new nodes (in the connector graph) g_i, g_j with state label $< S_1, .., \overline{S}_i, .., S_h, S_{h+1}, .., \overline{S}_j, .., S_n >$ and $< S_1, .., S'_i, .., S_h, S_{h+1}, .., S'_j, .., S_n >$ respectively,
      where in $AS_i\ S_i \stackrel{t}{\Rightarrow} S'_i$ and in $AS_j\ S_j \stackrel{\overline{v}}{\Rightarrow} S'_j$;
      ii. if $j \in \{1, .., h\}$, $i \in \{h+1, .., n\}$ and they do not already exist then create new nodes (in the connector graph) g_i, g_j with state label $< S_1, .., \overline{S}_j, .., S_h, S_{h+1}, .., \overline{S}_i, .., S_n >$ and $< S_1, .., S'_j, .., S_h, S_{h+1}, .., S'_i, .., S_n >$ respectively,
      where in $AS_i\ S_i \stackrel{t}{\Rightarrow} S'_i$ and in $AS_j\ S_j \stackrel{\overline{v}}{\Rightarrow} S'_j$;
      iii. create the arc $(g,\ t_i,\ g_i)$ in the connector graph;
      iv. mark g_i as visited;
      v. create the arc $(g_i,\ \overline{v}_j,\ g_j)$ in the connector graph.
   (e) Perform recursively this procedure on all not marked (as visited) adjacent nodes of current node.

In Figure 6 we show the connector graph for the example illustrated in this section. The resulting CBA system is built as defined in Definition 2.

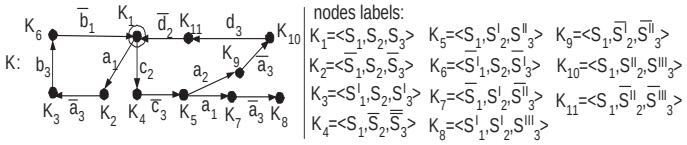

Figure 6: Connector graph of the example

In [11] 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 that the CFA-system can be simulated by the synthesized CBA-system (correctness of the synthesis) under a suitable notion of *"state based"*[2] equivalence called CB-Simulation [11]. The starting point of CB-Simulation is the stuttering equivalence [15]. We have also proved that the connector does not introduce in the system any new logic (completeness of the synthesis).

3.2 Second step: Concurrency conflicts avoidance

The second step concerns the concurrency conflicts avoidance, which is performed on the CBA system. In [11], we have proved that if a concurrency conflict (i.e. 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 [11]. In Figure 7 we show the concurrency conflict-free connector graph.

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

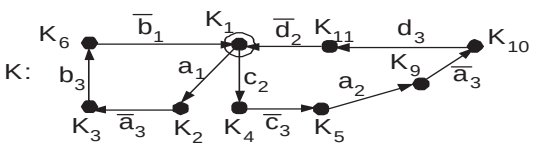

Figure 7: Deadlock-free connector graph of the example

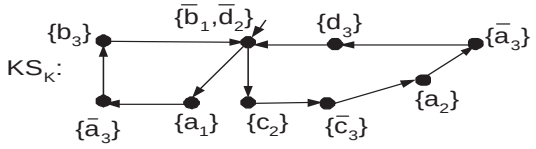

Figure 8: Kripke structure of K

3.3 Third step: Coordination policies enforcing

In this section we formalize the third step of the method of Figure 2. This step concerns the coordination policy enforcing on the connector graph.

3.3.1 Generic coordination policies 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 [5] we specify every property through a temporal logic formalism. We choose *LTL* [5] (*Linerar-time Temporal Logic*) as specification language. We define $AP = \{\gamma : \gamma = l_i \vee \gamma = \overline{l}_i \text{ with } l \in LA_{AC_i}, l \neq \tau, i = 1, .., n\}$ as the set of atomic proposition on which we define the LTL formulas corresponding to the coordination policies. Refer to [5] for the standard LTL syntax and semantics.

3.3.2 Enforcing a coordination policy:

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 7. 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 $KS_K = (N, A, \{k_1\}, LV)$ where $LV \subseteq 2^{LA}$ with
$LV(k_1) = \{\alpha_i : LA((\overline{k}, k_1)) = \alpha_i, (\overline{k}, k_1) \in A\}$. For each $v \in N$ then $LV(v)$ is interpreted as the set of atomic propositions true in state v.

In Figure 8, we show the Kripke structure of K. The node with an incoming little-arrow is the initial state. In Section 3.3.1 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 [5] $B_{P_{cba}}$:

Definition 8. 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 9 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.

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 [5]. We can derive B_{KS_K} as the Büchi Automaton corresponding to KS_K (see Figure 9). The double-circled states are accepting states. Given $B_{KS_K} = (N, A', \Delta, \{s\}, N)$ and $B_P =$

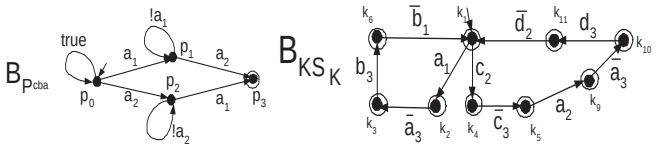

Figure 9: Büchi Automata $B_{P_{cba}}$ and B_{KS_K} of P_{cba} and KS_K respectively

$(S, A'', \Gamma, \{v\}, F)$ 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* $B^{K,P}_{intersection} = (S \times N, A', \Delta', \{< v, s >\}, F \times N)$ *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 10, we show $B^{K,P_{cba}}_{intersection}$.

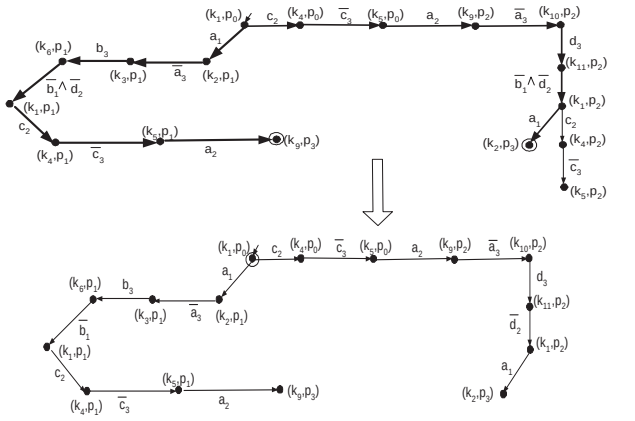

Figure 10: $B^{K,P_{cba}}_{intersection}$ and deadlock-free property-satisfying connector graph of the explanatory example

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 10); we define an *accepting execution path* of $B^{K,P_{cba}}_{intersection}$ as follows:

Definition 9. 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 $\exists\ k = 1, .., n\ :\ k \in F \times N$.

In Figure 10, 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 [10]. The implementation refers to Microsoft COM (*Component Object Model*) components and uses C++ wiht ATL (*Active Template Library*) as programming environment. 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 3). 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 $\boldsymbol{a}$ and $\boldsymbol{c}$ 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 10, 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;
}
```

In [13] we prove the correctness of the property enforcing procedure. 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.

4. 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, 18]. 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. There are two main problems with this approach: 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 [11] that our approach is always compositional. 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 [8]. 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. 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 [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.

5. CONCLUSION AND FUTURE WORKS

In this paper we have described a connector-based architectural approach to component assembly. Our approach focusses on detection and recovery of the assembly behavioral failures. 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. We have applied our approach to an example and we have discussed its implications on the actual nature of black-box components. As far as components are concerned we only assumed to have a CCS description of the components behavior. For the purpose of this paper this is an acceptable assumption. However our framework allows to automatically derive these CCS descriptions from specifications that are common practice in real-scale contexts. For behavioral properties we have shown in this paper how to go beyond deadlock. 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 [5], 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 behavioral property to be checked. It is interesting to find a more user-friendly property 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.

6. 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] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 2001.

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

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

[8] 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.

[9] P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *Journal of Systems and Software, Volume 65, Issue 3, 15 March 2003, Pages 173-183, Component-Based Software Engineering.*

[10] 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.

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

[12] P. Inverardi, M. Tivoli, and A. Bucchiarone. Automatic synthesis of coordinators of cots group-ware applications: an example. *International Workshop on Distributed and Mobile Collaboration (DMC 2003), http://www.di.univaq.it/tivoli/Publications /Full/DMC2003.pdf, 9-11 June, Linz, Austria WETICE 2003.*

[13] 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.

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

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

[16] P.Inverardi and M.Tivoli. Automatic failures-free connector synthesis: An example. *published on the post-workshop proceedings of Monterey 2002 Workshop: Radical Innovations of Software and Systems Engineering in the Future, Venezia (ITALY)*, September 2002.

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

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

[19] 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.