Coordination-Policy Based Composed System Behavior Derivation

Yiming Lau, Wenyun Zhao, Xin Peng, Zhixiong Jiang, Liwei Shen
Software Engineering Lab, Computer Science and Engineering Department,
Fudan University, Shanghai 200433, China
{051021050, wyzhao, pengxin, 051021049, 061021062}@fudan.edu.cn

Abstract

The coordination-policy that components interactions satisfied often determines the properties of nowadays component-based information systems, e.g. Safety, Liveness and Fairness etc. Therefore, how to derive coordination-policy satisfying behavior all out of such system to achieve better system properties is of a significant problem that needs to be solved. Aim to this problem, we propose an optimistic policy-satisfying behavior derivation approach in this paper. The main idea of the approach is to automatically construct a Coordination Environment (CE) for such composed system that system components can work together in a deadlock-free and policy-satisfying manner, and so as to obtain desired system properties. In this approach, component-based information system is modeled by Interface Automaton Network (IAN), and component coordination-policies are specified by LTL. To explain the correctness and validity of this approach, we give a corresponding example certification.

1. Introduction

Component-Based Software Development (CBSD) provides a novel way to utilized component to form nowadays complex information system, i.e. systems can be developed by composing and integrating existing reusable software components (e.g., commercial-off-the-shelf (COTS) components)[3]. Then the behaviors of the composed information system are represented by the interactions of its components. And the coordination-policy that components interactions satisfied often determines the quality and properties of such composed systems, e.g. Safety, Liveness and Fairness etc [4]. Thus, how to derive coordination-policy satisfying behavior all out of such composed system is of a significant problem that needs to be solved, in order to achieve better system properties. In practice, system developer usually adopts a pessimistic model checking approach to solve this problem, i.e., checking system against such properties then modifying system designs. But it is not always possible, e.g. to modify COTS components which are usually black-box.

Aim to this problem, we propose an optimistic policy-satisfying behavior derivation approach in this paper. The main idea of the approach is to automatically construct a Coordination Environment (CE) for such system that system components can work together in a deadlock-free and policy-satisfying manner, and so as to achieve desired system properties.

Interface Automata (IA) [1, 2] is a light-weight formal language which can be used to model the dynamic temporal behavior of software component interface. Since the interfaces are often much simpler than the corresponding implementations, one of the advantages of IA is that the state space of the model will be much less than the corresponding statecharts of implementations. Therefore, we use Interface Automata to model the behavior of component, and Interface Automaton Network (IAN) to specify the behavior of the composed information system. And by referring to the usual model checking approach [5], we employ Linear-time Temporal Logic (LTL) formalism to specify every coordination-policy. Coordination-policies usually model the desired components interactions, which are used to ensure composed system's specified properties [12, 16].

Concretely, our approach is divided into three steps. The first step is the unification of the underlying semantic of the behavior model of the composed system and its specified coordination-policy, i.e., IAN and LTL formula. In this step, our approach constructs corresponding Büchi automaton of IAN and the LTL-based coordination-policy P, e.g., B_{IAN} and B_P. Here, let $L(B_{IAN})$ and $L(B_P)$ be the languages consisting of all words accepted by B_{IAN} and B_P, respectively. Therefore, in our approach the derivation of policy-satisfying behaviors all out of composed system can be converted into the construction of the cross-product of B_{IAN} and B_P which can accept $L(B_{IAN}) \cap L(B_P)$, and then deriving all accepting paths [6] all out of the product Büchi automaton, in the second step. Finally,

1530-1362/07 $25.00 © 2007 IEEE
DOI 10.1109/ASPEC.2007.37

based on the derived system behaviors, our approach automatically generates code that implements the Coordination Environment for the composed system. Here, Coordination Environment is similar to connectors or adaptors [5], while it manages to drive components in the newly composed system to work together in a deadlock-free and coordination-policy satisfying manner.

The paper is organized as follows. Section 2 presents related works. In section 3, we introduce interface automata and its formal definitions. In section 4, we show our motivating example and introduce interface automaton network and its formal definitions. Section 5 shows how to specify component coordination-policy in LTL. Section 6 presents the formalization of our policy-satisfying behavior derivation approach. Section 7 concludes this paper and gives some future work.

2. Related Works

The approach presented in this paper is related to a large number of other problems that have been considered by many researchers. 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. They appear as supervisory control problem [7, 8, 14] in the discrete event domain. 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.

Our research is also related to work in the area of adaptor synthesis developed by [10, 11, 15, 16]. The main idea is to modify the interaction mechanisms that are used to glue components together so that compatibility can be achieved. And this is done by integrating the interaction protocol into components by means of adaptors. However, they are limited to only consider pessimistic syntactic incompatibilities between the interfaces of components and they do not allow the kind of optimistic coordination behavior derivation that our approach supports.

Other works that are related to ours appear in the model checking of software components, in which compositional reachability analysis techniques are largely used [9, 13]. They provide an optimistic approach to software components model checking. 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 is just used for runtime monitoring. While, in our approach the automatically synthesized coordination environment can be used to drive components in the newly composed system to work together in a deadlock-free and coordination-policy satisfying manner, and so as to achieve desired properties of the composed system.

3. Interface Automata

Interface automata is a light-weight formal language to describe the temporal aspects of software component interfaces [1]. The basic idea is an *optimistic* view on composition of components. That is, when the components are composed, their environment assumptions should be also composed. Accordingly, if there are some helpful environments satisfy both of their environment assumptions, these components are considered compatible. Specifically, interface automata are designed to capture effectively both input assumptions and output guarantees about the order of the interactions between component and its environment.

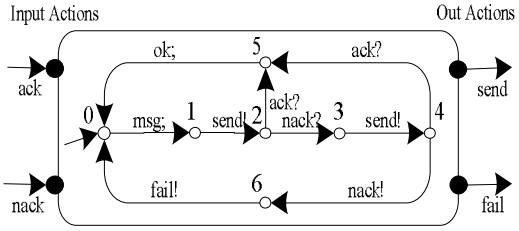

Fig.1.The example of an interface automaton

For example, Figure1 shows an interface automaton of a communication component [2]. The symbol "?" (resp. "!", ";") appended to the name of actions denotes that the action is an input (resp. output, internal) action. An arrow without source denotes the initial state of the interface automaton. In state 0, it only accepts the input msg, indicating the assumption that once the method msg is called, then before another call of msg, the environment will wait for an ok or fail response coming from other component.

Definition 3.1 (Interface Automata, IA). An interface automaton is a tuple $C = (V_C, v_C^{Init}, A_C^I, A_C^O, A_C^H, \Gamma_C)$ where:

- V_C is a finite set of states, each state $v \in V_C$;
- $v_C^{Init} \in V_C$ is the initial state;
- A_C^I, A_C^O and A_C^H are mutually disjoint sets of input, output and internal actions, and $A_C = A_C^I \cup A_C^O \cup A_C^H$ is the set of all actions;
- $\Gamma_C \subseteq V_C \times A_C \times V_C$ is a set of transitions.

If $a \in A_C^I$ (resp. $a \in A_C^O$, $a \in A_C^H$), then (v, a, v') is called an input (resp. output, internal) transition.

Definition 3.2 (Behavior of IA). For an interface automaton $C = (V_C, v_C^{Init}, A_C^I, A_C^O, A_C^H, \Gamma_C)$, a state sequence $v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} v_n \xrightarrow{a_n} v_{n+1}$ is a behavior of C iff $v_0 = v_C^{Init}$, and for each $i(0 \le i \le n)$, there is $(v_i, a_i, v_{i+1}) \in \Gamma_C$.

4. Interface Automaton Network and Component-Based Composed System

4.1 Motivating Example

A composed system can be seen as a components network. Then its behaviors are the product of its components behavior. So, in this paper, we propose to use interface automaton network (IAN) to model component-based composed information system. IAN is composed of a set of interface automata which represent the abstractions of individual software components. And in IAN, two IA is synchronized by their shared actions. In the following subsection, we will introduce the formal definition of IAN.

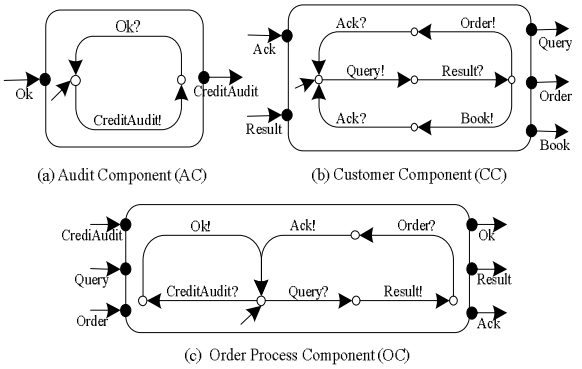

Fig.2. The IAN of the Online Shopping System

Consider the simplified Online Shopping System (OSS) in Figure 2. It consists of three business component, i.e. Audit Component (AC), Customer Component (CC) and Order Process Component (OPC). Audit Component, which is provided by the Banking System, enables online-shop to conduct a *Credit-Audit* operation upon online-customer's credit account. Customer Component enables online-customer to *Query* the list of products, and then *Order* or *Book* his favorite goods form online-shop. Order Process Component is the platform upon which online-shop and online-customer do their business. In our example, we assume that Order Process Component doesn't support Booking business between online-shop and online-customer. So it doesn't going to synchronize on *Booking* operation with Customer Component. And this will cause a deadlock in the overall composed system.

In order to ensure the *payment security* (which means online-customer has enough credit to order goods) of the OSS, if online-customer conducts an *Order* action then he will not perform it again if online-shop has not employed Audit Component conducting an *CreditAudit* operation upon online-customer's credit account and vice versa. Therefore, the designer of the OSS wants the composed system should satisfies a specified coordination-policy among Audit, Customer and Order Process components. That is Customer component and Audit component should perform *Order* and *CreditAudit* actions by necessarily using an *alternating-coordination-policy* in the newly composed OSS system. How to specify the coordination-policy in form of LTL formula [6] will be discussed in section 5.

4.2 Interface Automaton Network

An interface automaton network consists of a set of interface automata which represent the behavior abstractions of software components. An input action of one interface automaton may coincide with an output action of the other one, and then these two interface automata will synchronize on such shared actions, while asynchronously interleaving other actions. Those synchronized actions between any two interface automata are denoted by $shared(C_i, C_j) = A_{Ci} \cap A_{Cj} = (A_{Ci}^O \cap A_{Cj}^I) \cup (A_{Ci}^I \cap A_{Cj}^O)$. The states, actions and transitions of the IAN are defined as follows. The definition of IAN is the generalization of Interface automata composition defined in [1].

Definition 4.1 (Interface Automaton Network, IAN). Interface automaton network (IAN) is a tuple $N = (K, S)$, where

- $K = \{C_1, C_2, \ldots, C_n\}$ is a set of composable interface automata;
- $S = \{shared(C_i, C_j) \mid 1 \le i, j \le n, i \ne j\}$ is a set of shared actions.

Definition 4.2 (States and Actions set of IAN). Let $N = (K, S)$ be a IAN where $K = \{C_1, C_2, \ldots, C_n\}$:

- a state $\bar{v}$ of N is in $V_{C1} \times V_{C2} \times \ldots \times V_{Cn}$, that is, $\bar{v} = (v_1, v_2, \ldots, v_n)$ ($v_i \in V_{Ci}$, $1 \le i \le n$);
- the initial state of N is $v_N^{Init} \in V_N$, and $v_N^{Init} = (v_{C1}^{Init}, v_{C2}^{Init}, \ldots, v_{Cn}^{Init})$;
- the set of actions of N is $A_N = A_N^I \cup A_N^O \cup A_N^H$, where the set of input actions is $A_N^I = (\bigcup_{1 \le i \le n} A_{Ci}^I) \setminus S$,

the set of output actions is $A_N^O = (\bigcup_{1\le i\le n} A_{Ci}^O) \setminus S$, and the set of internal actions is $A_N^H = (\bigcup_{1\le i\le n} A_{Ci}^H) \cup S$.

Definition 4.3 (State transition of IAN). Let $N = (K, S)$ be an IAN, $\bar{v}$ and $\bar{v}'$ be its states where $\bar{v} = (v_1, v_2, ..., v_n)$ and $\bar{v}' = (v_1', v_2', ..., v_n')$. The system can change from state $\bar{v}$ to state $\bar{v}'$ by a transition $\bar{v} \xrightarrow{a} \bar{v}'$ if one of the following conditions holds:

- for an action $a \in shared(C_i, C_j)(1 \le i, j \le n, i \ne j)$, there is a transition $(v_i, a!, v_i') \in \Gamma_{Ci}$ (!denotes output), and $(v_j, a?, v_j') \in \Gamma_{Cj}$ (?denotes input), and $v_k = v_k'$ for any $k(k \ne i, j, 1 \le k \le n)$;
- for an action $a \notin shared(C_i, C_j)(1 \le i, j \le n, i \ne j)$, there is a transition $(v_k, a, v_k') \in \Gamma_{Ck}$ in $C_k(1 \le k \le n)$, and $v_i = v_i'$ for any $i(i \ne k, 1 \le i \le n)$.

In fact, there may be some contradictions between the environment assumptions of two automata which have shared actions [1]. That is, one automaton may produce an output action that is an input action of another automaton, but it is not accepted by the latter one. Then, those corresponding composed states of IAN which may cause composition *deadlock* are named *illegal states*. An algorithm has been presented in [2] to derive compatible composition of interface automata by removing such illegal states recursively. As a result, we use *comp*(*N*) to denote the compatible IAN which only contain *deadlock-free* composed behaviors of the composed system.

Definition 4.4 (Behavior of Compatible IAN). Let *comp*(*N*) = (*K, S*) be a compatible IAN. A state sequence $\bar{v}_0 \xrightarrow{a_0} \bar{v}_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} \bar{v}_n \xrightarrow{a_n} \bar{v}_{n+1}$ is a behavior of *comp*(*N*) iff $\bar{v}_0 = v_{comp(N)}^{Init}$, and for each i $(0 \le i \le n)$, there is $(\bar{v}_i, a_i, \bar{v}_{i+1}) \in \Gamma_{comp(N)}$.

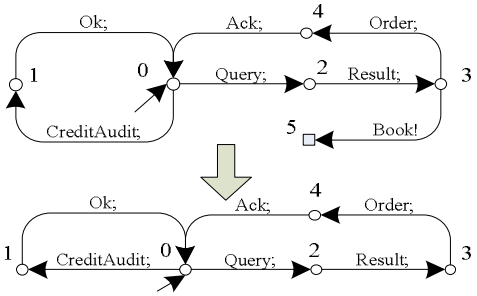

Fig.3. Deadlock-free behaviors of the IAN

For example, Figure 3 shows the original composed behaviors and deadlock-free behaviors model of our working example. In the original IAN, state 5 is an illegal state, which will lead system OSS into a deadlock. Because after performing an output action *Book*!, Customer Component engages in synchronizing on the shared action of *Ack* with Order Process Component, but in fact Order Process Component is not expected to share this action under such circumstances. In other words, according to Order Process Component's designed behavior, it is not assumed to support such booking business. Therefore, in order to derive the deadlock-free composed behaviors of Audit, Customer and Order Process components in OSS, i.e., compatible IAN, state 5 and its backward transitions in the original IAN should be removed.

Definition 4.5 (Reachability graph of Compatible IAN). Given a compatible IAN *comp*(*N*), a corresponding reachability graph $G = (V; T, LA)$ can be constructed easily as follows, where V is a set of nodes and T is a set of edges:

- for each compatible state $\bar{v}_i$ of *comp*(*N*), there is a node v_i in V ; and for the initial state $\bar{v}_{comp(N)}^{Init}$, the corresponding v_0 in the set V is called *root* node;
- for each transition $(v_i, a_i, v_{i+1}) \in \Gamma_{comp(N)}$, there is an edge $t_i = (v_i, l_i, v_{i+1})$ in T, where l_i in LA is the label on the edge, and $l_i = a_i$.
- for any behavior of *comp*(*N*), there is a path in the reachability graph G. Let $\rho = t_0 \hat{} t_1 \hat{} ... \hat{} t_n$ be a path of G, and its label sequence is $l_0 \hat{} l_1 \hat{} ... \hat{} l_n$.
- for the label sequence of the form $l_0 \hat{} l_1 \hat{} ... \hat{} l_m$ of the path ρ, correspondingly there is an action sequence $a_0 \hat{} a_1 \hat{} ... \hat{} a_m$. As being mentioned before, for those shared action a_i;, replacing them with a pair of output and input actions ($a!$, $a?$), then again we get an action sequence $a_0 \hat{} a_1 \hat{} ... \hat{} a_s$ $(s \ge m)$.

Actually, the reachability graph of compatible IAN is the deadlock-free behavior graph of its corresponding composed system. And composed system's behavior derivation is just based on this graph. Figure 4 shows the corresponding reachability graph of our working example.

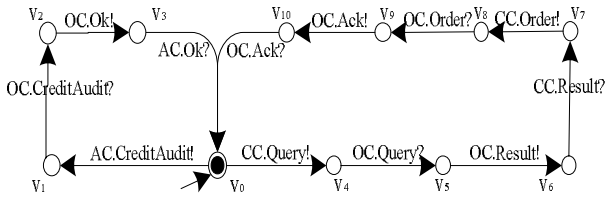

Fig.4. Reachability graph of compatible IAN

5. LTL-based Coordination-Policy Specification

Coordination-policies model the desired interactions of components, which is usually used to ensure specified composed system properties. The behaviors of composed system are given in terms of synchronized actions sequences performed by components interactions in IAN model. Therefore, in specifying coordination-policy we have to distinguish a synchronized action a performed by component C_i with the one performed by its corresponding interactive component C_j ($i \neq j$). Thus, the coordination-polices should be specified in terms of input/output actions of respective 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 $f_i(a) = C_i.a$ for all $a \in A_i$ and A_i is the actions set for C_i.

By referring to usual model checking approach [6], we specify every coordination-policy through temporal logic formalism and choose LTL as specification language. We define $AP = \{\gamma: \gamma=l \vee \gamma=\neg l$, with $l \in C_i[f_i]$, $i=1,...,n\}$ as the set of atomic proposition on which we define our coordination-policies as LTL formula.

6. Coordination-Policy Based Behavior Derivation

6.1 IAN and Coordination-Policy's Semantic Unification

Note that, the semantics of a LTL formula can be defined with respect to a model represented by a Kripke structure [6]. Therefore, in order to unify the underlying semantic of IAN and LTL-based coordination-policy, we consider the Kripke structure corresponding to IAN as K_{IAN}, which is defined as follows:

Definition 6.1 (Kripke structure of compatible IAN). Let G (V, T, LA) be the reachability graph of a compatible IAN. We define the Kripke Structure of the compatible IAN as $K_{IAN} = (V, T, \{v_0\}, LV)$ where $V=V$, $T=T$, $LV=2^{LA}$ with $LV(v_i) = \{a_i: LA((v_{i-1}, v_i)) = a_i, (v_{i-1}, v_i) \in T\}$. For each $v \in V$, $LV(v)$ is interpreted as the set of atomic propositions which are *true* in state v.

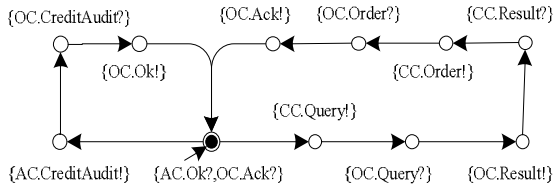

Fig.5. Kripke structure of compatible IAN

Figure 5 shows the Kripke structure of our working example's compatible IAN model. The node with an incoming arrow is the initial state.

In Section 5, we have described how to specify a coordination-policy in LTL, e.g. P, in terms of the desired behaviors of the composed system. Then we have also defined the Kripke structure of compatible IAN.

Definition 6.2 (Büchi Automaton). A Büchi Automaton B is a 5-tuple $<S,A,\Delta,q_0,F>$, where S is a finite set of states, A is a set of actions, $\Delta \subseteq S \times A \times S$ 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 $(q_i, a_i, q_{i+1}) \in \Delta$, $\forall i \geq 0$. 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 [6], a Kripke structure M corresponds to a Büchi automaton B_M. Therefore, based on the algorithms given in [6], we can construct the corresponding Büchi automaton of IAN and LTL-based coordination-policy respectively, e.g., B_{IAN} and B_P (for our working example, B_P is given in Figure 6, and B_{IAN} is given in Figure 7). And in Definition 6.2 shows the definition of Büchi automaton. Let $L(B_{IAN})$ and $L(B_P)$ be the languages consisting of all words accepted by B_{IAN} and B_P, respectively.

Concretely, according to the description of our working example OSS in section 4.1, we specify the coordination-policy of our working example as the following LTL formula:

$$P=\mathbf{F}((CC.Order! \wedge \mathbf{X}(\neg CC.Order! \mathbf{U} AC.CreditAudit!)) \vee (AC.CreditAudit! \wedge \mathbf{X}(\neg AC.CreditAudit! \mathbf{U} CC.Order!)))$$

In this formula, we use three LTL temporal operators: **F**, **X** and **U**. They are the "eventually" or "in the future", "next time" and "until" temporal operators.

This policy specifies the desired behaviors that we want to extract from the online shopping system (OSS). Concretely, it specifies that the Customer Component (CC) and Audit Component (AC) should execute *Order* and *CreditAudit* actions by necessarily using an *alternating-coordination-policy*. In other words, it means that if the Customer component performs an *Order* action then Customer component cannot perform it again if Audit Component has not performed a *CreditAudit* action and vice versa.

In Figure 6, we present the corresponding Büchi automaton of our working example's compatible IAN

behavior model. The double-circled states denote accepting states.

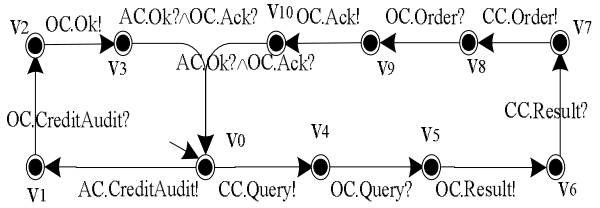

Fig.6. Büchi Automaton of the Compatible IAN

In Figure 7, we translate P into its corresponding Büchi automaton B_P, in which p_0 and p_3 are the initial and accepting state respectively.

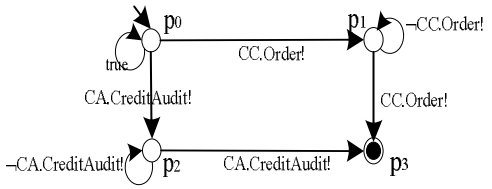

Fig.7. Büchi Automaton of P

Therefore, in our approach, deriving policy-satisfying behaviors all out of composed system can be converted into the construction of cross-product of B_{IAN} and B_P. Let us denote it by $B^{IAN,P}$, which accepts $L(B_{IAN}) \cap L(B_P)$. If $B^{IAN,P}$ is empty then for all possible execution of composed system' compatible behaviors, the coordination-policy P is violated. In this case, it is impossible to derive behaviors that satisfy specified coordination-policy from the composed system. Otherwise, we can derive composed system's coordination-policy satisfying behaviors by obtaining all accepting paths [6] in the $B^{IAN,P}$.

6.2 Policy-satisfying Behavior Derivation

Our approach performs the following procedure from an optimistic point of view to derive coordination-policy satisfying behaviors from composed system's compatible IAN model. Given Büchi automaton $B_{IAN} = (V, \Delta, \{v_0\}, V)$ and $B_P = (S, \Gamma, \{p_0\}, F)$, which are corresponding to composed system's behavior model and user specified coordination-policy respectively.

• Firstly, the procedure constructs the product automaton of B_{IAN} and B_P. The definition of the product automaton is $B^{IAN,P} = (V \times S, \Delta', \{< v_0, p_0 >\}, V \times F)$ where $(<v_i, p_j>, a, <v_m, p_n>) \in \Delta'$ if and only if $(v_i, a, v_m) \in \Delta$ and $(p_j, a, p_n) \in \Gamma$.

• Then, return $B^{IAN,P}$ as the Büchi automaton containing all coordination-policy satisfying execution paths of the composed system. Figure 8(a) shows the cross-product of our working example's compatible IAN and its desired coordination-policy.

• Derive from $B^{IAN,P}$ the corresponding behaviors satisfying given coordination-policy in IAN. And this is constructed by obtaining all accepting paths [6] of the $B^{IAN,P}$. Let $B^{IAN,P} = (V \times S, \Delta', \{<v_0, p_0>\}, V \times F)$ be the automaton that accepts $L(B_{IAN}) \cap L(B_P)$. Here, an accepting path of $B^{IAN,P}$ is a sequence of states $\gamma = s_0, s_1, .., s_n$, $\forall i=0,..,n: s_i \in V \times S$; such that for $0 \le i \le n-1$, $(s_n, s_0) \in \Delta'$ and $(s_i, s_{i+1}) \in \Delta'$, $\exists k=0,..,n: s_k \in V \times F$ (see the bold arrows path in Figure 8).

Noticed that depending on the given coordination-policy, an execution path of $B^{IAN,P}$ may be cyclic. In this case, in order to derive policy-satisfying behaviors, we will not consider the cyclic execution paths without any accepting states.

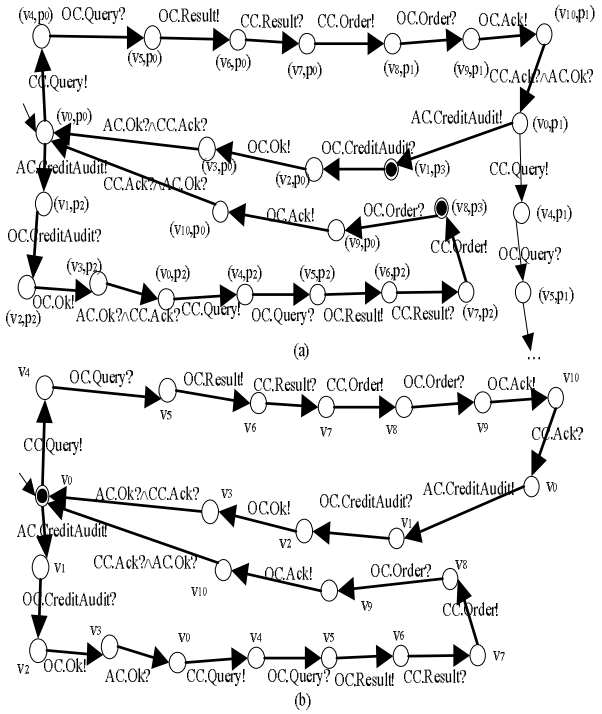

Fig.8. Derivation of P-satisfying behaviors of the compatible IAN

Figure 8 shows how to derive deadlock-free policy-satisfying behavior graph for our working example. In this example, we obtained the deadlock-free policy-satisfying behavior graph for the Online Shopping System (OSS) by only considering the accepting paths (i.e. execution paths which contained accepting states)

of $B^{IAN,P}$, thus the resulting IAN behaviors graph represents all coordination-policy P satisfying behaviors of the OSS (see Figure 8(b)).

6.3 Coordination Environment Construction and Implementation

The principle of the Coordination Environment (CE) is to coordinate components in the composed system to work together in a deadlock-free and policy-satisfying manner. Therefore, in our approach, the behavior graph of CE can be easily constructed by inverting the input and output action of the resulting compatible IAN's behaviors graph derived from the last section. That is to say, if there is an action $C_i.a$ in the derived behaviors of the IAN, accordingly, there is an action $C_i.\overline{a}$ in the CE, where $\overline{a}$ is the corresponding synchronous action of a, e.g. $\overline{a!} = a?$, $\overline{a?} = a!$. Figure 9 illustrates a coordinator-based software architecture (CBA) model [15, 16]. $C_1,..,C_5$ are components. Each line between two components is a connector.

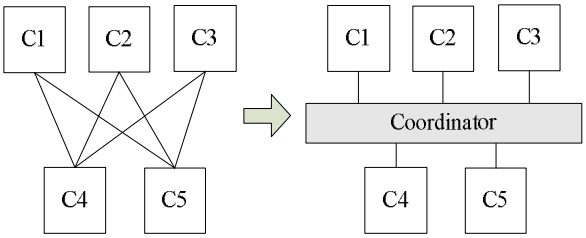

Fig.9. Coordinator-free Architecture vs. Coordinator-based Architecture

Definition 6.3 (Coordinator Based Architecture, CBA). CBA$\equiv$($C_1[f_1] | C_2[f_2] |...| C_n[f_n] |$*Coordinator*) $\backslash \cup_{i=1}^{n} Act_i[f_i]$, where for all $1 \le i \le n$, f_i is a relabelling functions such that $f_i(a) = C_i.a$ for all $a \in Act_i$, and Act_i is the actions set of the behavior graph of C_i, and *Coordinator* is the behavior graph representing the Coordination Environment.

Definition 6.3 shows that CE acts as a coordinator in the composed system, which means components in the composed system will no longer directly synchronize with each other, but through CE. Moreover, CE does not simply synchronize every action between interactive components, while it selectively synchronizes actions according to its derived behavior graph (which is composed of inverted input/output actions of the anticipated synchronized actions of corresponding interactive components). So CE plays a key role in restricting composed system's interactions within its deadlock-free and policy-satisfying behaviors.

Used a method similar to [12], we can automatically derive the code that implements the deadlock-free policy-satisfying coordination environment (i.e. the coordinator component), by visiting the aforementioned behavior graph and by exploiting the information stored in its states and transitions. As an example, we can successfully generate a Coordinator component for our Online Shopping System, which can be used to ensure the *payment security* of the composed system.

```
import orderprocess.idl; ... library Coordinator_Lib {
...
coclass Coordinator {
    [default] interface OrderProcess;
}}
...
class Coordinator : public OrderProcess {
    // stores the current state of the coordinator
    private static int sLabel;
    // stores the current state of the coordination-policy automaton
    private static int pState;
    // stores the number of clients
    private static int clientsCounter = 0;
    // channel's number
    private int channelId;
    // COM smart pointer is a reference to the
    // Order Process Component server object
    private static OrderProcess* orderprocessObj;
    ...
    Coordinator () {// the constructor
          sLabel = 0;
          pState = 0;
          clientsCounter++;
          channelId = clientsCounter;
          orderprocessObj = new OrderProcess();
          ...
    }
// implemented method
...
HRESULT order (/* params list of order */) {
  if(pState == 0) {
      if((channelId == 1) && (sLabel == 7)) {
          pState = 1; sLabel = 8; //it goes on the state preceding the next
                                  //request of a method order
          return orderprocessObj-> order (/* params list */);
      }
  }
  else if (pState == 2 {
       if((channelId == 2) && (sLabel== 7)) {
           pState = 3; sLabel = 8;
           return orderprocessObj-> order (/* params list */);
           pState = 0;//since it has found an accepting stop node,
                      //pState returns to its initial state
       }
       }
  return E_HANDLE;
  }}
  ... }
```

Fig.10. Implementation of the Coordinator Component (Partial)

In Figure 10, we show fragments of our Coordinator's IDL definition, COM library and COM class respectively. The Coordinator component implements the COM interface *OrderProcess* of the Order Process component by defining a COM class *Coordinator*, and by implementing a wrapping mechanism to wrap the requests that Audit and Customer component perform on Order Process Component. *orderPrcessObj* is an instance of the inner COM server corresponding to Order Process

Component and encapsulated into Coordinator component. And we also show the deadlock-free policy-satisfying code implementing the *Order* method of the Coordinator in Figure 10.

7. Conclusions and Future Work

In this paper, we use interface automata and its network to model the behavior of component and its composed system, and also present an approach to deriving policy-satisfying behavior all out from such component-based composed system. The main idea of our approach is to construct a Coordination Environment such that system components can work together in a deadlock-free and policy-satisfying manner, and so as to achieve better composed system properties, e.g., Safety , Liveness, and Fairness, etc.

Coordination-policy based composed system's behavior derivation can firstly increase the reusability of components in CBSD. Secondly, the approach can be used automatically in constructing environment for composed system, to avoid system deadlock and achieve better system properties. And these two aforementioned factors undoubtedly increase the productivity and quality of nowadays component-based information system's development.

In this paper, we have only discussed the problem of how to derive deadlock-free and policy-satisfying behavior all out from composed system. But in practice, during behavior derivation, there is usually more than one coordination-policy available for different system property prospects. So in the future, we are to extend our work for extracting user desired policy-satisfying behaviors in the presence of multiple system property objectives.

8. Acknowledgments

This work is supported by the National High Technology Development 863 Program of China under Grant No.2006AA01Z189, the National Natural Science Foundation of China under Grant No.60473061, the National Basic Research Program of China (973 Program) granted No.2006CB3003002.

9. References

[1] Hu J, Yu XF, Zhang Y,etc. Checking component-based designs for scenario-based specification. *Chinese Journal of Computers*, 2006, 29(4): 513-525 (in Chinese, with English abstract).

[2] de Alfaro, L. and T. A. Henzinger, Interface Automata, in: *Proceedings of 9th Annual ACM Symposium on Foundations of Software Engineering (FSE 2001), ACM* Press, New York (2001), 109-120.

[3] Szyperski, C. Component software: beyond object-oriented programming. ACM Press and Addison Wesley, New York, USA (1998).

[4] Heineman, G. T. and W. T. Councill, "Componet-Based Software Engineering: Putting the Pieces Together", Addison-Wesleg, Boston, 2001.

[5] 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, July 1997.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. The MIT Press, 2001.

[7] 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.

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

[9] 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.

[10] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45-54, 2005.

[11] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Trans. On Programming Languages and Systems*, 19(2):292–333, March 1997.

[12] P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *Elsevier Journal of Systems and Software Special Issue Component-based Software Engineering* - Journal No.: 7735 - Vol.: 65 - 3 - Pages: 173 - 183 - Article No.: 7346, 2003.

[13] 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.

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

[15] M. Tivoli and M. Autili. SYNTHESIS: a Tool for Synthesizing Correct and Protocol-Enhanced Adaptors. *L'Objet journal. Special Issue on Software Adaptation.* Volume 12, Number 1, pp. 77-103, 2006.

[16] M. Autili, P. Inverardi, M. Tivoli and D. Garlan. Synthesis of "correct" adaptors for protocol enhancement in component based systems. In *Proceedings of Specification and Verification of Component-Based Systems (SAVCBS'04) Workshop* at FSE 2004. pp.79-86, 2004.