Correct Components Assembly for a Product Data Management Cooperative System

Massimo Tivoli[1], Paola Inverardi[1], Valentina Presutti[2], Alessandro Forghieri[3], and Maurizio Sebastianis[3]

[1] University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila
{inverard,tivoli}@di.univaq.it
fax: +390862433057
[2] University of Bologna
Dip. Informatica
Mura Anteo Zamboni 7, 40127 Bologna
presutti@cs.unibo.it
fax: +390512094510
[3] Think3 company
via Ronzani 7/29, 40033 Bologna
{alessandro.forghieri,maurizio.sebastianis}@think3.com
fax: +39051597120

Abstract. In this paper we report on a case study of correct automatic assembly of software components. We show the application of our tool (called *Synthesis*) for correct components assembly to a software system in the area of CSCW (*Computer Supported Cooperative Work*). More specifically we consider a product data management (*PDM*) cooperative system which has been developed by the company *Think3* in Bologna, ITALY (*www.think3.com*). In the area of CSCW, the automatic enforcing of desired interactions among the components forming the system requires the ability to properly manage the dynamic interactions of the components. Moreover once a customer acquires a CSCW system, the vendor of the CSCW system has to spend many further resources in order to integrate the CSCW system with the client applications used by the customer organization. Thus the full automation of the phase of integration code development has a great influence for a good setting of a CSCW system on the market. We present the application of our approach and we describe our experience in automatic derivation of the code which integrates the components forming the PDM cooperative system above mentioned. The case study we treat in this paper represent the first attempt to, successfully, apply *Synthesis* in real-scale contexts.

1 Introduction

Correct automatic assembly of software components is an important issue in CBSE (*Component Based Software Engineering*). Integrating a system with

I. Crnkovic et al. (Eds.): CBSE 2004, LNCS 3054, pp. 84–99, 2004.
© Springer-Verlag Berlin Heidelberg 2004

reusable software components or with COTS (*Commercial-Off-The-Shelf*) components introduces a set of problems. One of the main problems is related to the ability to properly manage the dynamic interactions of the components. In the area of CSCW (*Computer Supported Cooperative Work*) [5,10,9], the management of dynamic interactions of the components forming the application can become very complex in order to prevent and avoid undesired interactions. A CSCW application constitutes an integrated environment formed by one or more CSCW servers and many CSCW clients [10,9]. In general, both servers and clients are heterogeneous components built by different organizations of software development. CSCW servers are black-box components providing the main functionalities concerning a cooperative activity (e.g. repository management functionalities as data check-in and check-out, data persistence, concurrent accesses to data, group-aware information management, etc.). CSCW clients are black-box and third-party components which exploit the services provided by the CSCW servers in order to execute a group-aware task. Typically, the servers are sold by the vendor of the CSCW framework. The clients are used by the customer organization of the CSCW framework which is the organization acquiring the CSCW framework on the market. A very important issue concerns the integration between the CSCW servers and CSCW clients. Depending on the customer organization and on the typology of its product manufacture, a CSCW client can be a text editor or a spreadsheet or a computer aided design (*CAD*) application. Given the huge diversity of applications that could be clients of a CSCW server, it is worthwhile noticing that a CSCW server has to be integrated with a CSCW client by following ad-hoc strategies. This means that once the customer acquires the servers forming the CSCW framework, the vendor will have to implement the code integrating the clients with the servers. Moreover the vendor will have to repeat this heavy phase of deployment of the CSCW framework for each different customer. Thus an issue of great influence on the good setting of the vendor on the market concerns the full automation of the phase of integration code development.

Our approach to the integration problem in the CBSE setting is to compose systems by assuming a well defined architectural style [8,6,11,12] in such a way that it is possible to detect and to fix integration anomalies. Moreover we assume that a high level specification of the desired assembled system is available and that a precise definition of the coordination properties to satisfy exists. With these assumptions we are able to automatically derive the assembly code for a set of components so that, if possible, a properties-satisfying system is obtained (i.e. the integration 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 coordination (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 di-

rectly 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 [2] 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 [7] and generic coordination policies of the components interaction behavior [8].

In this paper, by exploiting our approach to correct and automatic components assembly [8,6,11,12], we describe our experience in automatic derivation of the assembly code for a set of components forming a product data management (*PDM*) cooperative system. The PDM system we refer to has been developed by *Think3* company [1] in Bologna, ITALY.

The paper is organized as follows. Section 2 briefly describes our tool for correct and automatic components assembly called *Synthesis*. *Synthesis* implements our theoretical approach to correct and automatic components assembly presented in [8,6,11,12]. Section 3 describes a realistic application of our tool to the PDM cooperative system *ThinkTeam*. Section 4 concludes and discusses future work.

2 *Synthesis*: A Tool for Correct and Automatic Components Assembly

In this section we describe our tool for correct and automatic components assembly called *Synthesis*. For the purposes of this paper, we briefly recall the theory underlying our approach to correct and automatic components assembly implemented in *Synthesis*. For a formal description of the whole approach we refer to [8].

2.1 The Reference Architectural Style

The architectural style *Synthesis* refers to, 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 types of messages: notifications and requests. A notification is sent downward, while a request is sent upward. A top-domain of a component or of a connector is the set of requests sent upward and of received notifications. Instead a bottom-domain is the set of received requests 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. Since it is always possible to decompose a

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

n-layered CBA system in n single-layered CBA systems, in the following of this paper we will only deal with single layered systems. Refer to [8] for a description of the above cited decomposition.

2.2 Configuration Formalization

Synthesis refers to two different ways of composing 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.* Components and system behaviors are modelled as Labelled Transition Systems (LTS). *Synthesis* derives these LTS descriptions from *"HMSC (High level Message Sequence Charts)"* and *"bMSC (basic Message Sequence Charts)"* [2] specifications of the system to be assembled [8]. This derivation step is performed by applying a suitable version of the translation algorithm from bMSCs and HMSCs to LTS (*Labelled Transition Systems*) presented in [14]. HMSC and bMSC specifications are common practice in real-scale contexts thus LTL can merely be regarded as internal to the *Synthesis* specification language.

2.3 *Synthesis* at Work

Synthesis aims at solving the following problem: *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.*

The CBA system V is obtained by automatically deriving the connector assembling the components forming the CFA system T. This connector coordinates the assembled components interactions by following the behaviors corresponding to every coordination policy in P. In order to automatically derive the code implementing the connector in CBA system, *Synthesis* refers to a specific development platform. This platform is Microsoft COM with ATL [13]. This means that the connector will be derived by generating the ATL code implementing the COM server corresponding to it. *Synthesis* assumes that a specification of the system to be assembled is provided in terms of bMSCs and HMSCs specification. By referring to the COM framework [13], *Synthesis* also assumes that a specification of components interfaces and type libraries is given in terms of Microsoft Interface Definition Language (*MIDL*) and binary type library (*.tlb*) files respectively. Moreover it assumes that a specification of the coordination policies to be enforced exists in terms of Linear-time Temporal Logic (*LTL*) formulas or directly in terms of Büchi automata[2] [4]. With these assumptions *Synthesis* is able to automatically derive the assembly code for the components forming the

[2] A Büchi automata is an operational description of a LTL formula. It represents all the system behaviors satisfying the corresponding formula.

specified software system. This code implements the connector component. *Synthesis* implements the connector component in in such a way that all possible interactions among components only follow the behaviors corresponding to the specified coordination policies.

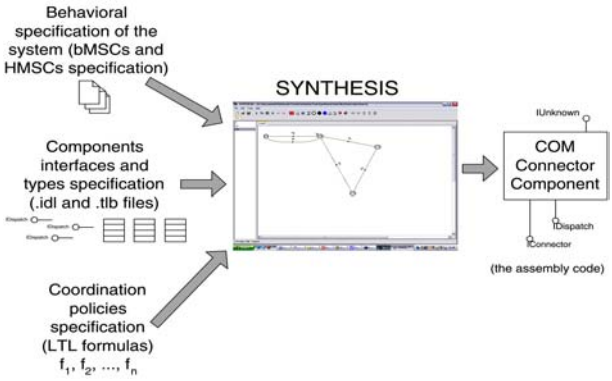

Fig. 1. Input and output data performed by Synthesis tool.

Figure 1 shows the input and output data performed by *Synthesis*.

The method performed by *Synthesis* proceeds in three steps as illustrated in Figure 2.

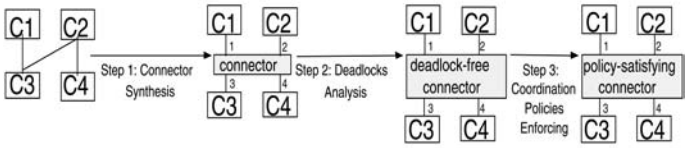

Fig. 2. 3 step method.

The first step builds a connector (i.e. a coordinator) following the CBA style constraints. The second step performs the concurrency conflicts (i.e. deadlocks) detection and recovery process. Finally, by exploiting the usual *automata-based model checking* approach [4], the third step performs the enforcing of the specified coordination policies against the model for the conflict-free connector and then synthesizes the model of the coordination policy-satisfying connector. From the latter we can derive the code implementing the coordinator component which is by construction correct with respect to the coordination policies.

Note that although in principle we could carry on the three steps together we decided to keep them separate. This has been done to support internal data structures traceability.

3 The PDM System ThinkTeam

In this section, we use our tool described in Section 2 to automatically derive the code integrating components forming a PDM system. The PDM system we consider has been developed by *Think3* company [1] in Bologna, ITALY. This system is called *ThinkTeam*. *ThinkTeam* has been developed by using Microsoft Component Object Model (*COM*) [13] with Active Template Library (*ATL*) in Microsoft Visual Studio development environment. *ThinkTeam* is a PDM solution that provides a solid platform for a successful product life-cycle management implementation. For engineering departments, *ThinkTeam* provides the data and document management capabilities required to manage all product documentation, including 3D models, 2D drawings, specifications, analysis and test results. Multiple users will always have access to updated, released and work in progress product information. Also provided is a changes management solution that enables engineers to interface and communicate with the rest of the organization. *ThinkTeam* is packaged into five modules to provide specific capabilities and solution features. For the purposes of this paper the module we are interested on is the *ThinkTeam client* (*TTClient*) component. This is a stand-alone application that is integrated into a CAD application and Microsoft Office applications and provides features to manage documents, versions, data attributes, and relationships among documents. We are interested in applying our approach in order to automatically derive the integration code assembling the *TTClient* component with the distributed instances of the third-party CAD application. This code is derived to force the composed system to satisfy the coordination policies that will be described later.

3.1 ThinkTeam Architecture

In Figure 3 we show a *ThinkTeam* network.

The following are the interacting components in a *ThinkTeam* network:

– the *TTClient* component which provides general purposes PDM functionalities such as documents, multiple users, and changes management, ver-

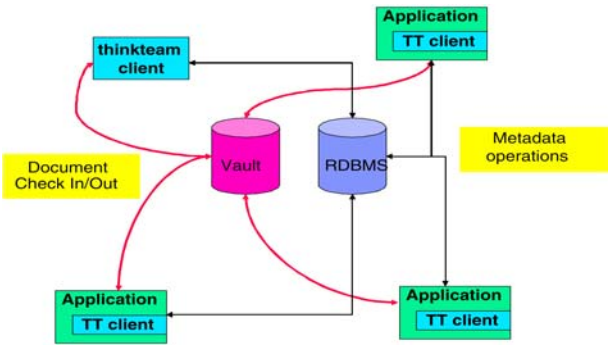

Fig. 3. A ThinkTeam network.

sions controlling, data attributes and relationships among documents management. *ThinkTeam* owns the documents and *metadata* flow and mediates between the applications and the stores. The backing store is the *RDBMS* component;
- the distributed applications used by the *ThinkTeam*'s customer organization. Depending on the kind of customer organization manufacture, these applications can be either a CAD application or a text editor applicationor any other kind of application managing the product data;
- a centralized repository (i.e. the *Vault*) for the documents related to the products of the customer organization and for the relationships among hierarchical documents. A hierarchical document can be either a document with all information itself contained or a document making use of references to other hierarchical documents. *Vault* operations pertain to document data and allow reservation (i.e. check-out), publishing (i.e. check-in) and unreserved read access. The backing store is a filesystem-like entity. Actually an entity corresponds to a file into *Vault*.
- a centralized repository (i.e. the *RDBMS*) for the *metadata*.

In a *ThinkTeam* network, the *TTClient* component manages many types of data related to documents, to the work flow, to product parts and to the organization. All these data are classified in terms of *entity* types and their attributes. Thus an *entity* represents any data which has an associated set of attributes.

3.2 ThinkTeam/Application Integration Schema

As showed in Figure 3, the distributed applications used by the customer share an instance of the *TTClient* component. One of the goals of *Think3* company is to automatically derive the code integrating the *TTClient* component with a particular application which, in our case study, is a CAD system. This is an important goal for the company because an automatic and correct integration of *TTClient* with the customer application makes the *ThinkTeam* system more competitive on the market. In Figure 4, we show the integration schema for *TTClient* component and the CAD system used by the customer organization.

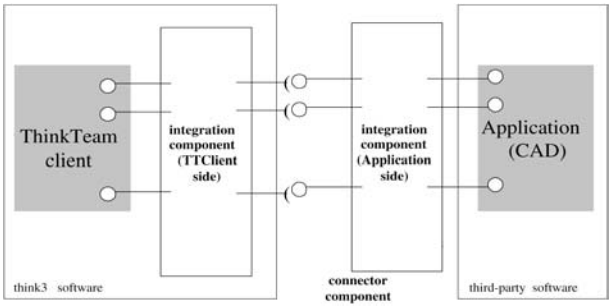

Fig. 4. Integration between ThinkTeam and the customer's application.

By referring to Sections 2.1 and 2.2, the integration schema of Figure 4 represents the CBA version of the system we are considering for the case study of this paper. On the left side we show the component provided by *Think3*. It is the *TTClient* black-box component plus an auxiliary component (i.e. the integration component on the *TTClient* side) which has the only function to export to its clients the services provided by the *TTClient* component. In the following of this paper, we consider this composite component (i.e. *TTClient* plus the auxiliary component) as a single component called *ThinkTeam* component. *ThinkTeam* component is a black-box component which provides to its clients a specified set of services. The following is the subset of all services provided by the *ThinkTeam* component relevant to our purposes: 1) **afterinit**: this method has to be called when the application integrated with the *ThinkTeam* component is completely initialized; 2) **checkout**: locks the specified file into *Vault* for writing operations; 3) **checkin**: releases the lock activated for writing operations on the specified file into *Vault*; 4) **get**: gets a local copy of a file; 5) **import**: copies a local file into *Vault*; 6) **getattrib**: obtains a read only copy of the attributes of an entity into *Vault*; 7) **setvalue**: sets/modifies the value of a certain entity attribute; 8) **setvalues**: set/modify all entity attributes values; 9) **remove**: removes the entity from *Vault*; 10) **start**: starts-up the integration between *ThinkTeam* and the application used by the customer; 11) **stop**: shuts-down the integration between *ThinkTeam* and the application used by the customer. On the right side of Figure 4 we show the application used by the customer organization. In our case the customer's application is a CAD system and the *ThinkTeam* component has to be integrated into it. In the following of this paper, we refer to the customer's application as *CAD* component. The *CAD* component is a black-box component which provides to its clients the following services: 1) **ttready**: this method has to be called when the *ThinkTeam* component integrated into the customer's application is completely initialized; 2) **openfile**: opens a file; 3) **save**: saves the changes made on a file; 4) **closefile**: closes a file. Between *ThinkTeam* and *CAD* components we show the connector component whose aim is to integrate the *ThinkTeam* component and the *CAD* component. The connector mediates the interaction between *ThinkTeam* and the distributed instances of *CAD* by following specified coordination policies. We use *Synthesis* to automatically derive the code implementing the connector component. As described in Section 2.3, we start from the following input data: i) a bMSCs and HMSCs specification of the CFA version of the system to be assembled; ii) *MIDL* and *Type Libraries* files for *ThinkTeam* and *CAD* components; iii) a Büchi automata specification of the desired coordination policies. In Section 3.3, we show the application of *Synthesis* to the automatic and correct integration of the *ThinkTeam* and *CAD* components.

3.3 *Synthesis* for Integrating ThinkTeam and CAD

In this section we apply our tool *Synthesis* in order to automatically derive the code of the connector component showed in Figure 4. This code is derived in order to limit all possible interactions among *ThinkTeam* and *CAD* to a subset of

interactions corresponding to a set of specified and desired coordination policies. In Figure 5 we show the bMSCs representing the execution scenarios of the composed system formed by *ThinkTeam* (i.e. *TT* in Figure 5) and the *CAD* components.

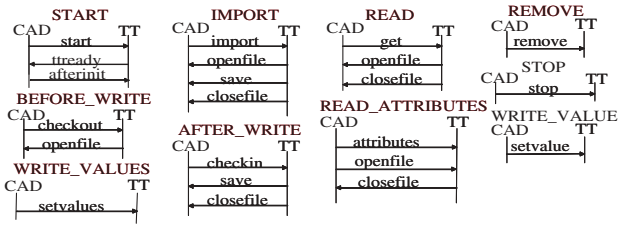

Fig. 5. bMSCs specification for the CFA version of ThinkTeam/CAD system.

The scenarios in Figure 5 are defined in terms of the messages exchanged between *ThinkTeam* and *CAD*. These messages correspond to the methods provided by *ThinkTeam* and *CAD* components. By referring to the methods definitions listed in Section 3.2, the scenarios of Figure 5 do not need further explanations. In Figure 6 we show the HMSCs specification of the composed system formed by *ThinkTeam* and the *CAD* components.

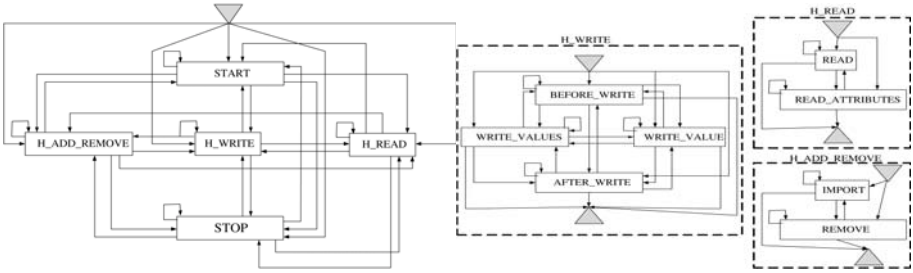

Fig. 6. HMSC specification for the CFA version of ThinkTeam/CAD system.

Informally, a HMSC is a graph where each node, except two special nodes (i.e. the starting and the final node), is a reference to a bMSC or to a sub-HMSC. An arc from a node n_1 to a node n_2 represents that the execution of the scenario corresponding to n_2 follows the execution of the scenario corresponding to n_1. The starting node is the grey triangle with the downward arrow. Conversely, the final node is the grey triangle with the upward arrow. An arc into a HMSC from a bMSC b_i to a sub-HMSC h_j, means that the system's execution goes from b_i to all bMSCs b_k^j reachable in one step from the starting node of h_j. An arc into a HMSC from a sub-HMSC h_j to a bMSC b_i means that the system's execution goes from all bMSCs b_k^j reaching in one step the final node of h_j to b_i. The HMSC of Figure 6 is defined in terms of three sub-HMSCs (i.e. H_WRITE, H_READ and H_ADD_REMOVE). In all HMSCs we have showed in Figure 6, each bMSC

is reachable from every other bMSC into the HMSC. For the sake of brevity, we do not show the MIDL files for *ThinkTeam* and the *CAD* components. This is not a limitation for the understanding of the paper. Actually by referring to Section 3.2, we can consider as MIDL files for *ThinkTeam* and *CAD* the two set of services provided by *ThinkTeam* and *CAD* respectively. The *.tlb* files for *ThinkTeam* and *CAD* are binary files and they are used internally to *Synthesis*. In addition to the bMSCs and HMSCs specifiation plus the MIDL and *.tlb* files, *Synthesis* needs to know how many instances of *ThinkTeam* and *CAD* components have to be considered. This information is provided by interacting with a dialog control of the *Synthesis*'s user interface. In our case study we consider two instances of the *CAD* component sharing an instance of the *ThinkTeam* component. From this additional information (i.e. components instances) and from the two input data considered above (i.e. i) bMSCs and HMSCs specification and ii) MIDL + *.tlb* files), *Synthesis* is able to automatically derive a graph representing the component's behavior for each component's instance forming the specified composed system. This graph is called AC-Graph. In order to automatically derive these AC-Graphs from the bMSCs and HMSCs specification, *Synthesis* executes our implementation of the algorithm developed in [14]. Figure 7 is a screen-shot of *Synthesis* in which we show the automatically derived AC-Graph for the instance of the *ThinkTeam* component.

Refer to [8] for a formal definition of AC-Graph. Informally, an AC-Graph describes the behavior of a component instance in terms of the messages (seen as input and output actions) exchanged with its environment (i.e. all the others components instances in parallel). Each node is a state of the behavior of the component's instance. The node with the incoming arrow (i.e. S1721 in Figure 7) is the starting state. An arc from a node n_1 to a node n_2 is a transition from n_1 to n_2. The transitions labels prefixed by **"!"** denote output actions, while the transitions labels prefixed by **"?"** denote input actions. For the sake of brevity we do not show the AC-Graph for the two instances of the *CAD* component (i.e. C1 and C2432 on the left panel of the *Syntheisis*'s user interface in Figure 7). The last input data for *Synthesis* is the Büchi automata specification of the coordination policies to be enforced on the composed system through the automatically synthesized connector component. The coordination policy we want to enforce in our case study is the following: *"a document cannot be removed if someone has checked it out. Moreover, the attributes cannot be modified if someone is getting a copy of the entity as a reference model."*. Figure 8 is a screen-shot of *Synthesis* in which we show the provided automaton for the above coordination policy.

Informally, the automaton in Figure 8 describes a set of desired behaviors for the composed system formed by the *ThinkTeam*'s instance and the two *CAD*'s instances in parallel under the point of view of a hypothetical observer. Each node is a state of the observed composed system. The node with the incoming arrow is the initial state. The black nodes are the accepting states. Once the automaton execution reaches an accepting state, it restarts from the initial state.

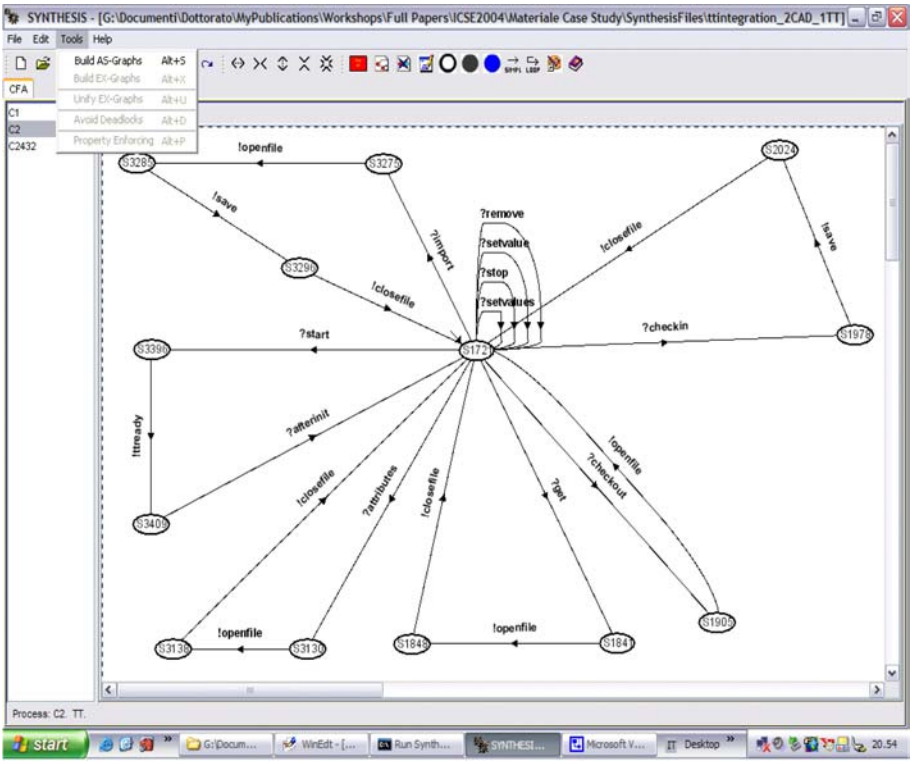

Fig. 7. *Synthesis* screen-shot of an instance of *ThinkTeam* component.

Input[3] transition labels are prefixed by **"?"**, instead output[4] transition labels are prefixed by **"!"**. Each transition label (except for a particular kind of transition) is postfixed by "_" followed by a number. This number is an identifier for a component's instance. Referring to Figure 8, 1 identifies an instance of the *CAD* component, 2432 identifies the other instance of the *CAD* component and 2 identifies the instance of the *ThinkTeam* component. For each state s_i, we can label the outgoing transitions from s_i with three kinds of action: i) a real action (e.g. **?checkout_1** from the initial state in Figure 8) which represents the action itself, ii) a universal action (e.g. **?true_** from the initial state in Figure 8) which represents any action different from all actions associated to the other outgoing transitions from s_i and iii) a negative action (e.g. **?-remove_2432** from the "S11765" state in Figure 8) which represents any action different from the real action corresponding to the negative action itself (i.e. the negative action without "-") and from all actions associated to the others outgoing transitions from s_i. From the AC-Graph for each component and from the automaton of the desired coordination policy, *Synthesis* derives the model of the connector component

[3] Input for the hypothetical observer.
[4] Output for the hypothetical observer.

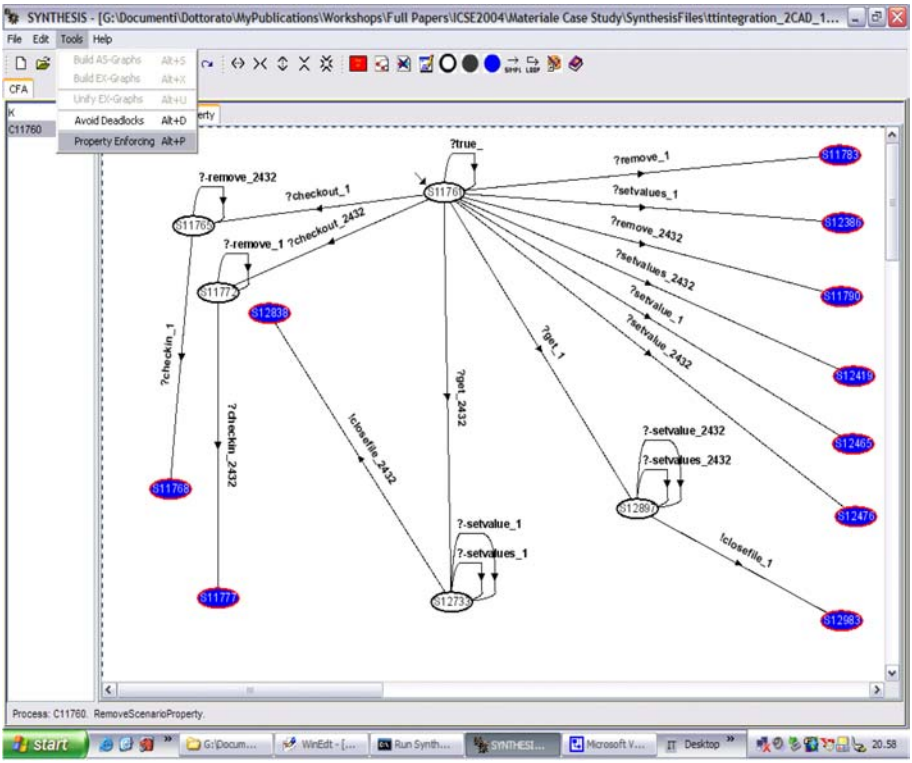

Fig. 8. *Synthesis* screen-shot of the coordination policy to be enforced.

that assembles *ThinkTeam* and the two *CADs* by implementing the coordination policy of Figure 8. For the sake of brevity we do not show the model of the connector synthesized before the coordination policy enforcing step. Figure 9 is a screen-shot of *Synthesis* in which we show the model of the connector after the coordination policy enforcing step.

The sink black states identify the achievement of a desired behavior. Once the connector component's execution achieves a desired behavior, it restarts from the initial state. From this model by exploiting the information stored in each node and arc, *Synthesis* automatically derives the code implementing the policy-satisfying connector. This code implements an ATL COM component. It is constituted by a MIDL file (.idl), an ATL header file (.h) and an ATL source file (.cpp). In order to produce the code implementing the COM connector component *Synthesis* uses also the MIDL and *.tlb* files for *ThinkTeam* and *CAD* provided in input. In the following we only report the meaningful parts of the implementing code of the ATL COM connector component produced by *Synthesis*. For the sake of brevity we do not report the MIDL code (.idl) specifying the interface of the ATL COM connector component. That interface is defined by exploiting the interfaces of *ThinkTeam* and *CAD* components. The following is the ATL header file (.h):

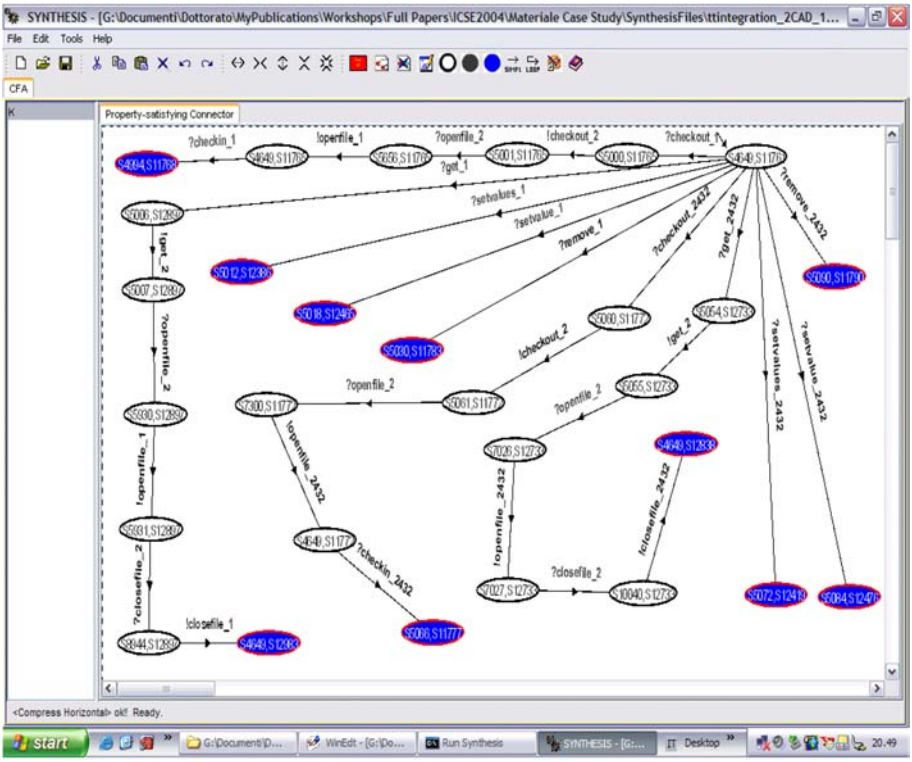

Fig. 9. *Synthesis* screen-shot of the model of the coordination policy satisfying connector.

```
#import "TTIntegrator.tlb"
#import "CAD.tlb"
...
class ATL_NO_VTABLE TTConnector : ... {
public:
    TTConnector()
    {
        sLbl = S4640_S11760;
        clientsCounter++;
        chId = clientsCounter;
        ttObj = new ThinkTeam();
        cadObj = new CAD();
    }
...
public:
    //implemented methods
    STDMETHOD(get)(...); STDMETHOD(checkout)(...); STDMETHOD(checkin)(...); STDMETHOD(afterinit)(...); STDMETHOD(import)(...);
    STDMETHOD(getattrib)(...); STDMETHOD(setvalue)(...); STDMETHOD(setvalues)(...); STDMETHOD(remove)(...); STDMETHOD(start)(...);
    STDMETHOD(stop)(...); STDMETHOD(ttready)(...); STDMETHOD(openfile)(...); STDMETHOD(save)(...); STDMETHOD(closefile)(...);
private:
    //stores the current state of the connector
    static int sLbl;
    //stores the number of connector clients
    static int clientsCounter;
    //channel number of a client; it works as connector clients identifier
    int chId;
    //COM smart pointers; them are references to the ThinkTeam and CAD object
    static ThinkTeam *ttObj;
    static CAD *cadObj;
};
...
```

The class declaration for the ATL COM connector component exploits the class declarations of the *ThinkTeam* and *CAD* components. The ATL COM

connector component encapsulates references to *ThinkTeam* and *CAD* objects and uses a set of private members in order to identify a caller of a service and to store the state reached during the execution. This is needed in order to reflect the behavior of the connector model in Figure 9. The following is the ATL source file (.cpp):

```
...
STDMETHODIMP TTConnector::get(...) {
    HRESULT res;

    if(sLbl == S4640_S11760)
    {
        if(chId == 1) // it corresponds to an instance of CAD
        {
            res = ttObj->get(...);
            sLbl = S5007_S12997;
            return res;
        }
        else if(chId == 2) // it corresponds to the other instance of CAD
        {
            res = ttObj->get(...);
            sLbl = S5055_S12733;
            return res;
        }
    }

    return E_HANDLE;
}
...
```

For the sake of brevity, we have only reported the code for the **get** connector method. It reflects the structure of the model in Figure 9. All other methods are synthesized analogously to the **get** method. The only difference is that while **get**, **setvalue**, **setvalues**, **remove**, **checkout** and **checkin** contain delegations of the corresponding methods on the *ThinkTeam* object (i.e. **ttObj**), the methods **openfile** and **closefile** contain delegations of the corresponding methods on the *CAD* object (i.e. **cadObj**). All remaining methods (i.e. **afterinit**, **import**, **getattrib**, **start**, **stop**, **ttready** and **save**) are synthesized as simple delegations toward the object (i.e. *ttObj* or *cadObj*) which provides them.

4 Conclusion and Future Work

In this paper we have applied the tool *Synthesis* implementing our connector-based architectural approach to component assembly to integrate a PDM system into a CAD application. *Synthesis* focusses on enforcing 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.

Synthesis requires 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 or directly a Büchi automata specification of the coordination policies to be enforced.

By referring to the case study described in Section 3, the main advantage in applying *Synthesis* is the full automation of the integration phase of *ThinkTeam* into *CAD* application used by the customer organization. This reduces the cost and the effort needed for the *ThinkTeam* component deployment phase making *ThinkTeam* more competitive on the market. The code of the synthesized ATL COM connector component could be not fully optimized. *Think3* can modify by hand the synthesized code in order to apply all the requested optimizations. This is obviously better than write the whole adaptation code by hand.

Limits of the current version of *Synthesis* are: i) *Synthesis* completely centralizes the connector logic and it provides 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 the connector logic is centralized we are working on a new version of *Synthesis* which derives a distributed implementation of the connector component if needed; ii) *Synthesis* assumes that a HMSC and bMSC specification for the system to be assembled is provided. It is interesting to investigate the usage into *Synthesis* of UML2 *Interaction Overview Diagrams* and *Sequence Diagrams* [3] instead of HMSCs and bMSCs respectively. This aspect would improve the applicability in real-scale contexts of the tool; iii) *Synthesis* assumes also an LTL (or directly a Büchi automata) specification for the coordination policy to be enforced. We are currently working also in this area trying to find more user-friendly coordination policy specifications; for example by extending the HMSC and bMSC notations to express more complex system's components interaction behaviors.

Acknowledgements

This work has been supported by Progetto MURST CNR-SP4.

References

1. *ThinkTeam overview - http://www.think3.it/m/en/tt8.4_datasheet_en.pdf.*
2. Itu telecommunication standardisation sector, itu-t reccomendation z.120. message sequence charts. (msc'96). Geneva 1996.
3. Uml2 superstructure final adopted specification. *ptc/03-07-06 document - http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML*, August 2003.
4. O. G. Edmund M. Clarke, Jr. and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, London, England, 2001.
5. J. Grudin. Computer-supported cooperative work: History and focus. *In IEEE Computer Journal*, 27(5):19–26, 1994.
6. 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.*

7. 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.
8. P. Inverardi and M. Tivoli. *Software Architecture for Correct Components Assembly. - Chapter In Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture.* Springer, Volume: LNCS 2804, September, 2003.
9. M. Koch. Design issues and model for a distributed multi-user editor. *Computer Supported Cooperative Work, International Journal*, 5(1), 1996.
10. 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.
11. P.Inverardi and M.Tivoli. Automatic failures-free connector synthesis: An example. *in 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.
12. P.Inverardi and M.Tivoli. Deadlock-free software architectures for com/dcom applications. *The Journal of Systems and Software. Journal No. 7735, Vol./Iss. 65/3, pp. 173 - 183*, 2003.
13. D. S. Platt. *Understanding COM+.* Microsoft Press, 1999.
14. S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE*, Vienna, Sep 2001.