SYNTHESIS, a Tool for Synthesizing Correct and Protocol-Enhanced Adaptors

Massimo Tivoli — Marco Autili

University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila
{tivoli,marco.autili}@di.univaq.it

ABSTRACT. *Adaptation of software components is an important issue in Component Based Software Engineering (CBSE). Building a system from reusable or Commercial-Off-The-Shelf (COTS) components introduces a set of problems, mainly related to compatibility and communication aspects. On one hand, components may have incompatible interaction behavior. On the other hand, it might be necessary to enhance the current communication protocol. We address these problems by means of our tool (called SYNTHESIS) which allows one both for preventing incompatible interactions and for enhancing the communication protocol by synthesizing a suitable coordinator. We have validated and applied SYNTHESIS for assembling Microsoft COM/DCOM components.*

RÉSUMÉ. *L'adaptation des composants logiciels est une problématique importante dans le cadre du génie logiciel basé composants (CBSE). La construction de systèmes à partir de composants réutilisables, ou sur l'étagère (COTS), induit plusieurs problèmes, principalement liés aux aspects de compatibilité et de communication. Les composants peuvent avoir des comportements incompatibles d'un point de vue interaction. D'un autre côté, il est parfois nécessaire d'améliorer un protocole de communication. Nous nous intéressons à ces problématiques grâce à notre outil, SYNTHESIS, qui permet à la fois la détection/l'évitement des interactions incompatibles et l'amélioration des protocoles de communication grâce à la synthèse de coordinateurs adéquats. Nous avons validé et appliqué SYNTHESIS à l'assemblage de composants Microsoft COM/DCOM.*

KEYWORDS: *Component Based Software Engineering, Component Adaptation, Adaptors Synthesis, Component Assembly, Protocol Transformation, Protocol Enhancement.*

MOTS-CLÉS : *génie logiciel basé composants (CBSE), adaptation de composants, synthèse d'adaptateurs, assemblage de composants, transformation de protocoles, amélioration de protocoles.*

RSTI - L'objet – 12/2006. WCAT'04, pages 77 to 103

1. Introduction

Adaptation of software components is an important issue in *Component Based Software Engineering* (CBSE). Nowadays, a growing number of systems are built as composition of reusable or *Commercial-Off-The-Shelf* (COTS) components. Building a system from reusable or from COTS (Szyperski 1998) components introduces a set of problems mainly related to communication and compatibility aspects. Often, components may have incompatible interaction behavior. This might require one to restrict the system's behavior to avoid undesired interactions among components. For example, restrict to the set of deadlock-free component interactions or, in general, to a specified set of desired interactions. Moreover, it might be necessary to enhance the current communication protocol. This might require one to augment the system's behavior to introduce more sophisticated interactions among components. These enhancements might be needed to achieve dependability, to add extra-functionality and/or to properly deal with system's architecture updates (*i.e.* components aggregating, inserting, replacing and removing).

We address these problems by means of our tool, called *SYNTHESIS*[1], which allows one both for detecting/avoiding incompatible interactions and for enhancing the communication protocol by synthesizing a suitable coordinator (Inverardi *et al.*, 2003b, Tivoli *et al.*, 2004b, Inverardi *et al.*, 2003a). This coordinator represents an initial glue code. Since, in this paper, we are focusing only on presenting the *SYNTHESIS* tool, we use a simple explanatory example to present it (see Section 5) and we omit approach formalization details which are completely described in (Autili *et al.*, 2004b).

Starting from the specification of the system to be assembled and from the specification of the desired component interactions, *SYNTHESIS* automatically derives the initial glue code for the set of components. This initial glue code is implemented as a coordinator which mediates the cooperation among components by enforcing each desired interaction as reported in (Inverardi *et al.*, 2003b, Tivoli *et al.*, 2004b, Inverardi *et al.*, 2003a). Subsequently, taking into account the specification of possible protocol enhancements, *SYNTHESIS* automatically derives, in a compositional way, the enhanced glue code for the set of components. The enhanced glue code implements a software coordinator which avoids not only incompatible interactions but also provides a protocol-enhanced version of the composed system. More precisely, this enhanced coordinator is defined as composition of new coordinators and new components that are assembled with the initial coordinator in order to enhance its protocol. Each new component represents a wrapper component. A wrapper intercepts the interactions defined by the initial coordinator's protocol in order to apply the specified enhancements without modifying[2] the initial coordinator and the components in the system. The new coordinators are needed to assemble the wrappers with the initial

1. http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.html
2. This is needed to achieve compose-ability in both specifying the enhancements and implementing them.

coordinator and the rest of the components forming the composed system. It is worthwhile noticing that, in this way, we are readily compositional. That is, we can treat the enhanced coordinator as a new *composite* initial coordinator and enforce new desired interactions as well as apply new enhancements. This allows us to perform a protocol enhancement as composition of modular protocol enhancements by improving on the reusability of the synthesized glue code.

We point out that *SYNTHESIS* can be used either to derive the actual code implementing a coordinator or to only derive its behavioral model (*i.e.* state machine). When the final goal of *SYNTHESIS* is to derive the actual code, the underlying approach depends by the particular development platform which is chosen to implement that code. Otherwise the approach does not depend by any particular platform. That is, while keeping the automatic synthesis of the model of a coordinator independent from the automatic synthesis of its actual code, *SYNTHESIS* has to partially refer to one or more particular coordinator development platforms. So far we have validated and applied *SYNTHESIS* for assembling only Microsoft COM/DCOM components. Thus, the code synthesized by *SYNTHESIS* refers to *Microsoft Visual Studio* with *Active Template Library* (ATL) as reference development platform.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces background notions helpful to understand the approach developed by *SYNTHESIS*. Section 4 describes the method implemented by *SYNTHESIS* by distinguishing two main phases. Section 5 presents the *SYNTHESIS* project by using a simple explanatory example. Section 6 discusses future work and concludes.

2. Related work

The approach presented in this paper is related to a number of other approaches that have been considered in the literature. The most closely related work is the scheduler synthesis for discrete event physical systems using supervisory control (Balemi *et al.*, 1993). In those approaches the system allowable executions are specified as a set of traces. The role of the supervisory controller is to interact with the running system in order to cause it to conform to the system specification. This is achieved by restricting behavior so that it is contained within the desired behavior. To do this, the system under control is constrained to perform events only in strict synchronization with a synthesized *supervisor*. The synthesis of a supervisor that restrict the system's behavior resembles one aspect of our approach described in Section 4.1 and in Section 4.2, since we also eliminate certain incompatible interactions through synchronized coordination. However, our approach goes well beyond simple behavioral restriction, also allowing augmented interactions through protocol enhancements.

Recently a reasoning framework that supports modular checking of behavioral properties has been proposed for the compositional analysis of component-based design (de Alfaro *et al.*, 2001, Passerone *et al.*, 2002). In (de Alfaro *et al.*, 2001), an automata-based approach is used to capture both input assumptions about the order in

which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The formalism supports automatic compatibility checks between interface models, where two components are considered to have compatible interfaces if there exists a legal environment that lets them correctly interact. Each legal environment is an adaptor for the two components. However, they provide only a consistency check among component interfaces, but differently from our work do not treat automatic synthesis of *adaptors* of component interfaces. In (Passerone *et al.*, 2002), a game theoretic approach is used for checking whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. This approach is able to automatically synthesize the converter. The idea they develop is the same idea we have developed in our precedent works (Inverardi *et al.*, 2003b, Tivoli *et al.*, 2004b, Inverardi *et al.*, 2003a), that is the restriction of the system's behavior to a subset of safe interactions. Unlike the approach developed by *SYNTHESIS*, they are only able to restrict the system's behavior to a subset of desired interactions and they are not able to augment the system's behavior to introduce more sophisticated interactions among components.

Our research is also related to work in the area of protocol adaptor synthesis (Yellin *et al.*, 1997). The main idea of this approach is to modify the interaction mechanisms that are used to glue components together so that compatibility is achieved. This is done by integrating the interaction protocol into components. However, they are limited to only consider syntactic incompatibilities between the interfaces of components and they do not allow the kind of protocol transformations that our synthesis approach supports. In other words, they act only on the *signature* level while we go beyond the signature level by also acting on the *interaction/communication* level.

In other work in the area of wrapper formalization, it is showed how to use formalized protocol transformations to augment connector behavior (Spitznagel *et al.*, 2003). The key result was the formalization of a useful set of connector protocol enhancements. Each enhancement is obtained by composing wrappers. This approach characterizes wrappers as modular protocol transformations. The basic idea is to use wrappers to enhance the current connector communication protocol by introducing more sophisticated interactions among components. Informally, a wrapper is new code that is interposed between component interfaces and communication mechanisms. The goal is to alter the behavior of a component with respect to the other components in the system, without actually modifying the component or the infrastructure itself. While this approach deals with the problem of enhancing component interactions, unlike our work it does not provide automatic support for composing wrappers, or for automatically eliminating incompatible interaction behaviors.

In other work in the area of component adaptation, it is showed how to automatically generate a concrete adaptor from a specification of components interfaces, a partial specification of the components interaction behavior, a specification of the adaptation in terms of a set of correspondences between actions of different components and a partial specification of the adaptor (Brogi *et al.*, 2004). The key result was the set-

ting of a formal foundation for the adaptation of heterogeneous components that may present mismatching interaction behavior. Although this work provides a fully formal definition of the notion of component adaptor, its application domain is quite different than the our. Since, in specifying a system, we want to maintain a high abstraction level, assuming a specification of the adaptation in terms of a set of correspondences between methods (and their parameters too) of two components requires to know so much implementation details (about the adaptation) that we do not consider in order to synthesize the adaptor.

In other previous work, by one of the authors, we have shown how to apply protocol enhancements by dealing with components that might have incompatible interfaces at signature level (Tivoli *et al.*, 2004a). However the approach described in (Tivoli *et al.*, 2004a) is limited only to consider deadlock-free coordinators.

3. Background

In this section we briefly recall some basic notions related both to a possible application domain (*i.e.* COM/DCOM applications) and to the theory underlying the approach implemented in *SYNTHESIS*.

3.1. *COM/DCOM overview*

A COM/DCOM application is based on a client/server architecture. A server is a component providing services to its clients. A COM server is represented by an object library called *Type Library*. A type library is a container for a set of objects defined by a server component. Each object into the library is an instance of a COM class defined through the code implementing the server. A client of a component is any program that can execute the component code. To allow clients to access its functions, a COM/DCOM server implements one or more interfaces by defining a COM class for each interface. COM/DCOM supports multiple interfaces of a component. An interface is a binary structure in the address space of a component, whose layout is defined as a table of pointers to functions. A client refers to an interface's pointer. The specification of all functions that can be called through an interface is defined by the type of that interface. A type may inherit from another type by extending its list of function specifications. All interface types are organized in a single inheritance hierarchy with a special type *IUnknown* as root. Typically, COM/DCOM does take Microsoft version of the DCE/IDL (MIDL) as preferred notation to define an interface type. The DCE/IDL and MIDL are two different languages. MIDL is a Microsoft's extension of the standard DCE/IDL. A client can invoke a server method using information contained in the *Type Library*. A *Type Library* specification is defined by MIDL code. A COM client has all the information to use a COM server after the MIDL compiler has processed the *Type Library* specification and the marshaling/unmarshaling code defined in the server MIDL file. A *Type Library* is best thought of as a binary version of a MIDL file. It contains a binary description of the interfaces exposed by a

component, defining the methods along with their parameters and return types. COM defines two approaches to use existing components as building blocks of new ones: **containment/delegation** and **aggregation**. For the purposes of this paper, since the **aggregation** is not a general composition mechanism (Jackson *et al.*, 2000, Sullivan *et al.*, 1999, Sullivan *et al.*, 1997), we are interested only in the **containment/delegation** composition mechanism. That is, an outer component encapsulates one or more inner components and uses their services. In order to make a client able to uniquely locate a server, COM/DCOM associates a Globally Unique[3] IDentifier (GUID) to each component's interface, class and type library. In C++, there are data types defined by COM header files for GUID, CLSID (COM class identifier GUID), and IID (COM interface identifier GUID).

3.2. *COM/DCOM and the Windows registry*

All the information that the COM/DCOM runtime (*i.e.*,`ole32.dll`) needs to uniquely locate and execute components is stored in the *"Windows"* registry. This information takes into account the location of a COM/DCOM component, its type library, its interfaces and its execution and access rights. The *"Windows"* registry has a hierarchical structure which is a tree structure. It has a root node (labeled with "My Computer") and a set of sub-trees called *"hives"*. Each *"hive"* contains a lot of keys. Each key may contain one or more sub-keys, and so on. A key may have multiple values and each value has assigned a name and a data.

A very interesting aspect for our purposes is that, by simply acting on the registry, COM/DCOM provides an easy way to interpose a component between others interacting components. For example if we have one client C interacting with one server S, we can interpose another server K between C and S in such a way that when C creates an instance of S, the COM/DCOM runtime returns to C a reference to an instance of K (instead of a reference to an instance of S). This mechanism is hidden to C and S. First, we have to create a *TreatAs* key for S under the hive *HKEY_CLASSES_ROOT/CLSID/*$<$*...unique class identifier of* S*...*$>$. Then, we have to set the value of *TreatAs* to the unique class identifier of K. We can do that either by hand, during the deployment phase of K, or automatically (*i.e.* at run-time) by using the COM API function *CoTreatAsClass*. *CoTreatAsClass* takes two unique class identifiers: *clsid1* and *clsid2*. It creates the key *HKEY_CLASSES_ROOT/CLSID/*$<$*clsid1*$>$*/TreatAs* and sets its value to *clsid2*.

3.3. *The reference architectural style*

The starting point for our work is the use of a formal architectural model of the system representing the components to be integrated and the connectors over which the components will communicate (Shaw *et al.*, 1996). To simplify matters we will

3. They are unique across all machines.

consider the special case of a generic layered architecture in which components can request services of components below them, and notify components above them. Specifically, we assume each component has a top and bottom interface. The top (bottom) interface of a component is a set of top (bottom) ports. Connectors between components are synchronous communication channels defining top and bottom ports. Components communicate by passing two types of messages: notifications and requests. A notification is sent downward, while a request is sent upward. We will also distinguish between two kinds of components (i) *functional components* and (ii) *coordinators*. Functional components implement the system's functionality, and are the primary computational constituents of a system (typically implemented as COTS components). Coordinators, on the other hand, simply route messages and each input they receive is strictly followed by a corresponding output. We make this distinction in order to clearly separate components that are responsible for the functional behavior of a system and components that are introduced to aid the integration/communication behavior. Within this architectural style, we will refer to a system as a *Coordinator-Free Architecture* (CFA) if it is defined without any coordinators. Conversely, a system in which coordinators appear is termed a *Coordinator-Based Architecture* (CBA) and is defined as *a set of functional components directly connected to one or more coordinators, through connectors, in a synchronous way*. Refer to (Autili *et al.,* 2004b) for a formal definition of CFA and CBA.

Figure 1. *A sample of a CFA and the corresponding CBA*

Figure 1 illustrates a CFA (left-hand side) and its corresponding CBA (right-hand side). C_1, C_2 and C_3 are functional components; K is a coordinator. The communication channels identified by 1, 2 and 3 are connectors.

3.4. *Configuration formalization*

To specify the behavior of a system we use *High level Message Sequence Charts* (HMSCs) and *basic Message Sequence Charts* (bMSCs) (ITU 1996). From this MSC specification, we can derive a *Labeled Transitions System* (LTS) for each component in the system. The LTS of a component C describes the interaction behavior of C with its environment (*i.e.* the parallel composition of all the other components). This is done by applying the translation algorithm described in (Uchitel *et al.,* 2001). HMSC and bMSC specifications are useful as input language, since they are commonly used in software development practice. Later we will see an example of derivation of LTSs from a bMSC and HMSC specification (Section 5). To define the behavior of a *composition* of components, we simply place in parallel the LTS descriptions of those components, "*hiding*" the actions to force synchronization. This gives a CFA for a set

of components. We can also produce a corresponding CBA for these components with equivalent behavior by automatically deriving and interposing a "*no-op*" coordinator between communicating components. The "*no-op*" coordinator interacts with the other components in the system as a simple delegator of their requests/notifications. That is, it does nothing (at this point), it simply passes events between communicating components (as we will see later the coordinator will play a key role in restricting and augmenting the system's interaction behavior).

4. Method description

In this section we briefly describe the method implemented by *SYNTHESIS*. We can distinguish two main phases. The first phase concerns the deadlock-free restriction of the system's behavior to a specified set of desired interactions. The second one is related to the augmentation of the system's behavior to introduce more sophisticated interactions among components. Refer to (Inverardi *et al.*, 2003b, Inverardi *et al.*, 2004) and to (Autili *et al.*, 2004b) for a detailed description of the first and second phase respectively. However this is not required to grasp the main idea underlying the approach.

4.1. *First phase: initial coordinator synthesis*

The first phase concerns the automatic synthesis of the so-called "*initial*" coordinator. This coordinator represents an initial glue code. It behaves as the "*no-op*" coordinator restricted to the only interactions which are deadlock-free and which reflect the ones specified by means of coordination policies (*i.e.* it performs only the desired interactions). Given a CFA system T for a set of black-box interacting components and a set of coordination policies P, *SYNTHESIS* automatically derives, if it is possible, the corresponding CBA system V which is deadlock-free and which performs only every policy in P. The CBA system V is obtained by interposing the synthesized "*initial*" coordinator between the components forming T.

SYNTHESIS assumes that a specification of the system to be assembled is provided in terms of bMSC and HMSC specification. Moreover, it can take as input a specification of the coordination policies to be enforced in terms of Büchi Automata (Clarke *et al.*, 2001). With these specifications, *SYNTHESIS* is able to automatically derive the assembly code for the components forming the specified system. This code implements the "*initial*" coordinator component. During the first phase, *SYNTHESIS* proceeds in three steps as we have shown in Figure 2.

(1) The first step builds a behavioral model (*i.e.* a LTS) of the *"no-op"* coordinator. Starting from the bMSC and HMSC specification, for each component forming the system to be assembled *SYNTHESIS* derives a set of LTSs. Each set of component LTSs characterizes different aspects of the component dynamics, from the actual component behavior to its assumptions on the environment. More precisely, for each

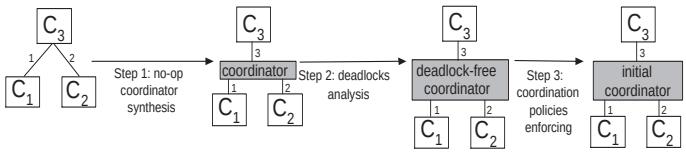

Figure 2. *First phase method*

component instance in the bMSC and HMSC specification, *SYNTHESIS* firstly extracts a LTS describing the component actual behavior in terms of input/output actions exchanged with its environment. We denote it as AC-Graph. Secondly, for each component's AC-Graph *SYNTHESIS* derives a LTS representing the environment expected by the component in order to not block. We denote it as AS-Graph. Finally, for each component's AS-Graph, by following the CBA style constraints, *SYNTHESIS* derives a LTS which represents the behavior that the component expects from the *"no-op"* coordinator in order to not block. We denote it as EX-Graph. Each EX-Graph represents a partial view of the *"no-op"* coordinator behavior. It is partial since it only reflects the expectations of a single component. By means of an EX-Graph unification algorithm, *SYNTHESIS* derives the LTS that models the *"no-op"* coordinator global behavior.

(2) The second step performs the deadlock detection and recovery process against the LTS of the *"no-op"* coordinator. It simply prunes the paths of this LTS ending with a sink node. This provides us with the LTS of the deadlock-free coordinator.

(3) Finally, the third step synthesizes the LTS of the *"initial"* coordinator by enforcing the specified coordination policies against the LTS of the deadlock-free coordinator.

From the LTS of the *"initial"* coordinator *SYNTHESIS* derives the code implementing it which is by construction correct with respect both to deadlock-freeness and 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.

4.2. *Second phase: from the initial coordinator to the protocol-enhanced coordinator*

The second phase of the method implemented in *SYNTHESIS* starts with a deadlock-free CBA system V, which performs only the specified desired behaviors (*i.e.* the coordination policies), and produces the corresponding protocol-enhanced CBA system V'.

Let P be the set of desired behaviors specified within the first phase of our method, given the deadlock-free and P-satisfying[4] CBA system V (automatically obtained after the execution of the first phase), and a set of coordinator protocol enhancements E, if it is possible, *SYNTHESIS* automatically derives the corresponding enhanced, deadlock-free and P-satisfying CBA system V'.

The second phase assumes a specification of E in form of bMSCs and HMSCs. In the following, we discuss the method performed by *SYNTHESIS*, during this phase, proceeding in two steps as illustrated in Figure 3.

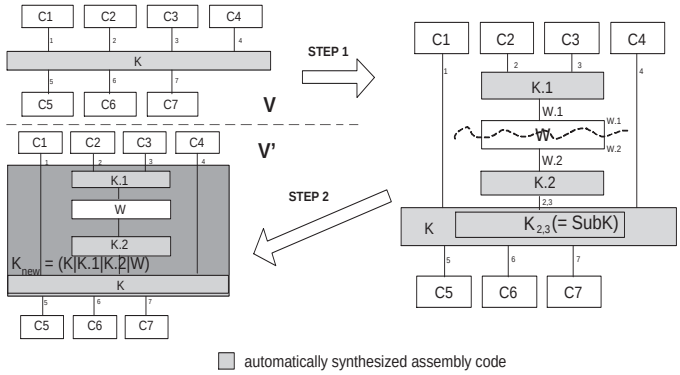

Figure 3. *Second phase method*

(1) In the first step, by taking into account P and V, if it is possible, *SYNTHESIS* applies each protocol enhancement in E. This is done by inserting a wrapper component W between the initial coordinator K (see Figure 3) and the portion of V affected by the specified protocol enhancements (*i.e.* the set of $C2$ and $C3$ components of Figure 3). Note that we do not need to consider the entire model of K but we just consider a *"sub-coordinator"* which represents the portion of K that communicates with $C2$ and $C3$ (*i.e.* the *"sub-coordinator"* $K_{2,3}$ of Figure 3). In general, *SYNTHESIS* denotes this *"sub-coordinator"* as $SubK$. $K_{2,3}$ (*i.e.* $SubK$) represents the *"unchangeable"*[5] environment that K *"offers"* to W. The wrapper W is a component whose interaction behavior is specified in each enhancement of E. Depending on the logic it implements, we can either built it by scratch or acquire it as a pre-existent (COTS) component (e.g., a data translation component). W intercepts the messages exchanged between $K_{2,3}$, $C2$ and $C3$ and implements the enhancements in E by acting on the interaction behaviors performed on the communication channels 2 and 3 (*i.e.* connectors 2 and 3 of Figure 3). We first decouple K (*i.e.* the *"sub-coordinator"* $K_{2,3}$), $C2$ and $C3$ to ensure that they no longer synchronize directly. Then, we automatically derive two behavioral models of W (*i.e.* two AC-Graphs)

4. The P-satisfying system is the one which performs only the desired behaviors in P.
5. Since we want to be readily compositional, our goal is to apply the enhancements without modifying the coordinator and the components.

from the bMSC and HMSC specification of E. These two AC-Graphs are: i) $W.1$ which is the behavior of W only related to the interactions with the components affected by the enhancement and ii) $W.2$ which is the behavior of W only related to the interactions with K. We do this analogously to what we have done, in Section 4.1, to derive the AC-Graph for each component in the CFA system. Finally, if the insertion of W in V will allow the resulting composed system (*i.e.* V' after the execution of the second step) to still satisfy each desired behavior in P, W is interposed between $K_{2,3}$, C_2 and C_3. To insert W, we automatically synthesize two new coordinators $K.1$ and $K.2$. In general, $K.1$ always refers to the coordinator between $W.1$ and the components affected by the enhancement; $K.2$ always refers to the coordinator between K (*i.e.* $SubK$) and $W.2$.

(2) In the second step, we derive the implementation of the synthesized glue code used to insert W in V. This glue code is the actual code implementing $K.2$ and $K.1$. By referring to Figure 3, the parallel composition K_{new} of K, $K.1$, $K.2$ and W represents the enhanced coordinator. By iterating the whole approach, K_{new} may be treated as K with respect to the enforcing of new desired behaviors and the application of new enhancements. This allows us to achieve compose-ability of different coordinator protocol enhancements (*i.e.* modular protocol's transformations). In this way, our approach is compositional in the automatic synthesis of the enhanced glue code.

5. The SYNTHESIS project

In this section, we describe the architecture of our *SYNTHESIS* tool by only looking at its main constituent software modules. For some module we also point out both implementation decisions and justification of these choices. It is worth mentioning that the current version of *SYNTHESIS* should be considered a prototype. We are developing it by following an evolutionary approach to systems development. Thus the current prototypal version of *SYNTHESIS* is still subjected to further extensions. *SYNTHESIS* is implemented using the *Java* language and some of its input and output data are coded using the *XML* meta-language. This is done to achieve both platform-independence and data format portability.

In Figure 4 we show the overall structure of *SYNTHESIS* focusing on the main software modules and their relationships. *SYNTHESIS* can be seen as a container of the above mentioned modules (shown in Figure 4) which defines relationships among them and also provides editing and viewing functionality. Analogously to what we have done in Section 4, in describing the *SYNTHESIS* architecture we distinguish two main phases.

5.1. *First phase*

In Figure 5 we show the input and output data performed by *SYNTHESIS* within the first phase. It is worthwhile noticing that, by means of capital letters, we obtain a

direct mapping between Figure 4 and Figure 5. This mapping allows us to correlate each module with the I/O data it manages, performs or builds.

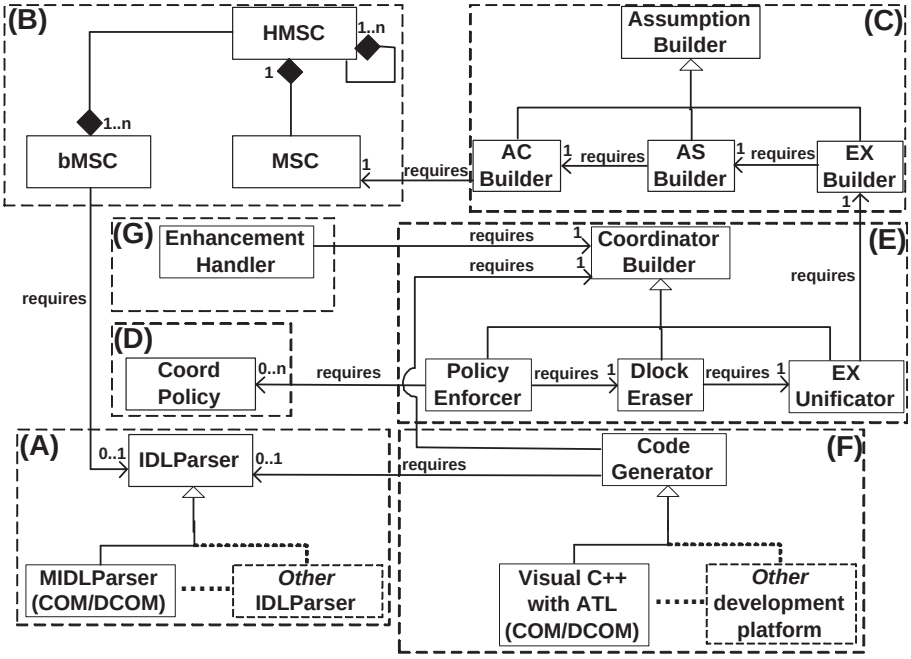

Figure 4. *SYNTHESIS architecture*

In the reminder of this section, by means of an explanatory example, we briefly describe each *SYNTHESIS* module. In doing this we refer to the method described in Section 4.1.

***Module* A:** *component interface parser*

This module contains a superclass (*i.e.* represented by the "IDLParser" entity in Figure 4) that manages a specific data structure which is for storing an abstract representation of an IDL file possibly given as input. That superclass has to be specialized in order to implement a parser of IDL files based on a particular IDL notation (e.g., Microsoft IDL for COM/DCOM, DCE/IDL for CORBA, etc.). In the current version of *SYNTHESIS*, we specialized that class to implement a parser of Microsoft IDL (MIDL) files. We have chosen to support MIDL because so far we have validated *SYNTHESIS* only in the context of COM/DCOM applications.

***Module* B:** *bMSC and HMSC specification of the CFA*

This module is used to specify the CFA system to be assembled in terms of a bMSC and HMSC specification. In doing that, this module exploits an *ad-hoc* library that we have developed to allow creation, validation and manipulation of bMSCs and

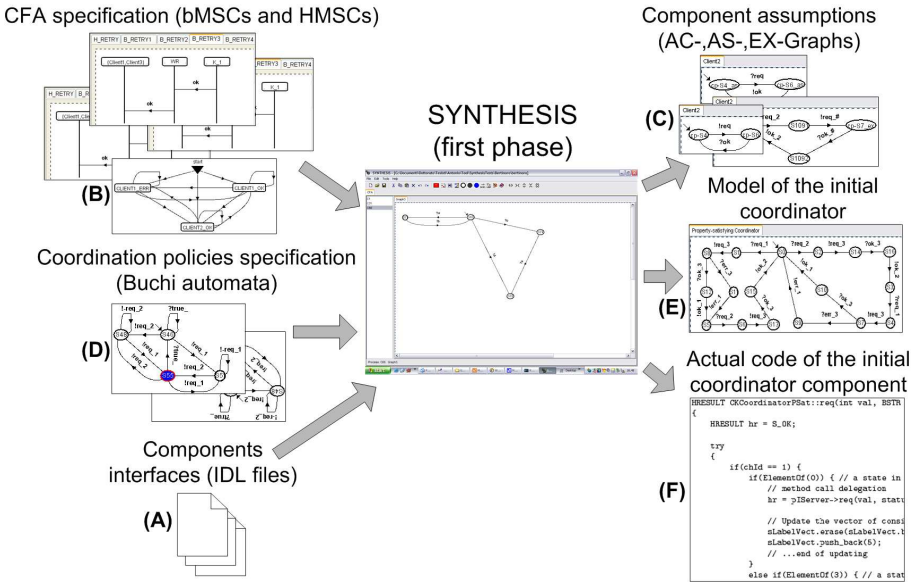

Figure 5. *Input and output data performed by SYNTHESIS within the first phase*

HMSCs coded in *XML*. To check if these *XML* files are valid, an *ad-hoc XML schema* is used. This module requires a suitable implementation of the "IDLParser" entity (see module **A** shown in Figure 4) only when the user's goal is to derive the actual code of the coordinator assembling the specified system. In this case, the parser provides the user with a list of all requests exported by a component. In this way, he/she can directly select those requests while specifying a bMSC.

Now, let us introduce an explanatory example. In this example, we consider a CFA system T formed by three components (*Client1*, *Client2* and *Server*). *Client1* performs a request (*i.e.* the request *req*) and waits for an erroneous or successful notification (*i.e.* the notification *err* and *ok* respectively). *Client2* simply performs the request and it never handles erroneous notifications. *Server* receives a request and then it may answer either with a successful or an erroneous notification[6].

Figure 6 is a screen-shot of *SYNTHESIS* showing the bMSC specification of the CFA system T.

Each bMSC represents a possible execution scenario of T (*i.e.* *CLIENT1_ERR*, *CLIENT1_OK* and *CLIENT2_OK*). Each execution scenario is described in terms of a set of interacting components, sequences of requests and possible corresponding notifications. To each vertical line we associate an instance of a component. Each

6. The error could be either due to an upper-bound on the number of requests that *Server* can accept simultaneously or due to a general transient-fault on the communication channel.

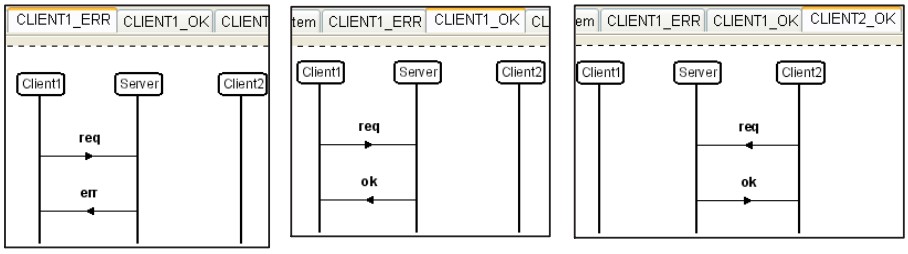

Figure 6. *bMSC specification*

horizontal arrow represents a request or a notification. *CLIENT1_ERR* describes the scenario in which *Client1* performs the request *req* and receives *err* as notification. *CLIENT1_OK* is analogous to *CLIENT1_ERR* but the notification which is *ok*. *CLIENT2_OK* is equal to *CLIENT1_OK* except for *Client2*, which performs the request *req*. As we will explain in the description of the module **F**, for our purpose, *SYNTHESIS* does not require to specify the actual parameters list for each request. Figure 7 is a screen-shot of *SYNTHESIS* showing the HMSC specification of T.

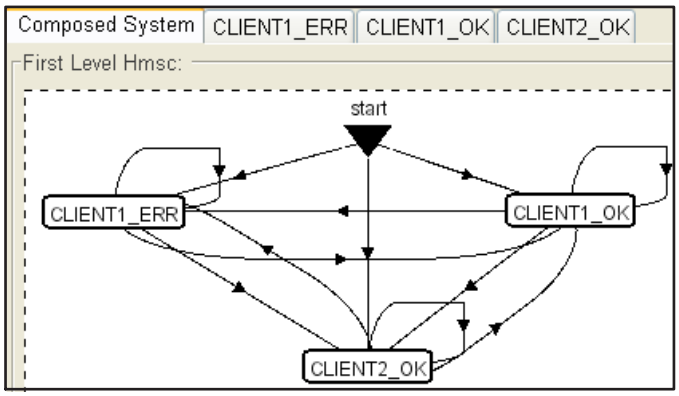

Figure 7. *HMSCs specification of the CFA system*

Each HMSC describes possible continuations from a scenario to another one. It is a graph with two special nodes: a starting and a possible ending node. Each other node is related either to a specified scenario or to a nested HMSC (*i.e.* an HMSC is a composition of either bMSCs or nested HMSCs). An arrow represents a transition from a scenario to another one. In other words, each HMSC composes the possible execution scenarios of T. By referring to Figure 7, from the starting state, the system T can execute one of the three specified scenarios. Then, from each scenario, T can re-execute that scenario or a different one.

***Module* C:** *component assumption generation*

By referring to Figure 4 this module exploits (*i.e.* requires) the module **B**. From the bMSC and HMSC specification of figures 6 and 7, this module outputs the component AC-Graphs. It implements a revisited version of the algorithm described in (Uchitel *et al.*, 2001). In accordance with the bMSC and HMSC data format, these AC-Graphs are coded in *XML* too. In Figure 8 we report *SYNTHESIS*'s screen-shots showing the AC-Graphs derived for *Client1*, *Client2* and *Server*.

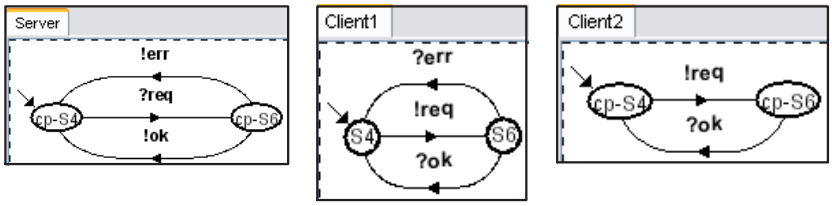

Figure 8. SYNTHESIS*'s screen-shots of* $Server$, $Client1$ *and* $Client2$ *AC-Graphs*

Each node is a state of the component's instance. The node with the incoming arrow 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 transition labels prefixed by **"!"** denote output actions, while the transitions labels prefixed by **"?"** denote input actions.

By referring to Section 4.1, from the component AC-Graphs, this module can also output the component AS- and EX-Graphs (figures 9 and 10 respectively). The only difference between an AC-Graph and an AS-Graph relates to the arc labels which are symmetric since they model the environment as each component expects it.

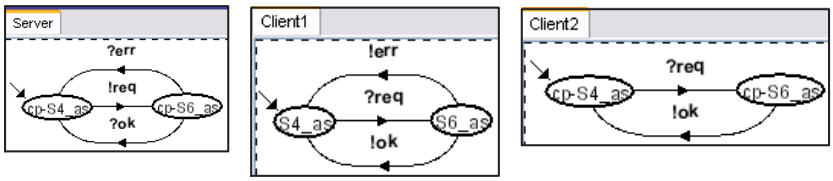

Figure 9. SYNTHESIS*'s screen-shots of* $Server$, $Client1$ *and* $Client2$ *AS-Graphs*

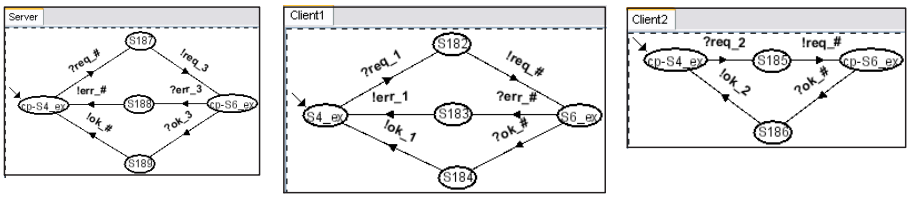

Figure 10. SYNTHESIS*'s screen-shots of* $Server$, $Client1$ *and* $Client2$ *EX-Graphs*

An EX-Graph refines an AS-Graph since it represents the behavior that the component expects from the initial coordinator. We recall that, within the CBA style, the component environment is represented only by the coordinator. The coordinator performs strictly sequential input-output operations only, thus if it receives an input from a component it will then output the received input message to the destination component. Analogously, if the coordinator outputs a message, this means that immediately before it inputs that message. Intuitively, for each transition labeled with $?\alpha$ ($!\alpha$) in the AS-graph, in the corresponding EX-graph there are two strictly sequential transitions labeled $?\alpha\_i$ and $!\alpha\_\#$ ($?\alpha\_\#$ and $!\alpha\_i$), respectively. Let C_i the component for which we are deriving the corresponding EX-Graph; action $?\alpha\_i$ ($!\alpha\_i$) denotes an input (output) action α on the communication channel that connects C_i to the coordinator (*i.e.* the communication channel i). Action $?\alpha\_\#$ ($!\alpha\_\#$) denotes an input (output) action α on a communication channel that connects the coordinator to a component different than C_i; thus this communication channel is unknown for C_i (we denote this unknown channel by using the symbol $\#$).

***Module* D:** *Büchi automata specification of the coordination policies*

This module is used to specify the coordination policies (*i.e.* the desired behavior) for the CFA system T in terms of Büchi Automata. The current implementation of this module codes each Büchi Automaton as a binary object by exploiting *Java* serialization. A corresponding version based on XML is still work in progress. Each Büchi Automaton describes a coordination policy that must be implemented by the coordinator to mediate the interaction among the components. In other words, the coordinator has to be derived in order to force each coordination policy on the interaction behavior of the components forming the CFA system. Figure 11 is a screen-shot of *SYNTHESIS* showing the *"AlternatingProtocol"* policy specified for our explanatory example.

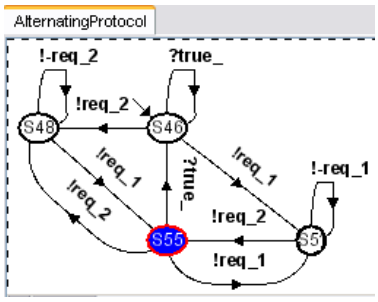

Figure 11. "AlternatingProtocol" *coordination policy*

AlternatingProtocol specifies the behavior of T that guarantees the evolution of all components. It specifies that *Client1* and *Client2* must perform the request *req* by necessarily using an alternating coordination protocol. Each node is a state of T. The node with the incoming arrow is the starting state. The filled nodes are the states accepting the desired behavior. That is, in these states the desired behavior has

been accomplished. The syntax and semantics of the transition labels are the same than those of the AC-Graphs except for the two kinds of actions: i) a universal action (e.g., $?true\_$) which represents any possible action, and ii) a negative action (e.g., $!-req\_2$) which represents any possible action different from the same negative action. Moreover, unlike actions in an AC-Graph, each action has associated an identifier specifying what component performs that action. For example $!-req\_2$, in Figure 11, matches all the actions labeled with a string different from the label $!req\_2$.

***Module* E:** *builder of the initial coordinator model*

This module is responsible for deriving the model of the initial coordinator. In particular, it is specialized by the "EXUnificator", "DlockEraser" and "PolicyEnforcer" entities shown in Figure 4.**E**. These entities respectively implement: (i) the EX-Graph unification algorithm which is for building the LTS of the "*no-op*" coordinator; (ii) the deadlock avoidance algorithm to derive the LTS of the deadlock-free coordinator; (iii) the coordination policy enforcing algorithm that is for obtaining the LTS of the initial coordinator.

In Figure 13 we show the synthesized LTS of the "*no-op*" coordinator for our example.

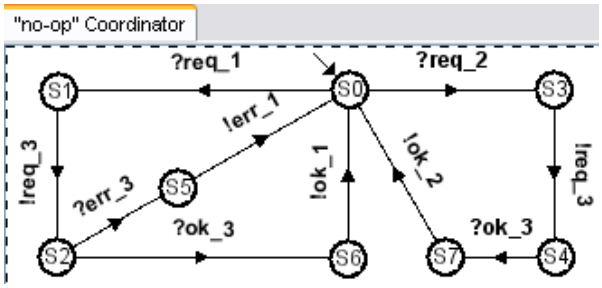

Figure 12. SYNTHESIS*'s screen-shot of the "no-op" coordinator graph*

Each node of the coordinator graph is a state of the coordinator component. The node with the incoming arrow is the starting state (*i.e.* the state $S0$ shown in Figure 12). An arc from a node n_1 to a node n_2 denotes a transition from n_1 to n_2. The transition labels prefixed by "!" denote output actions. The transition labels prefixed by "?" denote input actions. For each transition label, the number, that follows the symbol "_", identifies the connector (and hence the corresponding component) on which the action has been performed; 1 is for *Client1*, 2 is for *Client2* and 3 is for *Server*.

Since the "*no-op*" coordinator has no deadlock, in Figure 13 we directly show the initial coordinator that implements the *"AlternatingProtocol"* policy.

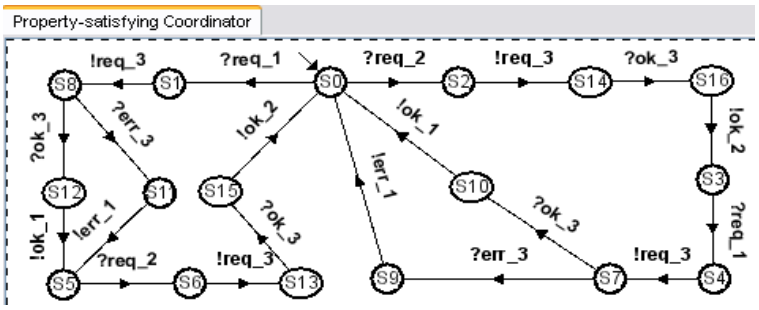

Figure 13. SYNTHESIS*'s screen-shot of the intial coordinator graph*

***Module* F:** *generator of the initial coordinator actual code*

This module implements a generator of the initial coordinator actual code. It is structured analogously to module **A**. Thus (when it is used) it refers to one or more specific development platforms. Currently it only supports the generation of the code implementing the coordinator COM/DCOM component. This code refers to Microsoft Visual Studio with Active Template Library (ATL) as reference development platform.

The LTS of the initial coordinator is a state machine and module **F** derives the actual code of a COM/DCOM server that reflects the state machine's structure and behavior.

By continuing our example and by recasting it in the COM/DCOM context, we show the automatically derived header (`KCoordinatorPSat.h`) file of the coordinator component as follows:

```
#pragma once
#include "resource.h"
#include <vector>
using namespace std;
typedef vector<int> SLABELVECT;
#import "_ServerPrj.tlb" raw_interfaces_only, raw_native_types,
no_namespace, named_guids, auto_search
[
    object,
    uuid("536F2DB6-398A-4c1d-AE02-2BC29AC9F35F"),
    dual,   helpstring("IKCoordinatorPSat Interface"),
    pointer_default(unique)
] __interface IKCoordinatorPSat: IDispatch {
    [id(1),helpstring("method req")] HRESULT  req([in] int val,
    [out,retval] BSTR *status);
};

[
```

```
    coclass,
    threading("apartment"),
    vi_progid("KCoordinatorPrjPSat.KCoordinatorPSat"),
    progid("KCoordinatorPrjPSat.KCoordinatorPSat.1"),
    version(1.0),
    uuid("56207063-CB18-43d3-BE53-705D8E1969C4"),
    helpstring("KCoordinatorPSat Class")
] class ATL_NO_VTABLE CKCoordinatorPSat:
    public IKCoordinatorPSat,
    public IDispatchImpl<IServer, &__uuidof(IServer), &LIBID_ServerPrj,
    1, 0>
{ private:
    static IServerPtr pIServer;
    static SLABELVECT sLabelVect;
    static int clientsCounter;
    int chId;

public:
    CKCoordinatorPSat()
    {
        int stateLabel;
        stateLabel = 0
        sLabelVect.push_back(stateLabel);
        clientsCounter++;
        chId = clientsCounter;
        CoTreatAsClass(__uuidof(CServer), CLSID_NULL);
        pIServer = new IServerPtr(__uuidof(CServer));
        CoTreatAsClass(__uuidof(CServer), __uuidof(CKCoordinatorPSat));
    }

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct(){ return S_OK; }

    void FinalRelease(){}

private: bool ElementOf(int sl);

public:
    STDMETHOD(req)(int val, BSTR * status);
};
```

In deriving this code, module **F** takes into account that the coordinator component is a single-thread and composite server that encapsulates all servers into the CFA system through a containment/delegation mechanism. Moreover it also exports all its inner interfaces (*i.e.* the interfaces exported by the encapsulated servers). *Server* exports to its clients an interface *IServer* that declares one method: *req*. The coordi-

nator component code imports the type library of *Server*[7] and implements *IServer* by defining a new COM class.

It is worthwhile noticing that the COM class of the coordinator component defines, for its objects, both a *shared* and *non-shared* state. The *shared* state is represented by a set of *static* private members while the *non-shared* state is represented by the remaining private members. "**pIServer**" is a COM/DCOM *smart pointer* (MSD 2004) referring to *IServer*. By defining and using "**sLabelVect**", *SYNTHESIS* is able to model a vector of coordinator graph's state labels. "**clientsCounter**" counts the number of clients connected to the coordinator component. "**chId**" is different for each client connected to the coordinator component. It models the identifier of the channel that the client uses to be connected to the coordinator (and hence it identifies a client connected to the coordinator component too).

In deploying the CBA version of the system, this module takes into account that the coordinator has to be registered in the same machine than the servers encapsulated into it. This, in turn, implies all the encapsulated servers have to be registered in the same machine as well. Moreover, by referring to Section 3.2, for each server S, the value of the key *HKEY_CLASSES_ROOT/CLSID/<...unique class identifier of S...>/TreatAs* is the value of the CLSID of the coordinator component (*i.e.* **uuid("56207063-CB18-43d3-BE53-705D8E1969C4")**). This is needed to interpose the coordinator component between the clients and the servers in order to ensure that they no longer communicate directly.

The following is the automatically derived source (`KCoordinatorPSat.cpp`) file of the coordinator component:

```
#include "stdafx.h" #include "KCoordinatorPSat.h"
#import "_ServerPrj.tlb" raw_interfaces_only, raw_native_types,
no_namespace, named_guids, auto_search
IServerPtr CKCoordinatorPSat::pIServer = NULL;
SLABELVECT CKCoordinatorPSat::sLabelVect;
int CKCoordinatorPSat::clientsCounter = 0;

HRESULT CKCoordinatorPSat::req(int val, BSTR *status) {
    HRESULT hr = S_OK;
    try
    {
        if(chId == 1) {
            if(ElementOf(0)) { // a state in which a request of req_1 is
                               // allowed
                // method call delegation
                hr = pIServer->req(val, status);

                // Update the vector of consistent state labels...
                sLabelVect.erase(sLabelVect.begin(), sLabelVect.end());
```

7. It is derived by executing the MIDL compiler which is bundled with *Microsoft Visual Studio*.

```
                sLabelVect.push_back(5);
                // ...end of updating
            }
            else if(ElementOf(3)) { // a state in which a request of
                                    // req_1 is allowed
                // method call delegation
                hr = pIServer->req(val, status);

                // Update the vector of consistent state labels...
                sLabelVect.erase(sLabelVect.begin(), sLabelVect.end());
                sLabelVect.push_back(0);
                // ...end of updating
            }
        }
        else if(chId == 2) {
            if(ElementOf(5)) { // a state in which a request of
                               // req_2 is allowed
                // method call delegation
                hr = pIServer->req(val, status);

                // Update the vector of consistent state labels...
                sLabelVect.erase(sLabelVect.begin(), sLabelVect.end());
                sLabelVect.push_back(0);
                // ...end of updating
            }
            else if(ElementOf(0)) { // a state in which a request of
                                    // req_2 is allowed
                // method call delegation
                hr = pIServer->req(val, status);

                // Update the vector of consistent state labels...
                sLabelVect.erase(sLabelVect.begin(), sLabelVect.end());
                sLabelVect.push_back(3);
                // ...end of updating
            }
        }
    }
    catch (_com_error e) {
        MessageBox(NULL, e.ErrorMessage(), "SMART K: COM Error", MB_OK);
    }
    return hr;
}

bool CKCoordinatorPSat::ElementOf(int sl) {
    for(unsigned int i=0; i<sLabelVect.size(); i++)
        if((int)sLabelVect[i] == sl)
                return true;
    return false;
}
```

In the source file, for each public method m, the module **F** derives the code implementing the logic of the coordinator related to m. This logic is trivially derived by visiting the coordinator graph (see Figure 13).

It is worthwhile noticing that the coordinator component does not care about checking the value of actual parameters of a method. In fact, it is only a method call delegator whose delegation logic respects both the deadlock-freeness and the specified coordination policies. It solves the problem of making the components (forming the CFA system) able to interact, in a deadlock-free way, by following only the specified desired behaviors (*i.e.* the coordination policies). Dealing in a black-box components setting, the coordinator does not care about details related to the components internal behavior. This is the reason why, in drawing the MSCs specification of the CFA system, *SYNTHESIS* does not require that a user must specify the actual parameters list of a method call.

5.2. *Second phase*

In Figure 14 we show the input and output data performed by *SYNTHESIS* within the second phase. Analogously to what we have done in Section 5.1, by means of capital letters, we obtain a direct mapping between Figure 4 and Figure 14.

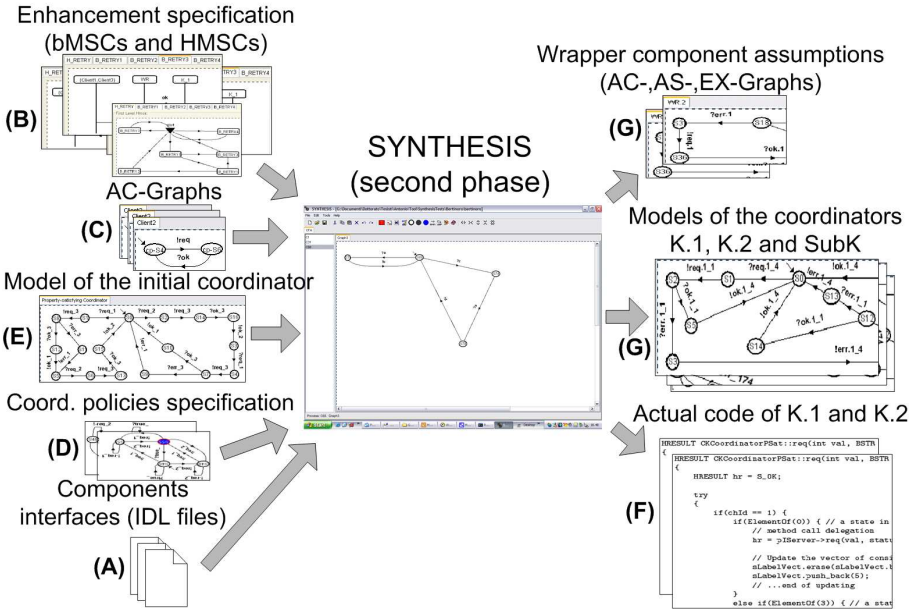

Figure 14. *Input and output data performed by SYNTHESIS within the second phase*

In the following, we describe the module **G** which is involved in the second phase. In doing this, we completely refer to Section 4.2.

***Module* G:** *wrapper component assumptions*

This module is responsible for deriving behavioral models of the wrapper component corresponding to its assumptions on the environment (*i.e.* its AC-, AS- and EX-Graph). We recall that the wrapper is needed to apply a specified enhancement. Since an enhancement is specified in terms of a bMSC and HMSC specification of the wrapper, to do that, this module exploits both the module **B** and **C**. More precisely, it directly requires the module **E** which, in turn, exploits **C** and subsequently **B**. Moreover it might also use the module **A** to parse IDL files of the wrapper component and of possible new components. We recall that depending on the enhancement, new components might be required in order to deal with architectural updates.

By continuing the example of Section 5.1, *Client1* is an interactive client and once an erroneous notification occurs, it shows a dialog window displaying information about the error. The user might not appreciate this error message and he might lose the degree of trust in the system. By recalling that the dependability of a system reflects the user's degree of trust in the system, this example shows a commonly practiced dependability-enhancing technique. The wrapper *WR* attempts to hide the error to the user by re-sending the request a finite number of times. This is the *RETRY* enhancement specified in Figure 15.a). The wrapper *WR* re-sends at most two times. Moreover, the *RETRY* enhancement specifies an architectural update of V obtained by inserting a new component (*i.e. Client3*).

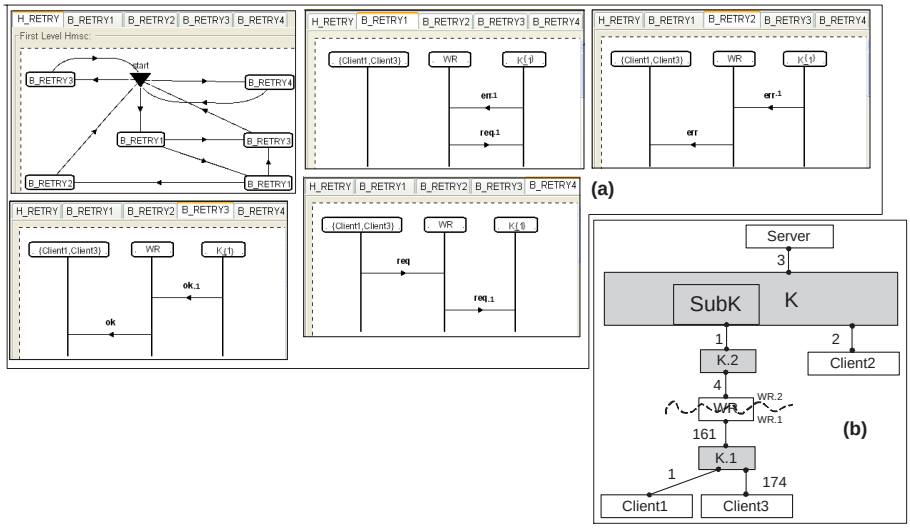

Figure 15. $bMSCs$ *and* $hMSC$ *specification of the* RETRY *enhancement*

In order to apply the specified enhancement, this module performs the following operations: i) from the MSC specification (see **B** in Figure 14), it derives both the AC-Graphs of the new inserted components (*i.e. Client3*) and the AC-Graphs $WR.1$ and $WR.2$. Within the CBA style, $WR.1$ ($WR.2$) describes the actual behavior of

the wrapper only related to its cooperation with the components below (above) it; ii) from the model of the initial coordinator (*i.e.* K in Figure 15.b) corresponding to **E** in Figure 14) and the MSC specification of the enhancement, it derives the AC-Graph of $SubK$; iii) it decomposes the CBA system obtained by the execution of the first phase in two CFA systems. One is formed by $WR.1$, $Client1$ and $Client3$. In Figure 16 we show the AC-Graph of $WR.1$. The AC-Graphs of $Client1$ (see Figure 8) and $Client3$ are equals.

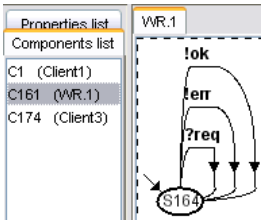

Figure 16. *AC-Graph of* $WR.1$

The other CFA is formed by $WR.2$ and $SubK$. In Figure 17 we show the AC-Graphs of $WR.2$ and $SubK$.

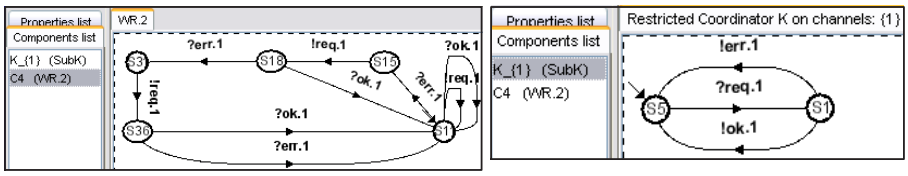

Figure 17. *AC-Graphs of* $WR.2$ *and* $SubK$

To insert *WR*, this module automatically synthesizes two new coordinators $K.1$ and $K.2$. This is done, analogously to what we have reported in Section 5.1, by performing the first phase for the two CFA systems above mentioned. In Figure 18 we show the coordinator graphs automatically synthesized for $K.1$ and $K.2$.

It is worthwhile noticing that, at this stage, it is also possible to enforce new desired behavior on $K.1$ and $K.2$ analogously to what we have done in Section 5.1.

The parallel composition K_{new} of K, $K.1$, $K.2$ and WR represents the model of the enhanced coordinator. In order to derive the code implementing K_{new}, this module (by means of the module **F**) automatically derives the actual code implementing $K.1$ and $K.2$ analogously to what we have done in Section 5.1 for deriving the code of the initial coordinator.

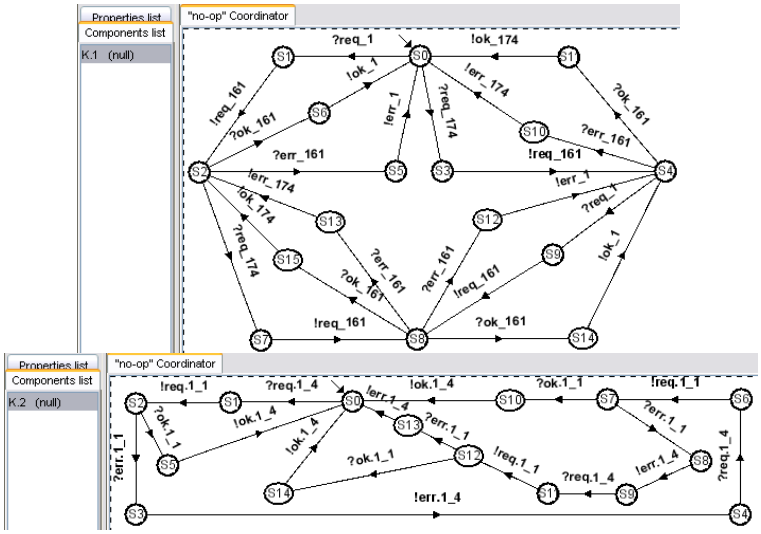

Figure 18. *Coordinator graphs for $K.1$ and $K.2$*

6. Conclusion and future work

In this paper, we have presented our tool *SYNTHESIS* for synthesizing "correct" and protocol-enhanced adaptors. *SYNTHESIS* implements the theoretical approach formalized in (Autili *et al.*, 2004b). That approach combines the approaches of protocol transformation formalization (Spitznagel *et al.*, 2003) and of automatic coordinator synthesis (Inverardi *et al.*, 2003b, Tivoli *et al.*, 2004b, Inverardi *et al.*, 2003a) to produce a new technique for automatically synthesizing failure-free and protocol enhanced coordinators. This paper is a companion paper of (Autili *et al.*, 2004b) since it points out technical issues related to the underlying theoretical approach (Autili *et al.*, 2004b) instantiated in the specific domain of COM/DCOM component-based application. *SYNTHESIS* is available at the *URL*: *http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.html*.

The key results are: (i) the approach implemented by *SYNTHESIS* is compositional in the automatic synthesis of the enhanced coordinator; that is, each wrapper represents a modular protocol transformation so that we can apply coordinator protocol enhancements in an incremental way by re-using the code synthesized for already applied enhancements; (ii) *SYNTHESIS* is able to add extra functionality to a coordinator beyond simply restricting its behavior; (iii) this, in turn, allows us to enhance a coordinator with respect to a useful set of protocol transformations such as the set of transformations referred in (Spitznagel *et al.*, 2003).

The current version of *SYNTHESIS* is a prototype produced by following an evolutionary approach to systems development. Thus *SYNTHESIS* is still subject to further

extensions. As future work, we plan to: (i) develop more user-friendly specification of both the desired behaviors and the protocol enhancements (e.g., UML2 Interaction Overview Diagrams and Sequence Diagrams); (ii) validate the applicability of the whole approach to large-scale examples; (iii) combine our approach with the approach of adaptors synthesis described in (Brogi *et al.*, 2004). Depending on the logic implemented by the wrapper, this might allow us to automatically synthesize the wrapper from an its partial specification rather than built it by scratch or acquire it as a pre-existent (COTS) component; (iv) derive a distributed implementation of the coordinator in order to do not require the coordinator and its encapsulated serves to be registered in the same machine.

Acknowledgements

We acknowledge prof. Paola Inverardi which has supervised us in developing the formal framework underlying the *SYNTHESIS* tool. We also acknowledge the reviewers that have provided authors with very valuable comments and constructive suggestions to improve our paper for this special issue.

7. References

Autili M., Sintesi Automatica di Connettori per Protocolli di Comunicazione Evoluti, Tesi di laurea in informatica, Apr, 2004. http://www.di.univaq.it/tivoli/AutiliThesis.pdf.

Autili M., Inverardi P., Tivoli M., "Automatic adaptor synthesis for protocol transformation", *Proc. of the WCAT Workshop at ECOOP*, p. 39-46, 2004a.

Autili M., Inverardi P., Tivoli M., Garlan D., "Synthesis of 'correct' adaptors for protocol enhancement in component-based systems", *Proc. of the SAVCBS Workshop at ESEC/FSE*, 2004b.

Balemi S., Hoffmann G. J., Gyugyi P., Wong-Toi H., Franklin G. F., "Supervisory Control of a Rapid Thermal Multiprocessor", *IEEE Transactions on Automatic Control*, vol. 38, n° 7, p. 1040-1059, July, 1993.

Brogi A., Canal C., Pimentel E., "Behavioural Types and Component Adaptation", *Proc. of AMAST*, vol. 3116 of *LNCS*, Stirling - UK, p. 42-56, July, 2004.

Clarke E. M., Grumberg O., Peled D. A., *Model Checking*, The MIT Press, 2001.

de Alfaro L., Henzinger T., "Interface automata", *Proc. of ESEC/FSE*, p. 109-120, 2001.

Garlan D., Allen R., Ockerbloom J., "Architectural mismatch: Why reuse is so hard", *IEEE Software*, vol. 12, n° 6, p. 17-26, Nov, 1995.

Inverardi P., Tivoli M., "Automatic failures-free connector synthesis: An example", *Proc. of the RISSEF Int. Workshop*, vol. 2941 of *LNCS*, Venice, Italy, p. 184-197, October, 2002.

Inverardi P., Tivoli M., "Failure-free Connector Synthesis for Correct Components Assembly", *Proc. of the SAVCBS Workshop*, 2003a.

Inverardi P., Tivoli M., *Software Architecture for Correct Components Assembly*, vol. 2804 of *LNCS*, Sept., 2003b.

Inverardi P., Tivoli M., Connectors synthesis for failures-free component based architectures, Technical report, University of L'Aquila, Department of Computer Science, http://www.di.univaq.it/tivoli/ffsynthesis.pdf, January, 2004.

ITU, ITU-T Recommendation Z.120. Message Sequence Charts. (MSC'96), Technical report, Geneva, 1996.

Jackson D., Sullivan K. J., "COM Revisited: Tool-Assisted Modelling of an Architectural Framework", *Proc. of ACM SIGSOFT FSE*, San Diego, CA, p. 149-158, 2000.

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

MSD, Microsoft Developer Network Library, Technical report, 2004.

Passerone R., de Alfaro L., Henzinger T., Sangiovanni-Vincentelli A. L., "Convertibility Verification and Converter Synthesis: Two Faces of the Same Coin", *Proc. of ICCAD*, p. 132-139, 2002.

Platt D. S., *Understanding COM+*, Microsoft Press, 1999.

Shaw M., Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

Spitznagel B., Garlan D., "A compositional formalization of connector wrappers", *Proc. of ICSE*, p. 374-384, May, 2003.

Sullivan K. J., Marchucov M., Socha J., "Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model", *IEEE Transactions on Software Engineering*, vol. 25, n° 4, p. 584-599, 1999.

Sullivan K. J., Socha J., Marchucov M., "Using Formal Methods to Reason about Architectural Standards", *Proc. of ICSE*, p. 503-513, 1997.

Szyperski C., *Component Software. Beyond Object Oriented Programming*, Addison Wesley, 1998.

Tivoli M., Garlan D., Coordinator synthesis for reliability enhancement in component-based systems, Technical report, Carnegie Mellon University, C.S.Dep., http://www.di.univaq.it/tivoli/CMUtechrep.pdf, 2004a.

Tivoli M., Inverardi P., Presutti V., Forghieri A., Sebastianis M., "Correct Components Assembly for a Product Data Management Cooperative System", *Proc. of CBSE*, vol. 3054 of *LNCS*, p. 84-99, 2004b.

Uchitel S., Kramer J., Magee J., "Detecting Implied Scenarios in Message Sequence Chart Specifications", *Proc. of ESEC/FSE*, p. 74-82, Vienna - Sep, 2001.

Yellin D. M., Strom R. E., "Protocol Specifications and Component Adaptors", *ACM Transactions on Programming Languages and Systems*, vol. 19, n° 2, p. 292-333, March, 1997.