Synthesis of "correct" adaptors for protocol enhancement in component-based systems

Article · April 2015

CITATIONS
12

READS
67

4 authors:

Marco Autili
Università degli Studi dell'Aquila
118 PUBLICATIONS **1,538** CITATIONS
SEE PROFILE

Paola Inverardi
Università degli Studi dell'Aquila
333 PUBLICATIONS **9,139** CITATIONS
SEE PROFILE

Massimo Tivoli
Università degli Studi dell'Aquila
127 PUBLICATIONS **3,221** CITATIONS
SEE PROFILE

David Garlan
Carnegie Mellon University
466 PUBLICATIONS **31,032** CITATIONS
SEE PROFILE

Synthesis of "correct" adaptors for protocol enhancement in component-based systems*

Marco Autili, Paola Inverardi, Massimo Tivoli
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila
{marco.autili, inverard, tivoli}@di.univaq.it

David Garlan
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
garlan@cs.cmu.edu

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. This might require to restrict the system's behavior to a subset of safe behaviors. On the other hand, it might be necessary to enhance the current communication protocol. This might require to augment the system's behavior to introduce more sophisticated interactions among components. We address these problems by enhancing our architectural approach which allows for detection and recovery of incompatible interactions by synthesizing a suitable coordinator. Taking into account the specification of the system to be assembled and the specification of the protocol enhancements, our tool (called *SYNTHESIS*) automatically derives, in a compositional way, the glue code for the set of components. The synthesized glue code implements a software coordinator which avoids incompatible interactions and provides a protocol-enhanced version of the composed system. By using an assume-guarantee technique, we are able to check, in a compositional way, if the protocol enhancement is consistent with respect to the restrictions applied to assure the specified safe behaviors.

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 [14] components introduces a set of problems, mainly related to communication and compatibility aspects. Often, components may have incompatible interaction behavior. This might require to restrict the system's behavior to a subset of safe behaviors. For example, restrict to the subset of deadlock-free behaviors or, in general, to a specified subset of desired behaviors. Moreover, it might be necessary to enhance the current communication protocol. This requires augmenting the system's behavior to introduce more sophisticated interactions among components. These enhancements (i.e.: protocol transformations) 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 enhancing our architectural approach which allows for detection and recovery of incompatible interactions by synthesizing a suitable coordinator [6, 9, 11]. This coordinator represents a initial glue code. So far, as reported in [6, 9, 11], the approach only focussed on the restriction of the system's behavior to a subset of safe (i.e.: desired) behaviors. In this paper, we propose an extension that makes the coordinator synthesis approach also able to automatically transform the coordinator's protocol by enhancing the initial glue code. We implemented the whole approach in our *SYNTHESIS* tool [9, 15] (*http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.html*). In [15], which is a companion paper, we also apply *SYNTHESIS* to a real-scale context. Since in this paper we are focusing only on the formalization of the approach, in Section 5, we will simply refer to an explanatory example and we will omit implementation details which are completely described in [15].

Starting from the specification of the system to be assembled and from the specification of the desired behaviors, *SYNTHESIS* automatically derives the initial glue code for the set of components. This initial glue code is implemented as a coordinator mediating the interaction among components by enforcing each desired behavior as reported in [6, 9, 11]. Subsequently, taking into account the specification of the needed protocol enhancements and performing the extension we formalize in this paper, *SYNTHESIS* automatically derives, in a compositional way, the enhanced glue code for the set of components. This last step represents the contribution of this paper with respect to [6, 9, 11]. The enhanced glue code implements a software coordinator which

*This work is an extended and revisited version of [7].

avoids not only incompatible interactions but also provides a protocol-enhanced version of the composed system. More precisely, this enhanced coordinator is the composition of a set of new coordinators and components assembled with the initial coordinator in order to enhance its protocol. Each new component represents a wrapper component. A wrapper intercepts the interactions corresponding to the initial coordinator's protocol in order to apply the specified enhancements without modifying [1] the initial coordinator and the components in the system. The new coordinators are needed to assemble the wrappers with the initial coordinator and the rest of the components forming the composed system. It is worthwhile noticing that, in this way, we are readily compose-able; we can treat the enhanced coordinator as a new *composite* initial coordinator and enforce new desired behaviors as well as apply new enhancements. This allows us to perform a protocol transformation as composition of other protocol transformations by improving on the reusability of the synthesized glue code.

When we apply the specified protocol enhancements to produce the enhanced coordinator, we might re-introduce incompatible interactions avoided by the initial coordinator. That is, the enhancements do not hold the desired behaviors specified to produce the initial coordinator. In this paper, we also show how to check if the protocol enhancement holds the desired behaviors enforced through the initial coordinator. This is done, in a compositional way, by using an assume-guarantee technique [5].

The paper is organized as follows: Section 2 discusses related work. Section 3 introduces background notions helpful to understand our approach. Section 4 illustrates the technique concerning the enhanced coordinator synthesis. Section 5 formalizes the coordinator synthesis approach for protocol enhancement in component-based systems, and uses a simple explanatory example to illustrate the ideas. Section 6 formalizes the technique used to check the consistency of the applied enhancements with respect to the enforced desired behaviors. Section 7 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 [3]. In those approaches system's 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 behaviors resembles one aspect of our approach defined in Section 4, since we also eliminate certain incompatible behaviors 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 [4, 10]. In [4], they use an automata-based approach 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 [10], they use a game theoretic approach 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 developed in our precedent works [6, 9, 11]. That is the restriction of the system's behavior to a subset of safe behaviors. Unlike the work presented in this paper, they are only able to restrict the system's behavior to a subset of desired behaviors 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 [18]. 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 previous work, of one of the authors, we showed how to use formalized protocol transformations to augment connector behavior [13]. 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 this work it does not provide automatic support for composing wrappers, or for automatically eliminating incompatible interaction behaviors.

In other previous work, by two of the authors, we showed how to apply protocol enhancements by dealing with components that might have syntactic incompatibility of interfaces [16]. However the approach described in [16] is limited only to consider deadlock-free coordinator.

3. BACKGROUND

In this section we discuss the background needed to understand the approach that we formalize in Section 4.

3.1 The reference architectural style

[1] This is needed to achieve compose-ability in both specifying the enhancements and implementing them.

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 [12]. To simplify matters we will 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.*

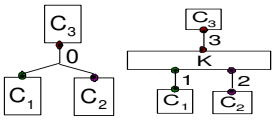

Figure 1: A sample of a CFA and the corresponding CBA

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 0, 1, 2 and 3 are connectors.

3.2 Configuration formalization

To formalize the behavior of a system we use *High level Message Sequence Charts* (HMSCs) and *basic Message Sequence Charts* (bMSCs) [1] specification of the composed system. From it, we can derive the corresponding CCS (*Calculus of Communicating Systems*) processes [8] (and hence *Labeled Transitions Systems* (LTSs)) by applying a suitable adaptation of the translation algorithm presented in [17]. HMSC and bMSC specifications are useful as input language, since they are commonly used in software development practice. Thus, CCS can be regarded as an internal specification language. Later we will see an example of derivation of LTSs from a bMSCs and HMSCs specification (Section 5.1).

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. That coordinator 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). The "*no-op*" coordinator is automatically derived by performing the algorithm described in [6, 9, 11].

Formally, using CCS we define the CFA and the CBA for a set of components $C_1, .., C_n$ as follows:

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

Definition 2. Coordinator Based Architecture (CBA) $CBA \equiv (C_1[f_1^0] \mid C_2[f_2^0] \mid ... \mid C_n[f_n^0] \mid K) \backslash \bigcup_{i=1}^{n} Act_{C_i}[f_i^0]$ where for all $i = 1, .., n$, Act_{C_i} is the action set of the CCS process C_i, and f_i^0 are relabelling functions such that $f_i^0(\alpha) = \alpha[i/0]$ for all $\alpha \in Act_{C_i}$; K is the CSS process corresponding to the automatically synthesized coordinator.

By referring to [8], $\mid$ is the *parallel composition* operator and $\backslash$ is the *restriction* operator. There is a finite set of visible actions $Act = \{a_i, \overline{a}_j, b_h, \overline{b}_k, ...\}$ over which α ranges. We denote by $\overline{\alpha}$ the action complement: if $\alpha = a_j$, then $\overline{\alpha} = \overline{a}_j$, while if $\alpha = \overline{a}_j$, then $\overline{\alpha} = a_j$. By $\alpha[i/j]$ we denote a substitution of i for j in α. If $\alpha = a_j$, then $\alpha[i/j] = a_i$. Each $Act_{C_i} \subseteq Act$. By referring to Figure 1, 0 identifies the only connector (i.e: communication channel) present in the CFA version of the composed system. Each relabelling function f_i^0 is needed to ensure that the components $C_1, ..C_n$ no longer synchronize directly. In fact by applying these relabelling functions (i.e.: f_i^0 for all i) each component C_i synchronizes only with the coordinator K through the connector i (see right-hand side of Figure 1).

3.3 Automatic synthesis of failure-free coordinators

In this section, we simply recall that from the MSCs specification of the CFA and from a specification of desired behaviors, the old version of *SYNTHESIS* automatically derives the corresponding deadlock-free CBA which satisfies each desired behavior. This is done by synthesizing a suitable coordinator that we call failure-free coordinator. Informally, first we synthesize a "no-op" coordinator. Second, we restrict its behavior by avoiding possible deadlocks and enforcing the desired behaviors. Each desired behavior is specified as a *Linear-time Temporal Logic* (LTL) formula (and hence as the corresponding *Büchi Automaton*) [5]. Refer to [6, 9, 11] for a formal description of the old approach and for a brief overview on the old version of our *SYNTHESIS* tool.

4. METHOD DESCRIPTION

In this section, we informally describe the extension of the old coordinator synthesis approach [6, 9, 11] that we formalize in Section 5 and we implemented in the new version of the *SYNTHESIS* tool. The extension starts with

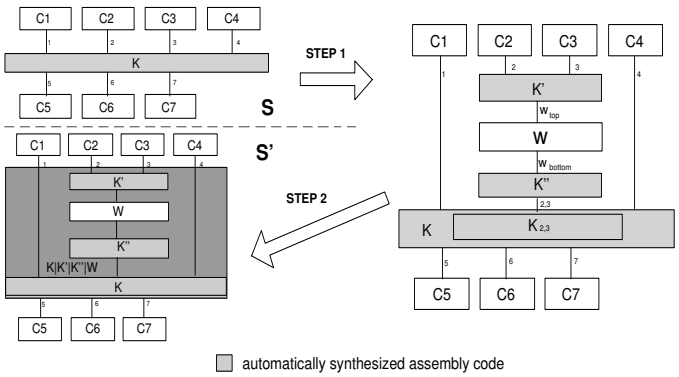

Figure 2: 2 step method

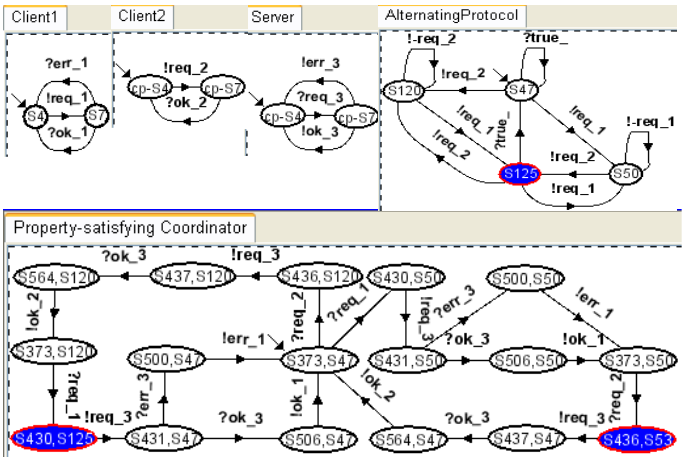

Figure 3: *LTSs* specification of S and Büchi Automata specification of P

a deadlock-free CBA which satisfies specified desired behaviors and produces the corresponding protocol-enhanced CBA.

The problem we want to face can be informally phrased as follows: *let P be a set of desired behaviors, given a deadlock-free and P-satisfying CBA system S for a set of black-box components interacting through a coordinator K, and a set of coordinator protocol enhancements E, if it is possible, automatically derive the corresponding enhanced, deadlock-free and P-satisfying CBA system S'.*

We are assuming a specification of: i) S in terms of a description of components and a coordinator as LTSs, ii) P in terms of a set of Büchi Automata, and of iii) E in form of bMSCs and HMSCs specification. In the following, we discuss our method proceeding in two steps as illustrated in Figure 2.

In the first step, by starting from the specification of P and S, if it is possible, we apply each protocol enhancement in E. This is done by inserting a wrapper component W between K (see Figure 2) and the portion of S concerned with the specified protocol enhancements (i.e.: the set of $C2$ and $C3$ components of Figure 2). It is worthwhile noticing that we do not need to consider the entire model of K but we just consider the *"sub-coordinator"* which represents the portion of K that communicates with $C2$ and $C3$ (i.e.: the *"sub-coordinator"* $K_{2,3}$ of Figure 2). $K_{2,3}$ represents the *"unchangeable"*[2] 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 applies the enhancements in E on the interactions performed on the communication channels 2 and 3 (i.e.: connectors 2 and 3 of Figure 2). We first decouple K (i.e.: $K_{2,3}$), $C2$ and $C3$ to ensure that they no longer synchronize directly. Then we automatically derive a behavioral model of W (i.e.: a LTS) from the bMSCs and HMSCs specification of E. We do this by exploiting our implementation of the translation algorithm described in [17]. Finally, if the insertion of W in S allows the resulting composed system (i.e.: S' 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' and K''. In general, K' always refers to the coordinator between W and the components affected by the enhancement. K'' always refers to the coordinator between K and W. By referring to Section 3.1, to do this, we automatically derive two behavioral models of W: i) W_TOP which is the behavior of W only related to its *top* interface and ii) W_BOTTOM which is the behavior of W only related to its *bottom* interface.

In the second step, we derive the implementation of the synthesized glue code used to insert W in S. This glue code is the actual code implementing K'' and K'. By referring to Figure 2, the parallel composition K_{new} of K, K', K'' 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 other words, our approach is compositional in the automatic synthesis of the enhanced glue code.

5. METHOD FORMALIZATION

In this section, by using an explanatory example, we formalize the two steps of our method. For the sake of brevity we limit ourselves to formalize the core of the extended approach. Refer to [2] for the formalization of the whole approach.

In Figure 3, we consider screen-shots of the *SYNTHESIS* tool related to both the specification of S and of P. *Client*1, *Client*2 and *Server* are the components in S. *Property-satisfying Coordinator* is the coordinator in S which satisfies the desired behavior denoted with *AlternatingProtocol*. In this example, *AlternatingProtocol* is the only element in the specification P. The CBA configuration of S is shown in the right-hand side of Figure 1 where C_1, C_2, C_3 and K are *Client*1, *Client*2, *Server* and *Property-satisfying Coordinator* of Figure 3 respectively.

Each LTS describes the behavior of a component or of a coordinator instance in terms of the messages (seen as I/O

[2]Since we want to be readily compose-able, our goal is to apply the enhancements without modifying the coordinator and the components.

actions) exchanged with its environment[3]. Each node is a state of the instance. The node with the incoming arrow (e.g.: the state $S4$ of $Client1$ in Figure 3) is the starting state. 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 (i.e.: sent requests and notifications), while the transition labels prefixed by "?" denote input actions (i.e.: received requests and notifications). In each transition label, the symbol "_" followed by a number denotes the identifier of the connector on which the action has been performed. The filled nodes on the coordinator's LTS denote states in which one execution of the behavior specified by the Büchi Automaton *AlternatingProtocol* has been accomplished.

Each Büchi Automaton (see *AlternatingProtocol* in Figure 3) describes a desired behavior for S. Each node is a state of S. The node with the incoming arrow is the initial state. The filled nodes are the states accepting the desired behavior. The syntax and semantics of the transition labels is the same of the LTSs of components and coordinator except two kinds of action: i) a universal action (e.g.: $?true\_$ in Figure 3) which represents any possible action[4], and ii) a negative action (e.g.: $!-req\_2$ in Figure 3) which represents any possible action different from the negative action itself[5].

$Client1$ performs a request (i.e.: action $!req\_1$) and waits for a erroneous or successful notification: actions $?err\_1$ and $?ok\_1$ 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].

AlternatingProtocol specifies the behavior of S that guarantees the evolution of all components. It specifies that $Client1$ and $Client2$ must perform requests by using an alternating coordination protocol. More precisely, if $Client1$ performs an action req (the transition $!req\_1$ from the state $S47$ to the state $S50$ in Figure 3) then it cannot perform req again (the loop transition $!-req\_1$ on the state $S50$ in Figure 3) if $Client2$ has not performed req (the transition $!req\_2$ from the state $S50$ to the accepting state $S125$ in Figure 3) and viceversa.

In Figure 4.(a), we consider the specification of E as given in input to the *SYNTHESIS* tool. In this example, the $RETRY$ enhancement is the only element in E.

$Client1$ 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 users 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 4.(a).

[3]The environment of a component/coordinator is the parallel composition of all others components in the system.

[4]The prefixed symbols "!" or "?", in the label of a universal action, are ignored by *SYNTHESIS*.

[5]The prefixed symbols "!" or "?", in the label of a negative action, are still interpreted by *SYNTHESIS*.

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

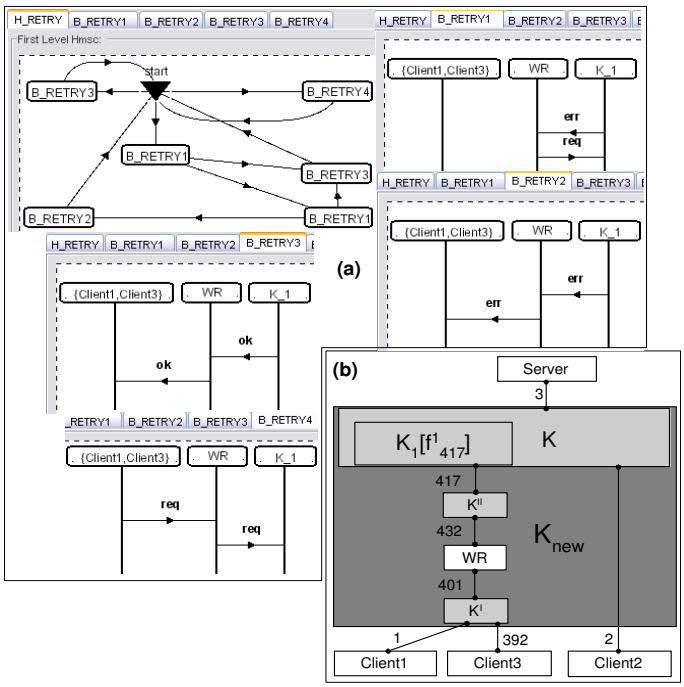

Figure 4: $bMSCs$ **and** $hMSC$ **specification of** E**:** ***RETRY*** **enhancement**

The wrapper WR re-sends at most two times. Moreover, the $RETRY$ enhancement specifies an update of S obtained by inserting $Client3$ which is a new client. In specifying enhancements, we use *"augmented"*-bMSCs. By referring to [1], each usual bMSC represents a possible execution scenario of the system. Each execution scenario is described in terms of a set of interacting components, sequences of method call and possible corresponding return values. To each vertical lines is associated an instance of a component. Each horizontal arrow represents a method call or a return value. Each usual HMSC describes possible continuations from a scenario to another one. It is a graph with two special nodes: the starting and the ending node. Each other node is related to a specified scenario. An arrow represents a transition from a scenario to another one. In other words, each HMSC composes the possible execution scenarios of the system. The only difference between *"augmented"*-bMSCs and usual bMSCs is that to each vertical line can be associated a set of component instances (e.g.: $\{Client1, Client3\}$ in Figure 4.(a)) rather than only one instance. This is helpful when we need to group components having the same interaction behavior.

5.1 First step: wrapper insertion procedure

By referring to Section 4, each enhancement MSCs specification (see Figure 4.(a)) is in general described in terms of the *sub-coordinator* K_1 (i.e.: K_1 in Figure 4.(a)), the wrapper (WR), the components in S ($Client1$) and the new components ($Client3$). The LTS of the *sub-coordinator* is automatically derived from the LTS of the coordinator in S (K) by performing the following algorithm:

Definition 3. $L_{j,\ldots,j+h}$ *construction algorithm*
Let L be the LTS for a component (or a coordinator) C,

we derive the LTS $L_{j,..,j+h}$, $h \geq 0$, of the *behavior of C on channels j,..,j+h* as follows:

1. set $L_{j,..,j+h}$ equal to L;
2. for each loop (ν, ν) of $L_{j,..,j+h}$ labeled with an action $\alpha = a_k$ where $k \neq j, .., j+h$ do:
   remove (ν, ν) from the set of arcs of $L_{j,..,j+h}$;
3. for each arc (ν, μ) of $L_{j,..,j+h}$ labeled with an action $\alpha = a_k$ where $k \neq j, .., j+h$ do:
   - remove (ν, μ) from the set of arcs of $L_{j,..,j+h}$;
   - if μ is the starting state then
     set ν as the starting state;
   - for each other arc (ν, μ) of $L_{j,..,j+h}$ do:
     replace (ν, μ) with (ν, ν);
   - for each arc (μ, ν) of $L_{j,..,j+h}$ do:
     replace (μ, ν) with (ν, ν);
   - for each arc (μ, υ) of $L_{j,..,j+h}$ with $\upsilon \neq \mu, \nu$ do:
     replace (μ, υ) with (ν, υ);
   - for each arc (υ, μ) of $L_{j,..,j+h}$ with $\upsilon \neq \mu, \nu$ do:
     replace (υ, μ) with (υ, ν);
   - for each loop (μ, μ) of $L_{j,..,j+h}$ do:
     replace (μ, μ) with (ν, ν);
   - remove μ from the set of nodes of $L_{j,..,j+h}$;
4. until $L_{j,..,j+h}$ is a non-deterministic LTS (i.e.: it contains arcs labeled with the same action and outgoing the same node) do:
   - for each pair of loops (ν, ν) and (ν, ν) of $L_{j,..,j+h}$ labeled with the same action do:
     remove (ν, ν) from the set of arcs of $L_{j,..,j+h}$;
   - for each pair of arcs (ν, μ) and (ν, μ) of $L_{j,..,j+h}$ labeled with the same action do:
     remove (ν, μ) from the set of arcs of $L_{j,..,j+h}$;
   - for each pair of arcs $((\nu, \mu)$ and $(\nu, \upsilon))$ or $((\nu, \nu)$ and $(\nu, \upsilon))$ of $L_{j,..,j+h}$ labeled with the same action do:
     - remove (ν, υ) from the set of arcs of $L_{j,..,j+h}$;
     - if υ is the starting state then
       set ν as the starting state;
     - for each ingoing arc *in* in υ, outgoing arc *out* from υ and loop l on υ do:
       move the extremity on υ of *in*, *out* and l on ν;
     - remove υ from the set of nodes of $L_{j,..,j+h}$.

Informally, the algorithm of Definition 3 *"collapses"* (steps 1,2 and 3) linear and/or cyclic paths made only of actions on channels $k \neq j, .., j+h$. Moreover, it also avoids (step 4) possible *"redundant"* non-deterministic behaviors[7].

By referring to Figure 5, the LTS of K_1 is the *LTS Restricted Coord*.

In general, once we derived $K_{j,..,j+h}$, we decouple K from the components $C_j, .., C_{j+h}$ (i.e.: *Client*1) connected through the connectors $j, .., j+h$ which are related to the specified enhancement (i.e.: the connector 1). To do this, we use the decoupling function defined as follows:

[7]These behaviors might be a side effect due to the collapsing.

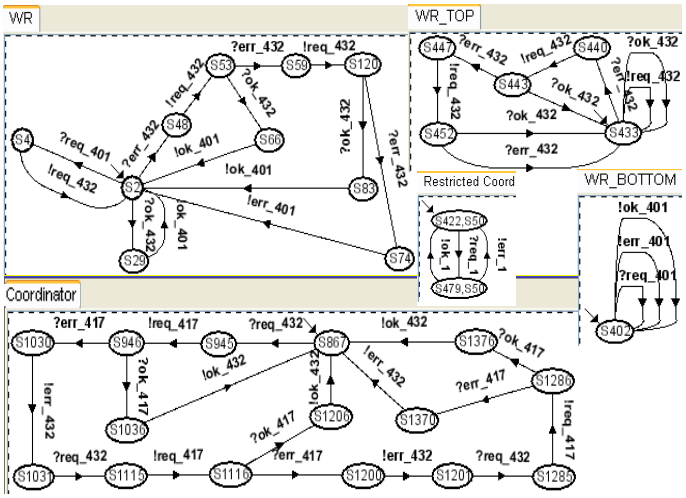

Figure 5: LTSs of wrapper, sub-coordinator and K''

Definition 4. Decoupling function
Let Act_K be the set of action labels of the coordinator K, let be $ID = \{j, \ldots, j+h\}$ a subset of all connectors identifiers[8] of K and let be $\delta \neq j, .., j+h$ a new connector identifier, we define the *"decoupling"* function $f_\delta^{j,..,j+h}$ as follows:

- $\forall\ a_i \in Act_K$, if $i \in ID$ then: $f_\delta^{j,..,j+h}(a_i) = a_\delta$;

The unique connector identifier δ is automatically generated by *SYNTHESIS*. In this way we ensure that K and *Client*1 no longer synchronize directly. In [15], we detail the correspondence between the decoupling function and components/coordinator deployment.
Now, by continuing the method described in Section 4, we derive the LTSs for the wrapper (see *WR* in Figure 5) and the new components (the *LTS* of *Client*3 is equal to the LTS of *Client*1 in Figure 3 except for the connector identifier). We recall that *SYNTHESIS* does that by taking into account the enhancements specification and by performing its implementation of the translation algorithm described in [17]. It is worthwhile noticing that *SYNTHESIS* automatically generates the connector identifiers for the actions performed by *WR* and *Client*3. By referring to Section 3.1, *WR* is connected to its environment through two connectors: i) one on its top interface (i.e.: the connector 432 in Figure 4.(b)) and ii) one on its bottom interface (i.e.: the connector 401 in Figure 4.(b)). Finally, as we will see in detail in Section 6, if the insertion of *WR* allows the resulting composed system to still satisfy *AlternatingProtocol*, *WR* is interposed between $K_1[f^1_{417}]$, *Client*1 and *Client*3. We recall that $K_1[f^1_{417}]$ is K_1 (see *Restricted Coord* in Figure 5) renamed after the decoupling. To insert *WR*, *SYNTHESIS* automatically synthesizes two new coordinators K' and K''. *Coordinator* in Figure 5 is the LTS for K''. For the purposes of this paper, in Figure 5, we omit the LTS for K'. K'' is derived by taking into account both the LTSs of $K_1[f^1_{417}]$ and *WR_TOP* (see Figure 5) and by performing the old synthesis approach [6, 9, 11]. While K' is derived analogously to the LTSs of *Client*1, *Client*3 and *WR_BOTTOM* (see Figure 5). The LTSs of *WR_TOP* and *WR_BOTTOM*

[8]By referring to Section 5, the connectors identifiers are postfixed to the labels in Act_K.

are WR_{432} and WR_{401} respectively. By referring to Definition 3, WR_{432} and WR_{401} model the behavior of WR on channels 432 and 401 respectively. The resulting enhanced, deadlock-free and *Alternating Protocol*-satisfying system is $S' \equiv ((Client_1 \mid Client_2 \mid Client_3 \mid Server \mid K_{new}) \setminus (Act_{Client1} \cup Act_{Client2} \cup Act_{Server} \cup Act_{Client3}))$ where $K_{new} \equiv ((K[f^1_{417}] \mid WR \mid K' \mid K'') \setminus (Act_{WR} \cup Act_{K_1[f^1_{417}]}))$ (see Figure 4.(b)). In [2], we proved the correctness and completeness of the approach.

5.2 Second step: synthesis of the glue code implementation

The parallel composition K_{new} represents the model of the enhanced coordinator. By referring to Section 4, we recall that K is the initial glue code for S and WR is a COTS component whose interaction behavior is specified by the enhancements specification E. That is, the actual code for WR and K is already available. Thus, in order to derive the code implementing K_{new}, *SYNTHESIS* automatically derives the actual code implementing K' and K''. Using the same technique described in [6, 9, 11, 15], this is done by exploiting the information stored in the nodes and arcs of the LTSs for K' and K''. More precisely, the code implementing K' and K'' reflects the structure of their LTSs which describe state machines. For the sake of brevity, here, we omit a detailed description of the code synthesis. Refer to [6, 9, 11, 15] for it. In [9, 15], we validated and applied *SYNTHESIS* for assembling *Microsoft COM/DCOM* components. The reference development platform of the current version of *SYNTHESIS* is *Microsoft Visual Studio 7.0* with *Active Template Library*.

6. CHECKING ENHANCEMENT CONSISTENCY

In this section we formalize a compositional technique to check if the applied enhancements are consistent with respect to the previously enforced desired behaviors. In other words, given the Büchi Automata specification of a desired behavior P_i, given the deadlock-free and P_i-satisfying coordinator K and given the MSCs specification of an enhancement E_i, we check (in a compositional way) if the enhanced coordinator K_{new} still satisfies P_i ($K_{new} \models P_i$).

In general, we have to check $((K[f^{j,..,j+h}_\delta] \mid WR \mid K' \mid K'') \setminus (Act_{WR} \cup Act_{K_{j,..,j+h}[f^{j,..,j+h}_\delta]})) \models P_i$. By exploiting the constraints of our architectural style, it is enough to check $((K[f^{j,..,j+m}_\delta] \mid K''_\delta) \setminus Act_{K_{j,..,j+m}[f^{j,..,j+m}_\delta]}) \models P_i$ where $\{j,..,j+m\}$ is the set of channel identifiers which are both channels in $\{j,..,j+h\}$ and in the set of channel identifiers for the action labels in P_i. In order to avoid the state explosion phenomenon we should decompose the verification without composing in parallel the processes $K[f^{j,..,j+m}_\delta]$ and K''_δ. We do that by exploiting the *assume-guarantee paradigm* for compositional reasoning [5].

By recasting the typical proof strategy of the *assume-guarantee paradigm* in our context, we know that if $\langle A \rangle K[f^{j,..,j+m}_\delta] \langle P_i \rangle$ and $\langle true \rangle K''_\delta \langle A \rangle$ hold then we can conclude that $\langle true \rangle \ ((K[f^{j,..,j+m}_\delta] \mid K''_\delta) \setminus Act_{K_{j,..,j+m}[f^{j,..,j+m}_\delta]}) \ \langle P_i \rangle$ is true. This proof strategy can also be expressed as the following inference rule:

$$\frac{\langle true \rangle K''_\delta \langle A \rangle \quad \langle A \rangle K[f^{j,..,j+m}_\delta] \langle P_i \rangle}{\langle true \rangle ((K[f^{j,..,j+m}_\delta] | K''_\delta) \setminus Act_{K_{j,..,j+m}[f^{j,..,j+m}_\delta]}) \langle P_i \rangle}$$

where A is a LTL formula (and hence it is modeled as a Büchi Automaton). We recall that, in S, K already satisfies P_i. Once we applied E_i to obtain the enhanced system S', A represents the assumptions (in S') on the environment of K that must be held on the channel δ in order to make K able to still satisfy P_i. Without loss of generality, let $\{j,..,j+m\}$ be $Channels_{P_i} \cap \{j,..,j+h\}$ where $Channels_{P_i}$ is the set of channel identifiers for the action labels in P_i; then A is the Büchi Automaton corresponding to $K_{j,..,j+m}[f^{j,..,j+m}_\delta][f_{env}]$ where $f_{env}(?\alpha) = !\alpha$ and $f_{env}(!\alpha) = ?\alpha$. For the example illustrated in Section 5, K''_δ and A are the Büchi Automata corresponding to K''_{417} (i.e.: $K2$ showed in Figure 6) and to $K_1[f^1_{417}][f_{env}]$ (*Assumption* showed in Figure 6) respectively.

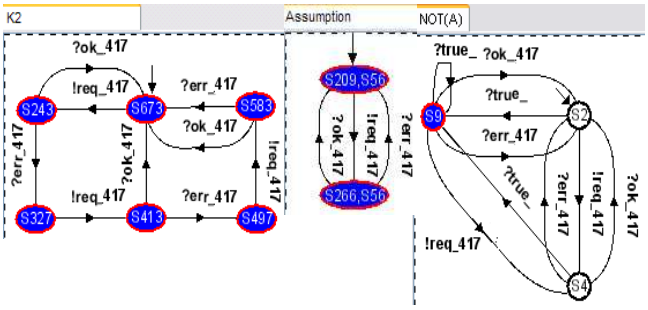

Figure 6: Büchi Automata of K''_{417}, $K_1[f^1_{417}][f_{env}]$ and $\overline{K_1[f^1_{417}][f_{env}]}$

In general, a formula $\langle true \rangle M \langle P \rangle$ means $M \models P$. While a formula $\langle A \rangle M \langle P \rangle$ means if A holds then $M \models P$. In our context, P is modeled as the corresponding Büchi Automaton B_P. M is modeled as the corresponding LTS. By referring to [5], to a LTS M always corresponds a Büchi Automaton B_M. With $L(B)$ we denote the language accepted by B. Exploiting the usual automata-based model checking approach [5], to check if $M \models P$ we first automatically build the product language $L_{M \cap P} \equiv L(B_M) \cap \overline{L(B_P)}$ and then we check if $L_{M \cap P}$ is empty.

Theorem 1. Enhancement consistency check
Let P_i be the Büchi Automata specification of a desired behavior for a system S formed by $C_1,..,C_n$ components; let K be the deadlock-free and P_i-satisfying coordinator for the components in S; let E_i be the MSCs specification of a K-protocol enhancement; let K'' be the adaptor between K and the wrapper implementing the enhancement E_i; let δ the identifier of the channel connecting K'' with K; let $\{j,..,j+m\}$ the set of channel identifiers which are both channels in the set of channel identifiers for the action labels in P_i and in the set of channels identifiers affected by the enhancement E_i; and let f_{env} be a relabeling function in such a way that $f_{env}(?\alpha) = !\alpha$ and $f_{env}(!\alpha) = ?\alpha$ for all $\alpha \in Act_K$, if $L_{K''_\delta \cap K_{j,..,j+m}[f^{j,..,j+m}_\delta][f_{env}]} = \emptyset$ then $((K[f^{j,..,j+m}_\delta] \mid K''_\delta) \setminus Act_{K_{j,..,j+m}[f^{j,..,j+m}_\delta]}) \models P_i$ and hence E_i is consistent with respect to P_i.

PROOF. Let A be the Büchi Automaton corresponding to $K_{j,..,j+m}[f_\delta^{j,..,j+m}][f_{env}]$, if $L_{K''_\delta \cap K_{j,..,j+m}[f_\delta^{j,..,j+m}][f_{env}]} = \emptyset$ then $K''_\delta \models A$. That is, $\langle true \rangle K''_\delta \langle A \rangle$ holds. Moreover, by construction of A, $\langle A \rangle K[f_\delta^{j,..,j+m}] \langle P_i \rangle$ holds too. By applying the inference rule of the *assume-guarantee paradigm*, $\langle true \rangle ((K[f_\delta^{j,..,j+m}] \mid K''_\delta) \backslash Act_{K_{j,..,j+m}[f_\delta^{j,..,j+m}]}) \langle P_i \rangle$ is true and hence $((K[f_\delta^{j,..,j+m}] \mid K''_\delta) \setminus Act_{K_{j,..,j+m}[f_\delta^{j,..,j+m}]}) \models P_i$. $\square$

By referring to Theorem 1, to check if E_i is consistent with respect to P_i, it is enough to check if $\langle true \rangle K''_\delta \langle A \rangle$ holds. In other words, it is enough to check if K'' provides K with the environment it expects (to still satisfy P_i) on the channel connecting K'' to K (i.e.: the connector identified by δ). In the example illustrated in Section 5, *RETRY* is consistent with respect to *AlternatingProtocol*. In fact, by referring to Figure 6, $NOT(A)$ is the Büchi Automaton for $\overline{K_1[f^1_{417}][f_{env}]}$ (i.e.: for $\overline{A}$ of Theorem 1) and $K2$ is the Büchi Automaton for K''_{417} (i.e.: for K''_δ of Theorem 1). By automatically building the product language between the languages accepted by $K2$ and $NOT(A)$, *SYNTHESIS* concludes that $L_{K''_{417} \cap K_1[f^1_{417}][f_{env}]} = \emptyset$ and hence that $((K[f^1_{417}] \mid K''_{417}) \setminus Act_{K_1[f^1_{417}]}) \models$ *AlternatingProtocol*. That is *RETRY* is consistent with respect to *AlternatingProtocol*.

7. CONCLUSION AND FUTURE WORK

In this paper, we combined the approaches of protocol transformation formalization [13] and of automatic coordinator synthesis [6, 9, 11] to produce a new technique for automatically synthesizing failure-free coordinators for protocol enhanced in component-based systems. The two approaches take advantage of each other: while the approach of protocol transformations formalization adds compose-ability to the automatic coordinator synthesis approach, the latter adds automation to the former. This paper is a revisited and extended version of [7]. With respect to [7], the novel aspects of this work are that we have definitively fixed and extended the formalization of the approach, we have implemented it in our *"SYNTHESIS"* tool and we have formalized and implemented the enhancement consistency check.

The key results are: (i) the extended approach 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) we are 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 [13]. The automation and applicability of both the old (presented in [6, 9, 11]) and the extended (presented in this paper and in [15]) approach for synthesizing coordinators is supported by our tool called *"SYNTHESIS"* [9, 15].

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 different than the case-study treated in [15] which represents the first attempt to apply the extended version of *"SYNTHESIS"* (formalized in this paper) in real-scale contexts.

8. REFERENCES

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

[2] M. Autili. Sintesi automatica di connettori per protocolli di comunicazione evoluti. Tesi di laurea in Informatica, Universitá dell'Aquila - April,2004 - http://www.di.univaq.it/tivoli/AutiliThesis.pdf.

[3] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

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

[5] O. G. Edmund M. Clarke, Jr. and D. A. Peled. *Model Checking*. The MIT Press, 2001.

[6] 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, LNCS 2804, Sept. 2003.

[7] M.Autili, P.Inverardi, and M.Tivoli. Automatic adaptor synthesis for protocol transformation. In *WCAT04.*

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

[9] M.Tivoli, P.Inverardi, V.Presutti, A.Forghieri, and M.Sebastianis. Correct components assembly for a product data management cooperative system. In *proceedings of the Int. Symposium CBSE7. May,2004.* Springer, LNCS 3054.

[10] R. Passerone, L. de Alfaro, T. Heinzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proc. of ICCAD, 2002.*

[11] P.Inverardi and M.Tivoli. Failure-free connector synthesis for correct components assembly. In *Proceedings of SAVCBS'03.*

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

[13] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *proceeding of the 25th ICSE'03 - Portland, OG (USA)*, May 2003.

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

[15] M. Tivoli, M. Autili, and P. Inverardi. Synthesis: a tool for synthesizing correct and protocol-enhanced adaptors. submitted for publication - Aug,2004 - http://www.di.univaq.it/tivoli/LastSynthesis.pdf.

[16] M. Tivoli and D. Garlan. Coordinator synthesis for reliability enhancement in component-based systems. Carnegie Mellon University, C.S.Dep. - Tech.Rep. - http://www.di.univaq.it/tivoli/CMUtechrep.pdf.

[17] 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.

[18] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, march 1997.