The Journal of Systems and Software 81 (2008) 2210–2236

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems ☆

Marco Autili [a], Leonardo Mostarda [c], Alfredo Navarra [b], Massimo Tivoli [a,*]

[a] Dipartimento di Informatica, Università dell'Aquila, Via Coppito, I-67100 L'Aquila, Italy
[b] Dipartimento di Matematica e Informatica, Università degli Studi di Perugia, Via Vanvitelli 1, I-06123 Perugia, Italy
[c] Department of Computing, Imperial College London, South Kensington Campus, London SW7 2AZ, United Kingdom

ARTICLE INFO

Article history:
Received 14 February 2007
Received in revised form 1 April 2008
Accepted 4 April 2008
Available online 12 April 2008

Keywords:
Software architecture
Component-based software engineering
Component assembly
Component adaptation

ABSTRACT

Building a distributed system from third-party components introduces a set of problems, mainly related to compatibility and communication. Our existing approach to solve such problems is to build a *centralized adaptor* which restricts the system's behavior to exhibit only *deadlock-free* and *desired interactions*. However, in a distributed environment such an approach is not always suitable. In this paper, we show how to automatically generate a *distributed adaptor* for a set of black-box components. First, by taking into account a specification of the interaction behavior of each component, we synthesize a behavioral model for a centralized *glue adaptor*. Second, from the synthesized adaptor model and a specification of the desired behavior that must be enforced, we generate one *local adaptor* for each component. The local adaptors cooperatively behave as the centralized one restricted with respect to the specified desired interactions.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

Reuse-based software engineering is becoming one of the main development approaches for business and commercial systems. Nowadays, a growing number of software systems are built as a composition of reusable or COTS (*Commercial-Off-The-Shelf*) components and CBSE is a reuse-based approach which addresses the development of such systems.

In an ideal world, component-based systems are assembled by simply connecting together compatible ready-to-use components,[1] that jointly provide the desired functionalities. However, in the practice of software development it turns out that the constituent components often do not perfectly fit together and adaptation is needed to eliminate the resulting mismatches (Becker et al., 2006; Yakimovich et al., 1999; Szyperski, 2004; Horwich, 1990; Yellin and Strom, 1997, 2002; Zaremski and Wing, 1995; Schmidt and Reussner, 2002; Becker et al., 2004). In particular, considering third-party and black-box components makes the problem worse since there is no way to inspect the source code for possibly solving mismatches from inside. In this setting, while assembling a distributed system from a set of black-box components interacting by message passing, the specific problem we want to face concerns how to automatically prevent deadlocking and undesired (externally observable) interactions of the resulting system. A widely used technique to deal with this problem is to use adaptors and interpose them among the components that are being assembled to form the system. The intent of the adaptors is to moderate the external communication of the components in a way that the resulting system is deadlock-free and complies with a desired behavior (*i.e.*, desired sequences of messages exchanged among the components).

Our previous approach (Inverardi and Tivoli, 2003) (implemented in the previous version of our SYNTHESIS tool (Tivoli and Autili, 2006)) is to build a *centralized adaptor* which restricts the system's behavior to exhibit only a set of *deadlock-free* or *desired interactions*. By exploiting an *abstract* and *partial* specification of the global behavior that must be enforced, SYNTHESIS automatically builds such an adaptor. It mediates the interaction among the components by allowing only the desired behavior specified by the assembler (*i.e.*, the SYNTHESIS user) and, simultaneously, avoiding possible deadlocks.

In a distributed environment it is not always possible or convenient to introduce a centralized adaptor. For example, existing distributed systems might not allow the introduction of an additional component (*i.e.*, the adaptor) which coordinates the information flow in a centralized way. Moreover, the coordination of several components might cause loss of information and bottlenecks,

☆ This paper is a revised and extended version of Autili et al. (2006) that has been presented at EWSA2006.

* Corresponding author.
E-mail addresses: marco.autili@di.univaq.it (M. Autili), lmostard@doc.ic.ac.uk (L. Mostarda), navarra@dipmat.unipg.it (A. Navarra), tivoli@di.univaq.it (M. Tivoli).

[1] Hereafter the terms *component* and *component instance* are used interchangeably.

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.04.006

hence slowing down the response time of the centralized adaptor. Conversely, building a distributed adaptor might extend the applicability of the approach to large-scale contexts.

In this paper, we describe our novel approach to the automatic generation of a *distributed adaptor* for a set of black-box components. Given (i) a Labeled Transition System (LTS) (Keller, 1976) specification of the *interaction behavior* (based on message passing[2]) of each component with its "expected environment"[3] and (ii) an LTS-based specification of the *desired behavior* that the system to be composed must exhibit, our approach generates *component local adaptors* (one for each component). These local adaptors suitably communicate in order to avoid possible deadlocks and to enforce the specified desired interaction behavior. They constitute the distributed adaptor for the given set of black-box components.

In Tivoli and Autili (2006) (and references therein), we have shown how it is possible to automatically derive LTS behavioral descriptions by assuming a partial specification of the system to be assembled. In particular, we give a partial specification of the interaction behavior of each component in the form of a *basic Message Sequence Chart* (bMSC) and *high-level MSC* (hMSC) specification (Uchitel et al., 2004; ITU Telecommunication Standardisation Sector, 1996). By applying our implementation of the algorithm described in Uchitel et al. (2004), the partial specification of each component is automatically translated into the corresponding LTS specification. hMSC and bMSC specifications are useful as an input language, since they are commonly used in software development practice. Thus, LTSs can be regarded as an internal specification language.

Starting from the specification of the components' interaction behavior, our approach synthesizes a behavioral model (*i.e.*, an LTS) of a centralized *glue adaptor*. At this stage, the adaptor LTS is built only for modeling, by interleaving, all the possible (externally observable) interactions considering synchronization on common actions, *i.e.*, the send event for a message and the corresponding receive event. It models a dummy routing component and each message it receives is forwarded strictly to the right component.

By taking into account the specification of the desired behavior that the composed system must exhibit, our approach explores the centralized glue adaptor model in order to find those states leading to deadlocks or to undesired behaviors. This process is used to automatically derive the actual code for the set of local adaptors that implement the *correct*[4] and *distributed* version of the centralized adaptor model. It is worth mentioning that the construction of the centralized glue adaptor model is required to deal with deadlocks in a fully-automatic way. Otherwise, in order to avoid the construction of the centralized adaptor, we should make the stronger assumption that the specification of the desired behavior itself ensures deadlock-freeness and it is consistent with respect to the centralized glue adaptor (*i.e.*, the desired behavior can be enforced against the glue adaptor).

The approach presented in this paper has various advantages with respect to the one described in Tivoli and Autili (2006) and Inverardi and Tivoli (2003) concerning the synthesis of centralized adaptors. The most relevant ones are

- no centralized point of information flow exists;
- the degree of parallelism of the system without the adaptor is maintained. Conversely, the approach in Tivoli and Autili (2006) does not permit parallelism since the adaptor is centralized, single-threaded and the communication with it is synchronous;
- all the domain-specific deployment constraints imposed on the adaptor can be removed. In Tivoli and Autili (2006), we applied the synthesis of centralized adaptors to COM/DCOM applications. In this domain, the centralized adaptor and the server components had to be deployed on the same machine. Now, the approach described in this paper allows one to deploy each component (together with its local adaptor) on different machines.

The SYNTHESIS tool has been extended accordingly in order to enable also the distributed implementation of the generated adaptor model. The distributed adaptor is implemented as a set of EJB component wrappers (Autili et al., 2007). Each wrapper is developed by using AspectJ that easily supports the wrapper tasks of intercepting the component messages and correctly coordinating them. Note that AspectJ is only one possible implementation choice.

2. Background notions

This section provides the reader with background concepts, definitions and assumptions needed for a full understanding of our work. Actually, the discussion has been kept as light as possible in order to give a good intuition to the reader without loosing his/her attention. Detailed formalisms and definitions are then referred to Appendixes A and B.

In our context, a distributed system is a network of interacting black-box and ready-to-use components $C = \{C_1, \ldots, C_n\}$ that can be simultaneously executed. Components communicate by message passing. Messages are exchanged by means of communication channels, performing precise communication protocols that specify (in some formalism) the set of all possible message sequences. Note that, dealing with black-box components, communication protocols specify *external* communication among components by the relatively simple nature of the message exchange (and hence by means of send and receive events) rather than *internal* computation within a component. Generally speaking, communication channels can be

- *asynchronous* – no synchronization points exist and message passing never blocks the sender. This implies a potentially unbounded buffer; in practice, a bounded buffer is used and the sender will block when the buffer is full. In this way, a higher degree of parallelism can be achieved because (possibly) the sender never has to wait.
- *synchronous* – message passing uses no buffer and, due to synchronization points, both senders and receivers can block. The term rendezvous is often used to evoke the image of two processes that have to meet at a specific synchronization point.

For the purposes of this work, we model component interaction by assuming that the components to be assembled communicate by means of synchronous communication channels. This is not a limitation since, in practice, by introducing a finite buffer component to decouple message passing, we can simulate a bounded asynchronous system with a synchronous one (Uchitel et al., 2004; Milner, 1989). Obviously, in this case, there is the necessity of explicitly programming the needed buffers by exploiting the native primitives (of the programming language being used) that are provided to support the synchronous communication. It is well known that reasoning (*e.g.*, deadlock prevention) with the presence of unbounded buffers is undecidable (Brand and Zafiropulo, 1983). From a practical point of view, this motivates the reasonable restriction to consider only synchronous systems (or, possibly, bounded asynchronous ones).

[2] Message exchanging can be used for delivering packages of data or for calling remote procedures.

[3] Dealing with third-party and ready-to-use components, the expected environment is actualized at assembly time by the set of all the other components that are being assembled to form the system.

[4] With respect to deadlock-freeness and the specified desired behavior.

In this work, we also consider both *stateful* and *stateless* components characterized as follows:

- *stateful* – The internal state of a component instance consists of the values of its instance variables. In a stateful component, the instance variables represent the internal state of a unique component-client session. Because the client interacts ("talks") with its component instance, this state is often called the conversational state since it is retained for the duration of the component-client session. If the client removes the component instance from the memory or terminates, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the component ends there is no need to retain the state.
- *stateless* – A stateless component does not maintain a conversational state for the client. When a client invokes the method of a stateless component, the component instance variables may contain an internal state, but only for the duration of the invocation. When the method is finished, the internal state is no longer retained.

Recalling that we promote the use of additional components, called local adaptors (hereafter also referred as *wrappers*), to be interposed among the system components for mediating their interaction, we distinguish between (i) *standard* and (ii) *additional* (*external*) *communication*. The former denotes the messages that system components can exchange for data package delivering or for remote procedure call; the latter denotes the additional messages that the local wrappers exchange in order to coordinate each other. In fact, due to synchronous communication, by letting the components interact in an uncontrolled way (*i.e.*, without adaptors), they might perform deadlocking or undesired interactions. To overcome this problem, local wrappers (through a wrapping and forwarding mechanism) will intercept the components' standard communication and mediate it by exchanging additional communication, when needed.

3. Problem description

The problem we want to treat can be phrased as follows: *Given a set of interacting black-box components C and a desired behavior P, automatically derive (when possible) a deadlock-free and distributed adaptor A that, after we assemble the components in C, conforms to P.*

The basic ingredients of this problem are: (i) the nature of the components we are considering, (ii) the type of desired behavior we want to check, and (iii) the type of systems we want to build. We consider truly *black-box* components and, hence, the source code of the component is not available. For now, a desired behavior *P* is a functional property expressing precise ways to coordinate the interaction behavior of the components that are being assembled to form the system. The architecture of the system, assembled by means of the distributed adaptor *A*, is constrained by the architectural style we consider. It defines the rules used to build the composed system and it is called DABA (*i.e.*, *Distributed Adaptor-Based Architecture*) style (defined in Section 3.1).

Besides assuming that the system architecture must reflect the rules of the DABA style, we also assume that a behavioral specification of each component is provided in the form of an LTS. Thus, when we say: *Given a set of interacting black-box components C* ... in the problem definition we mean that we consider a set of component behavioral specifications *C* (*i.e.*, a set of LTSs) that describe the standard (external) communication protocol. As already said in Section 1 it is possible to automatically derive these LTS descriptions by assuming a MSC-based specification of the components to be assembled. LTSs give a trace-based semantics of the interaction behaviour of the components with the external environment (see Appendixes A.1 and A.2). Informally, a transition of an LTS is labeled with a component message (send or receive) and a state of the LTS models a "logical" state of the component. This logical state represents a certain point of the component interaction in which the component has exchanged a message (*i.e.*, one of the messages labeling the incoming transitions) and is ready to exchange some other messages (the ones labeling the outgoing transitions). Note that, in this sense, the state of an LTS does not model the internal actual state of the component, *i.e.*, values of its instance variables.

Informally, our approach is the following. The method starts with a set of components, and builds a centralized *glue adaptor* following the reference style constraints. The glue adaptor models all possible component interactions and each received message is forwarded to the right component directly, hence serving as a *dummy* router[5] for the component interaction. Then deadlock-freeness and desired behavior analysis is performed. If the synthesized glue adaptor contains deadlocking and desired behavior violating interactions, a prevention strategy is applied. Depending on the specified desired behavior, the analysis of only the centralized adaptor is enough to automatically distribute it in a set of component wrappers (each of them local to each component), hence enforcing the only system interactions that are deadlock-free and do not violate the desired behavior.

Note that, in a first phase, our approach restricts the set of all possible composed system behaviors in order to keep only those component interactions that are deadlock-free. In doing so, it cannot be sure that those component interactions that are actually needed for the overall purpose of the system are still kept. The desired behavior analysis takes care of performing this check. That is, the desired behavior is an LTS specifying the only standard communication that all the interacting components should perform to realize the purposes of the resulting system. The desired behavior enforcing mechanism further restricts the behavior of the deadlock-free composed system in order to avoid the component interactions that violate the specified desired behavior (and, hence, those component interactions that are not required for the overall purpose of the system). It might be the case that, by taking into account the set of components given as input to our method, it is not possible to assemble a distributed and deadlock-free system that, in the same time, satisfies also the specified desired behavior. This can be due to the fact that the specified desired behavior contains interactions that either do not belong to all the possible component interactions or have been prevented by the deadlock analysis. Within our method, this condition is checked by means of a trace containment check (Milner, 1989) between the desired behavior LTS and the adaptor LTS where the deadlocking interactions have been removed (see also Section 5.1). In the case the check answers that all the traces of the desired behavior LTS are not contained in the deadlock-free traces of the adaptor LTS, since we are dealing with black-box components, there is nothing to do and our method answers to the user with an unsuccessful output. Otherwise, our method will synthesize a deadlock-free and distributed adaptor that permits only the component interactions specified through the desired behavior and that are the ones required for the purposes of the system (we recall that the adaptor is implemented as a set of component wrappers). In fact, the *correct* adaptor (with respect to the deadlock-freeness and the specified desired behavior) has not to necessarily let the components perform all their

[5] A router that does not apply any particular routing logic except for the simple reception/forwarding of component messages.

possible interactions but only the ones that are needed for the system's purposes.

Indeed, one cannot assume that the actual code of a (black-box) component has been developed in a way that it is always possible to disable/discard a component action by the external environment. Actually it can be done only if the developer had preemptively foreseen it and, for instance, an exception handling logic was aptly coded for such an action. Thus we would need to distinguish between *controllable* and *uncontrollable* actions. In other words, we should distinguish between component actions that can be discarded by the external environment (*e.g.*, the adaptor) and component actions that cannot be discarded. For example, inputs coming from a sensor are often considered as uncontrollable since they must be accepted and treated by the component. In contrast, controllable actions can be safely discarded, for instance to correctly prevent a deadlock. As it is usually done in the *discrete controller synthesis* research area (Ramadge and Wonham, 1987; Brandin and Wonham, 1994), the developer is in charge of specifying which component actions are controllable and which are uncontrollable. Therefore, for the purposes of our method, we should assume that the component developer specifies this kind of information, *e.g.*, by tagging, within a component's LTS, action labels as controllable or uncontrollable. In previous work from some of the authors (Tivoli et al., 2007), we applied this approach to the automatic synthesis of centralized adaptors for real-time components. Since in this paper, we mainly focus on the automatic distribution of the centralized adaptor (and this is the novel contribution with respect to our previous work), for the sake of simplicity, we avoid to address controllability issues and we assume that all component actions are controllable. This is not a limitation of the work presented in this paper since, as briefly discussed in Section 5.1, its extension to account for controllability issues is straightforward.

3.1. The reference architectural style

In this section, we define the reference architectural style that represents the starting point of our work. This style imposes constraints on the architecture of the system to be assembled that allow us to automatically derive, from a set of component behavior specifications, a behavioral model of the centralized adaptor. As we will see in Section 4, this model plays a key role in synthesizing the deadlock-free and distributed adaptor in a way that it does not violate the specified desired behavior.

Within this architectural style, we consider three kinds of system configurations concerning with the use of: (i) no adaptor, (ii) a centralized adaptor, or (iii) a distributed adaptor. As already said, the aim of our approach is to automatically derive, from a set of black-box components (communicating by exchanging messages), the code that implements new additional components to be inserted in the composed system. These new components implement wrappers. A wrapper mediates the interaction between the supervised component and the other ones in order to prevent possible deadlocks and/or undesired behaviors. To this aim, we distinguish two kinds of components: *functional components* and *adaptors*. Functional components implement the system's functionality, and are the primary computational constituents of a system (black-box components typically implemented by third-parties). They perform only standard communication. Adaptors, on the other hand, simply route messages and each message they receive is forwarded strictly to the right component. They intercept the standard communication and mediate it by exchanging additional communication (see Section 2). We make this distinction in order to clearly separate components that are responsible for the functional behavior of a system and additional components that are introduced to aid the integration/communication behavior.

Hereafter, we will refer to a system as an *Adaptor Free Architecture* (AFA) if it is defined without any adaptor; a system in which a centralized adaptor appears is termed *Centralized Adaptor-Based Architecture* (CABA); conversely, a system in which a distributed adaptor appears as a set of local wrappers (one for each functional component) is termed *Distributed Adaptor-Based Architecture* (DABA). The respective definitions follow:

Definition 1 (*AFA*). An *Adaptor Free Architecture* (AFA) is a set of components directly connected, through connectors, in a synchronous way.

Definition 2 (*CABA*). A *Centralized Adaptor-Based Architecture* (CABA) is a set of components directly connected to one centralized adaptor, through connectors, in a synchronous way.

Definition 3 (*DABA*). A *Distributed Adaptor-Based Architecture* (DABA) is a set of components each of them directly connected to its component wrapper, through connectors, in a synchronous way. Each wrapper is connected to all other wrappers, through connectors, in an asynchronous way.

Fig. 1a illustrates an AFA, Fig. 1b and c its corresponding CABA and DABA, respectively. $C1$, $C2$, $C3$ and $C4$ are instances of functional components. A is a centralized adaptor. $w1$, $w2$, $w3$ and $w4$ are local wrappers (forming the distributed version of A). The communication channels denoted as lines between components are connectors (*e.g.*, ORB for CORBA, RPC for COM+ and RMI for EJB).

Since we are considering a synchronous communication among functional components, the send event for a message and the corresponding receive event, that synchronize two components, are considered to be blocking events.

Thus, the behavior of a component is specified as an LTS and, as defined in Appendix A.3, the system configuration is specified by using the LTS parallel composition operator. Send and receive events of a component are also referred as output and input actions of the component LTS, respectively. To define the behavior of a *composition* of components, we simply place in parallel the LTS descriptions of those components. The parallel composition operator combines the behaviors of LTSs by synchronizing their *shared/common actions* and *interleaving* their *non-shared* ones. In this way, we force the components (in the parallel composition) to synchronize only on "complementary" common actions. In other words, if a component C_i, in the parallel composition, sends a message m then its only way to progress is to synchronize with some component C_j ($i \neq j$), which receives m. This gives an AFA for a set of components (see Appendix A.4).

Given an AFA for a set of components, we can also produce a corresponding CABA for these components by automatically deriving and interposing a centralized glue adaptor among communicating components. The adaptor at this point simply routes messages (see Appendix A.5). Informally, the trace-based semantic of the CABA-system behavior is achieved through the parallel composition operator applied to the set of component LTSs and the adaptor LTS. In Inverardi and Tivoli (2003), the correctness of this derivation step of the synthesis approach is proved by showing that the AFA-system can be simulated by the synthesized

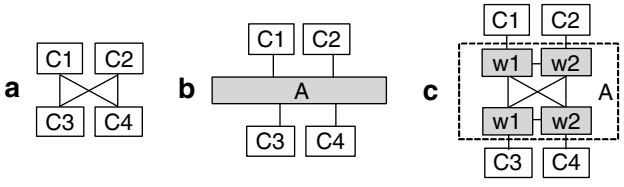


Fig. 1. A sample of (a) an AFA, (b) a CABA, and (c) a DABA.

CABA-system under a suitable notion of "state based" equivalence called CB-Simulation. The starting point of CB-Simulation is the stuttering equivalence (Nicola and Vaandrager, 1995). In Inverardi and Tivoli (2003), the completeness of the synthesis approach is also proved by showing that the centralized glue adaptor does not introduce in the system any new logic. As we will see later, the centralized glue adaptor will play a key role in synthesizing the set of component wrappers that implement the distributed adaptor. It restricts, under controllability (see Section 3), the system interactions to a subset of deadlock-free and (specified) desired interactions.

In the reminder of this section, we introduce the deadlock problem in a component-based setting and the desired behavior specification notation.

3.2. Deadlock and desired behavior modeling

In our context, the deadlock is the base failure because it is directly identifiable in the behavioral model of the synthesized glue adaptor. That is, we distinguish the prevention of deadlocks and of undesired behaviors. However, as already said in Sections 1 and 2, we provide the user with the option of specifying deadlock-freeness directly with the desired behavior specification, hence avoiding the synthesis of the glue adaptor model. In spite of this, we maintain a special handling of deadlock-freeness because we do not want to force the user to provide such a specification. This is a reasonable choice since, for large systems, deadlocks are very often unpredictable and, hence, automating the process for their detection and prevention is required in order not to involve the user in the process.

We give the following definition to describe the deadlock problem in a component-based context.

Definition 4 (*Deadlock*). A set C of components is *deadlocked* if each component in C is waiting for an event that only a different component in C can cause.

Informally we can say that in component-based architectures, there are two types of deadlock problems:

- *observable deadlocks*;
- *hidden deadlocks*.

For both kinds of deadlocks, the behavior of a component is wrong with respect to the behavior of its actual environment (*i.e.*, the assembly made by all other components in the system) although the component behavior might be correct with respect to the "stand-alone" context represented by the invariants assumed at development time. Specifically, both kinds of deadlocks occur during the interaction between a component and its environment. For the first kind of deadlock, the failure is an event that is observable by the component environment. For the second kind, the failure is an externally non-observable event since it might depend on internal characteristics of the component.

Thus, observable deadlocks can be treated in the component setting by operating on the architectural context – namely on the glue adaptor; the hidden deadlocks cannot be automatically addressed. The only way to solve hidden deadlocks is to modify the internal behavior of a component but this is not possible when dealing with black-box components. An example of a hidden deadlock type is offered by the Compressing Proxy problem (Compare et al., 1999). Thus, we focus only on the first class of problems, attempting to create adaptors that can prevent observable deadlocks. In the remainder of this paper, we use the term deadlock to mean an observable deadlock.

Note that observable deadlocks can occur not only when dealing with stateful components but also with stateless ones (see Section 2). For instance, consider a client that invokes the method m of a stateless component $C1$. Although the component-client session is kept only for the duration of m, from within m, $C1$ can in turn invoke a method of another component $C2$. The interaction between $C1$ and $C2$ might deadlock, hence blocking the client as well.

By referring to Section 3.1, in our setting, a deadlock in the AFA-system (*i.e.*, the component LTSs parallel composition) is directly identifiable as a *sink state* (*i.e.*, a deadlock state – see Appendix A.6) of the centralized glue adaptor LTS in the corresponding CABA-system (*i.e.*, the parallel composition of the component LTSs plus the adaptor LTS).

As already mentioned, our approach also tackles the analysis of failures beyond deadlock by exploiting a specification of the interactions that are needed for the overall purpose of the system. As previously done, we refer to such a set of needed interactions as the desired behavior since it represents, among all possible interactions of the AFA-system, the ones that we wish the resulting DABA-system will satisfy exclusively.

We recall that a desired behavior is specified in terms of an LTS. In particular, for the purposes of our method, this LTS has an enriched syntax for the action labels expressive enough to model a:

- *regular* action, *i.e.*, one specific component action;
- *negative* action, *i.e.*, all possible actions (of all the components) but one;
- *universal* action, *i.e.*, all possible actions (of all the components);
- *logical "AND"* of negative actions, *i.e.*, all possible actions but the ones within the "AND" operator;
- *logical "OR"* of regular actions, *i.e.*, at least one action among the ones within the "OR" operator.

By exploiting this enriched syntax, the desired behavior LTS allows the user of *SYNTHESIS* to abstract from irrelevant details and easily specify an high-level model of the behavioral requirements of the resulting DABA-system that is being assembled. In Appendix A.7, we formally define the enriched LTS syntax, and in Section 4 we discuss a simple explanatory example for introducing the kind of problem that our method aims at preventing and for showing how the above syntax is used for specifying a desired behavior LTS. Obviously, the user can specify a desired behavior that is always violated by the composed system. In this case, our tool will answer to the user that this behavior cannot be satisfied and, probably, something in the specification has to be changed.

4. Explanatory example

In Fig. 2, we show the component LTSs of the AFA-system for our explanatory example. The system is composed by three components: a server `C1`, and two clients `C2` and `C3`.

For the transition labels, we denote with, *e.g.*, `?C1.a` (resp., `!C1.a`) the receive (resp., send) message `C1.a`. The state with the incoming arrow is the initial *logical state* (*i.e.*, the state `S0` in each component LTS). Referring to Section 3, we recall that a logical state does not model the internal state of a component in terms of values of its instance variables, but it models a certain point of the component external interaction in which the component is ready to receive/send certain specified messages. For instance, from the state `S0`, the component `C1` is ready to receive the requests `C1.a` and `C1.c` (*i.e.*, `?C1.a` and `?C1.c`, respectively), and, from the state `S1`, it is ready to send the notifications `C1.b` (`!C1.b`).

We recall that we model the system behavior by composing in parallel the component LTSs (see Appendixes A.3 and A.4) and forcing synchronization on common send/receive messages. For example, `C2` can synchronize with `C1` on messages `C1.a` and `C1.b`.

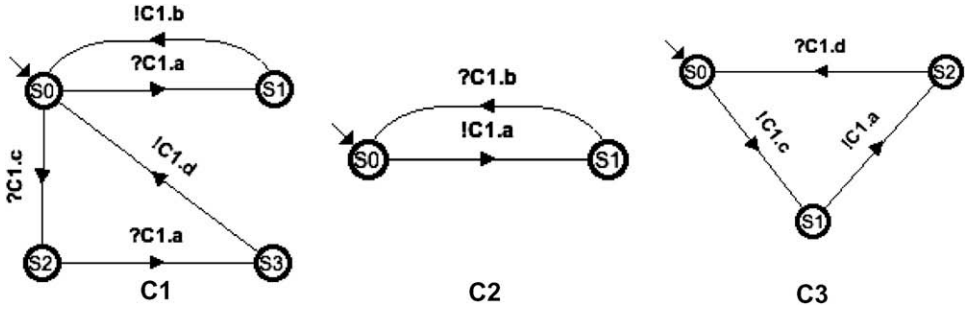


Fig. 2. LTSs of the components forming the AFA-system.

Indeed, since we want to deal with either reusable black-box components or COTS components, there might not be a direct syntactical correspondence between the action labels used by the different component LTSs. In general, this kind of mismatch cannot be solved automatically and it requires to develop (by hand) component wrappers solving that syntactical mismatch, as done in the work described in Tivoli and Autili (2006) and Autili et al. (2004). Since in this work we are focusing on automatically preventing interaction protocol mismatches, we consider this problem out of the scope of this paper and, hereafter, we will assume that the component interfaces syntactically match, either because they already match or because suitable component wrappers have been previously developed by the system assembler (*i.e.*, a possible user of the *SYNTHESIS* tool).

Proceeding with the description explanatory example, in the AFA-system a deadlock occurs whenever `C2` sends the message `C1.a` after that `C3` has previously sent the message `C1.c`. Actually, in this case, `C1` reaches the state `S3` in which it would send `C1.d` but there is no other component ready to receive it, hence blocking the execution of `C1`. On the other hand, `C2` is waiting to receive `C1.b` in the state `S1` and `C3` is waiting to send `C1.a` in the state `S1` but there is no other component either sending `C1.b` or receiving `C1.a`, hence blocking the execution of `C2` and `C3`. Thus, the *global state* ⟨`S3`, `S1`, `S1`⟩ of the AFA-system is a deadlock state (note that, `S3`, `S1`, `S1` are *local states* of `C1`, `C2` and `C3`, respectively).

In Fig. 3, we show the LTS of the centralized glue adaptor for `C1`, `C2`, and `C3`. Unlike action labels in a component LTS, each label carries a post-fixed identifier specifying which component performs that action. For instance, `?C1.a_2` models the action `?C1.a` performed by the component `C2`.

The centralized glue adaptor is synthesized to model (by interleaving) all the components' interaction in the AFA-system. It models a simple routing component and each message it receives is forwarded strictly to the right component. For instance, looking at the trace from the state `S0` to the state `S2` shown in Fig. 3, in the CABA-system, the adaptor synchronizes with `C2` by receiving the message `C1.a` (*i.e.*, the action label `?C1.a_2`), and then synchronizes with `C1` by forwarding `C1.a` (!*i.e.*, `C1.a_1`). The state `S1` models an "intermediary" logical state of the glue adaptor while the latter is forwarding the message `C1.a`. This means that the adaptor performs strictly sequential I/O behavior (see Definition 13 in Appendix A.5).

In Fig. 3, the deadlock has been detected as the filled sink state `S7` (of the glue adaptor) that encodes the global state ⟨`S3`, `S1`, `S1`⟩ of the AFA-system (see Definition 13 in Appendix A.5 and Definition 15 in Appendix A.6). That is, synthesizing the glue adaptor allowed us to detect a possible deadlock in the components' interaction. Note that in the corresponding CABA-system the deadlock occurs as well as in the AFA-system (since, by construction, the deadlock has been reflected also in the model of the glue adaptor). The LTS of the centralized glue adaptor and the desired behavior LTS shown in Fig. 4 (*i.e.*, *P*) are taken into account in order to automatically synthesize a distributed adaptor *A* that, in the corresponding DABA-system, will prevent the deadlock and make the components interact by following only the interactions specified by the LTS of *P*.

The syntax and the meaning of the transition labels of *P* is the same as the one used for the glue adaptor LTS except for two kinds of actions: (i) a universal action (*i.e.*, ?*true_* or, equivalently, *true_*) that represents any possible action, and (ii) a negative action (*e.g.*, $!-C1.a\_2$ or $?-C1.a\_2$) that represents any possible action except for the negative action itself (*e.g.*, all the possible component actions but $!\,C1.a\_2$ or $?C1.a\_2$, respectively). Moreover, it is possible to label transitions in the LTS of a desired behavior through the boolean formulas built as either "*OR*" or "*AND*" combination of actions (only negative actions can be operands of the "*AND*" operator, and only regular actions can be operands of the "*OR*" operator). For instance, let *a* and *b* be two regular actions, the logical "*OR*" of *a* and *b* is evaluated to true if (in the DABA-system) either *a* or *b* are performed. Let $x_1, \ldots, x_n$ be regular actions, we denote the logical "*OR*" of $x_1, \ldots, x_n$ by $[x_1, \ldots, x_n]$. Analogously, let *a* and *b* be two negative actions, the logical "*AND*" of *a* and *b* is evaluated to true if an action different from both *a* and *b* is performed. Let $x_1, \ldots, x_n$ be negative actions, we denote the logical "*AND*" of $x_1, \ldots, x_n$ by $\{x_1, \ldots, x_n\}$.

Informally, by referring to Fig. 4, *P* specifies all AFA-system behaviors that guarantee the evolution of all components in the system. It specifies that `C2` and `C3` can send `C1.a` by necessarily using an alternating coordination protocol that is initiated by `C2` (see the outgoing transition from the state `S0` labeled by

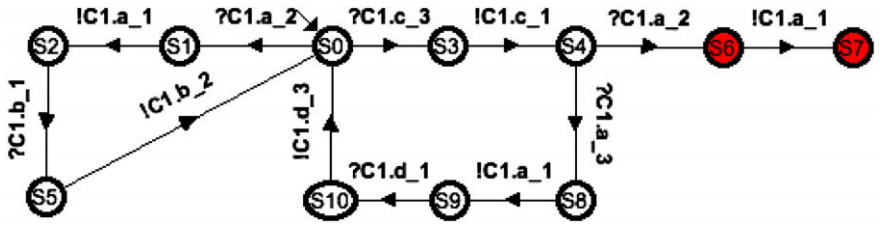


Fig. 3. LTS of the centralized glue adaptor in the CABA-system.

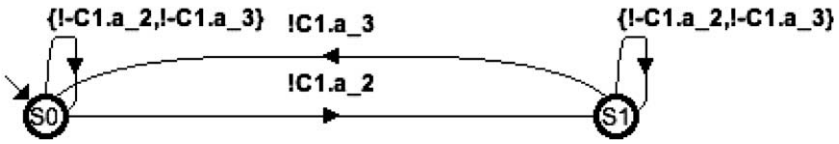


Fig. 4. LTS of the desired behavior *P*.

`! C1.a_2`). In other words, it means that if `C2` sends `C1.a` then it cannot sends `C1.a` again if `C3` has not sent `C1.a`. The distributed adaptor to be synthesized (to form the correct DABA-system) will ensure fairness by satisfying this desired behavior.

Since *P* predicates on the system global states, each node can model more then one (global) state of the DABA-system to be assembled. For instance, the state `S0` of *P* matches with `S0`, `S3`, `S4`, `S8`, `S9`, `S10` of the glue adaptor, and `S1` of *P* matches with the remaining states.

5. Method description and formalization

In this section, we start by describing our method and then we gradually formalize it by means of a detailed discussion and algorithmic descriptions. These descriptions concern the assembly-time procedures for analyzing both the centralized glue adaptor LTS and the desired behavior LTS in order to retrieve information that is required for prevention purposes. Furthermore, we describe the local wrappers run-time procedures for intercepting the components' standard communication and mediating it by exchanging additional communication, when needed. This section also provides the correctness of our approach and concludes with a brief discussion about the overhead due to additional communication.

5.1. Method description

Our method (see Fig. 5) assumes as input: (i) a behavioral specification of the AFA-system formed by interacting components. It is given as a set $\{C_1, \ldots, C_n\}$ of LTSs (one for each component). We recall that the behavior of the system is modeled by composing in parallel all the LTSs and by forcing synchronization on common events and (ii) the specification of the desired behavior that the system must exhibit. This is given in terms of an LTS, from now on denoted by P_{LTS}.

These two inputs are then processed in two main steps:

(1) by taking into account all component LTSs, we automatically derive the LTS *A* that models the behavior of a centralized glue adaptor. At this stage, *A* models all the possible component interactions and it does not apply any adaptation. In other words, *A* performs standard communication simply routing messages among the components. In this way, it represents all possible linearizations by using an interleaving semantics (see Appendix B). *A* is derived by performing an *LTS unification* algorithm. Informally, as shown in Section 4 and formalized in Appendix A, this algorithm is a kind of parallel composition of component LTSs where for each pair of LTSs that synchronizes on common send/receive actions (*e.g.*, *C2* and *C1* on *! C1.a* and *? C1.a*, respectively), two subsequential transitions (*i.e.*, a receive followed by a send transition) are produced on *A* (*e.g.*, *?C1.a_2* followed by *! C1.a_1*). It is worth recalling that each state of *A* (*i.e.*, an AFA-system global state) is a tuple $\langle S_1, \ldots, S_n \rangle$ where each S_i is a state of C_i (see Fig. 6). For instance, the state *S2* of the centralized glue adaptor, shown in Fig. 3, encodes the corresponding AFA-system global state into the tuple $\langle S1, S1, S0 \rangle$ of *C1*, *C2*, and *C3* states, respectively. Hereafter, when the current state of a component appears in a tuple representing a global state we simply say that the component "*is in*" that global state. It is worth mentioning that, in general, a component state S_i might appear in more than one state of the LTS of *A*. As introduced in Section 3, the first step terminates by checking whether enforcing P_{LTS} is possible or not. This check is implemented by a suitable notion of *refinement* (Milner, 1989). Refinement, in general, formalizes the relation between two LTSs at different level of abstractions. Refinement is usually defined as a variant of *simulation*. In this paper, we use a suitable notion of *strong simulation* (Milner, 1989) to check a refinement relation between two LTSs. For a formal description of this trace containment check, refer to Appendixes A.7 and A.8. This first step is inherited from the existing approach (Tivoli and Autili, 2006) to the synthesis of centralized adaptors. Deeply describing this step is out of the scope of this work since the novel contribution of this work mainly concerns the second step of our approach. As already mentioned in Section 1, whenever P_{LTS} ensures itself deadlock-freeness and its traces are all traces of the adaptor LTS, such a step is not required and, hence, *A* is not generated.

(2) In the second step, if *A* has been generated and it has been checked that P_{LTS} can be enforced on it, our method explores *A* looking for those states representing the *last chance* before entering an execution trace that leads to a deadlock. For instance, in Fig. 3, the state *S4* represents the last chance state before incurring in the deadlock state *S7*. This

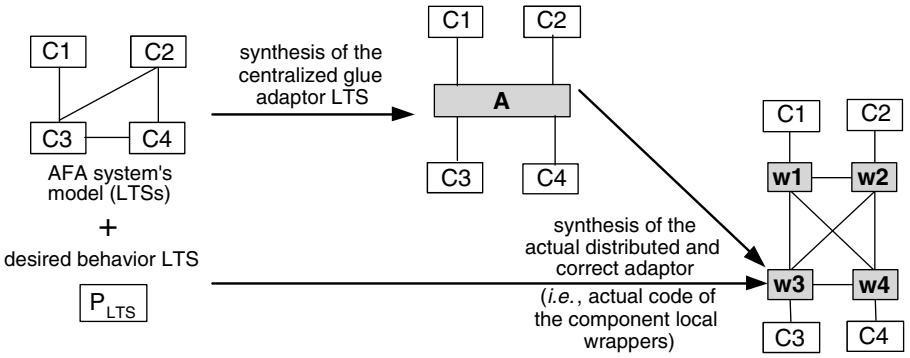


Fig. 5. Two-step method.

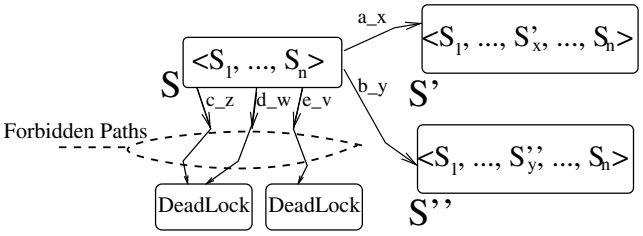


Fig. 6. A last chance node *S* of *A*.

information is crucial for deadlock prevention purposes. The search of the last chance states is realized by means of Procedure *AVisit*, see below. The aim is to save those states into the local wrappers of the components that could lead the system from a last chance state to a deadlock by means of a so-called *critical action*. The idea is therefore not to allow a component to perform a critical action before being sure that the system will not reach a deadlock state. By referring to the brief discussion about controllability issues given in Section 3, we recall that, for us, all the component actions are controllable and, hence, such a critical action can be discarded. Otherwise, if we would relax this assumption, *AVisit* should be slightly modified in order to perform a *controller synthesis step* (Ramadge and Wonham, 1987; Brandin and Wonham, 1994) that "backtracks" by looking for the first controllable action that can be discarded to prevent the execution of the critical action. The set of last chance states and associated critical actions will result modified accordingly.The second step also explores P_{LTS} to retrieve information crucial for undesired behavior prevention. The aim here is to split and distribute P_{LTS} in a way that each local wrapper knows which actions the wrapped component is *allowed* to execute. This is realized by means of Procedure *PVisit*, see below. Referring to Figs. 3 and 4 for instance, the wrapper of component *C*3 must not allow the component to send the request *C*1.*a*, if the current global state of the system matches the state *S*0 in *P*, hence enforcing the desired behavior *P*. The sets of *last chance states* and *allowed actions* are stored and, subsequently, used by the local wrappers as basis for correctly synchronizing with each other by exchanging additional communication. In other words, the local wrappers interact with each other to restrict the components' standard communication (modeled by *A*) by allowing only the part of the communication that is correct with respect to deadlock-freeness and P_{LTS}. By decentralizing *A*, the local wrappers preserve parallelism of the components forming the system.The exchanged messages among wrappers for synchronization purposes is realized by means of the two procedures *Ask* and *Ack*, see below. For now, it is sufficient to say that the first is used to ask the permission to the other wrappers before allowing a component to proceed with a critical action. The second is used to reply to a message sent by procedure *Ask* when the global state is safe. The description of how *Ask* and *Ack* allow the wrappers to correctly exchange, at run-time, synchronization communication (with respect to deadlock-freeness and the specified desired behavior) will be clear in Section 5.2.In the following section, we formalize the second step of our method. Hereafter, we assume that *A* has been generated.

5.2. Second step formalization

As described before, the second step gets as input: (i) the set $\{C_1, \ldots, C_n\}$, (ii) *A* and (iii) P_{LTS}. In order to detect deadlocks, our approach explores *A* and looks for sink states. A deadlock state (see Fig. 6) is in fact a sink of *A*. We call *Forbidden States* (FS_A) the set of deadlock states (hereafter, the terms state and node are used interchangeably) and all the ones within *forbidden traces* necessarily leading to deadlock states (see Appendix A.6). A forbidden trace in *A* is a path that starts at a node which has no transitions that can avoid a forbidden state and thus necessarily ends in a sink (see for instance Fig. 6). To prevent the system from reaching states in FS_A we need to identify a specific subset of *A*'s states that are critical with respect to FS_A (see for instance *S* in Fig. 6). In this way we can avoid storing the whole LTS at runtime as we just need to store the critical states. More precisely, in order to prevent the system from reaching a state in FS_A, we are only interested in those nodes representing the last chance just before entering a forbidden state. The last chance nodes have some outgoing edges leading to a forbidden state, the *dead* edges, and other ones, the *safe* edges (see for instance the edges labeled with *a_x* and *b_y* in Fig. 6). We denote by LC_A the set of all last chance nodes of the adaptor LTS *A* (see Appendix A.6). Note that if $LC_A = \emptyset$ and $FS_A \neq \emptyset$, it means that all the possible components' interactions are deadlocking and, hence, dealing with black-box components, there is nothing to do and our method stops answering the user with an unsuccessful output. In fact, in this case, it means that it is not possible to synthesize a deadlock-free glue adaptor for the components given as input to our method.

According to the labels of the dead edges we store in the local wrappers associated with the corresponding components the last chance nodes, and the *critical actions* that each component should not perform in order to prevent the system from reaching a state in FS_A (in Fig. 6, the action *c* is critical for the component *z*). From the implementation point of view, each local wrapper W_{C_i} uses a table $W^{LC}_{C_i}$ (*Last Chance table* of W_{C_i}) of pairs $\langle$ *last chance state of A, critical action of* $C_i\rangle$. Thus, once the whole LTS *A* has been visited, each local wrapper knows the critical actions of the corresponding component. Before a component can perform a critical action, its local wrapper has to ask for permission from the other components (see procedure *AVisit*). The last chance node table $W^{LC}_{C_x}$ for a component C_x is formally defined as follows:

$$W^{LC}_{C_x} = \{\langle S, \alpha\rangle | \exists S \in LC_A \text{ and } S' \in FS_A \text{ such that } S \xrightarrow{\alpha\_x} S' \text{ is a transition of } A\}$$

The following procedure *AVisit* computes and distributes the last chance node tables among the local wrappers.

Fig. 7 shows a possible execution of *AVisit* applied to the glue adaptor shown in Fig. 3. Informally, after the left- and right-hand side loops have been depth-first visited (see the arrowed arcs 1, 2, 3, and 4), *AVisit* backtracks to *S*4 since another branch must be visited. Note that, at this point the procedure has updated a counter referred to the number of safe branches of the visited nodes, and in particular of *S*4. From this state *AVisit* reaches the deadlock state *S*7 (see the arrowed line 5). Then *AVisit* backtracks to *S*6 and tags both it and *S*7 as forbidden states (see the arrowed line 6) since no other branches can be visited form those states. Now *AVisit* backtracks to *S*4 and tags it as a last chance state since from it either a deadlock state or safe states can be reached. Finally, *AVisit* terminates in *S*0 and the last chance tables of the local wrappers are built. The only not-empty last chance table is the one of *w*2 as shown in figure.

More precisely, *AVisit* takes as input the glue adaptor *A* that models all the possible interactions of the *n* components $(C_1, \ldots, C_n)$. The procedure makes use of the following variables: $W^{LC}_{C_i}$ is the table of last chance nodes associated to component C_i; $Flag\_Forbidden_S$ is a flag to check whether the current node *S* leads to deadlock or not in *A*; $Dead\_Son_S$ counts the number of sons of the current node *S* that lead to dead branches of *A*; $Safe\_Son_S$

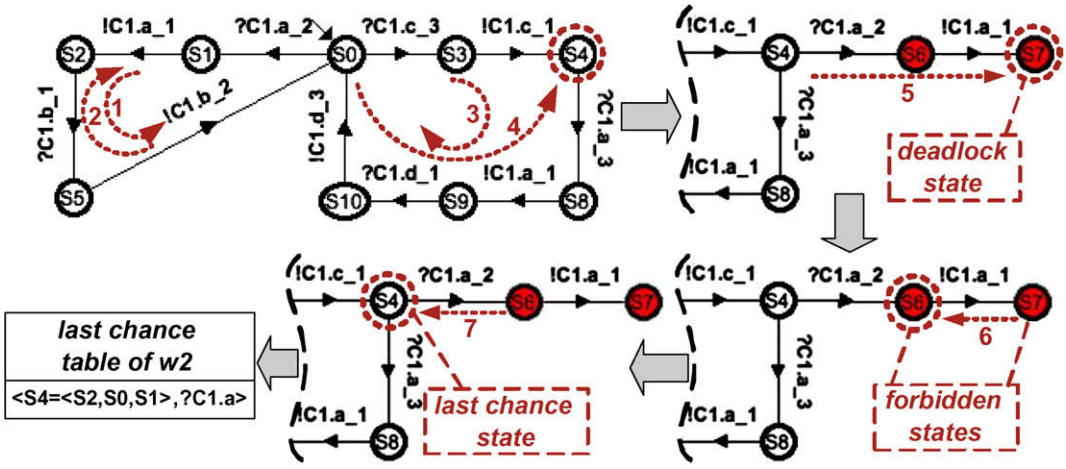


Fig. 7. Procedure *AVisit*: a possible execution.

counts the number of sons of the current node S that lead to safe branches of A.

Procedure (*AVisit(state of A*: S;)).

```
1:  for each i := 1 to n do
2:    W^LC_Ci := ∅;
3:  end for
4:  Flag_Forbidden_S := False;
5:  Dead_Son_S := 0;
6:  Safe_Son_S := 0;
7:  mark S as Visited;
8:  for each son S' of S do
9:    if the edge (S, S') is not visited then
10:     mark the edge (S, S') as Visited;
11:     if S' is not visited then
12:       AVisit (S');
13:     end if
14:     if Flag_Forbidden_S' then
15:       Dead_Son_S++;
16:     else
17:       Safe_Son_S++;
18:   end if
19:   end if
20: end for
21: if Safe_Son_S = = 0 then
22:   Flag_Forbidden_S :=True;
23: end if
24: if Safe_Son_S > 0 & & Dead_Son_S > 0 then
25:   for every dead edge, let α_x be the associated action,
      W^LC_Cx = W^LC_Cx ⋃⟨S, α⟩;
26: end if
```

Before performing a critical action (that might lead to a state in FS_A), a local wrapper has to verify (by performing additional communication) if the global state represents a last chance state with respect to that action. Since at runtime we do not store A, this verification is made by enquiring the other local wrappers about the states of the corresponding components, hence acting accordingly. If a component is not in the inquired state, its associated local wrapper immediately replies ensuring that the component will not reach such a state before the inquired component sends an unblocking message referred to such a state. In this sense it is *self-blocked* with respect to the enquired state. If the component is already in the enquired last chance state or it is in the process of reaching it (*i.e.*, it is performing its *Ask* procedure with respect to an action that leads the component to the enquired state), its local wrapper defers the answer and hence, it attempts to block the enquiring local wrapper. The only case in which an enquiring local wrapper has to ask for permission from all the other ones is when the global state is a last chance one (since nobody immediately replies). Once the inquiring local wrapper receives an answer, it allows its corresponding component to proceed with its standard communication by forwarding the critical action. After that, it sends a message to unblock all the other local wrappers previously inquired with respect to the enquired state (additional communication). The unblock message is needed because once a local wrapper allows an enquiring one to perform a critical action, it ensures also that it will not reach the last chance state before receiving an unblock message with respect to such a state (see code lines 5 and 9 of Procedure *Ack* below). In practice it is self-blocked just with respect to the enquired state.

P_{LTS} is visited and distributed among the local wrappers (see Procedure *PVisit* reported below). Such a distribution is done by means of another table $W^{UA}_{C_i}$ for each local wrapper W_{C_i} (called *Updating and Allowed actions table* of W_{C_i}) of tuples $\langle$*state of* P_{LTS}, *allowed action of* C_i, *state of* P_{LTS}, *set of components, set of components*$\rangle$.

Let $1, \ldots, n$ be the set of unique identifiers for the n components to be assembled. Let $C_1 = (S_1, T_1, D_1, s^1_0), \ldots, C_n = (S_n, T_n, D_n, s^n_0)$ be the set of associated (relabeled) LTSs (see Appendix A.5) where for each $i = 1, \ldots, n$, S_i is the set of states, T_i is the set of transition labels, D_i is the set of transitions, and s^i_0 is the initial state. Let $P_{\mathrm{LTS}} = (S_P, T_P, D_P, p_0)$, then $W^{UA}_{C_i}$ is formally defined as follows:

$$W^{UA}_{C_i} = \{\langle p, \alpha, p', AC_p, AC_{p'}\rangle | \exists \alpha\_i \in T_i : \alpha\_i \cong_U l \wedge (p, l, p') \in D_P\}$$

where

$$AC_p = \{j | \exists \alpha\_j \in T_j : \alpha\_j \cong_U l' \wedge (p, l', p'') \in D_P \text{ for some } p'' \in S_P \text{ and some } l' \in T_P\},$$
$$AC_{p'} = \{k | \exists \alpha\_k \in T_k : \alpha\_k \cong_U l'' \wedge (p', l'', p''') \in D_P \text{ for some } p''' \in S_P \text{ and some } l'' \in T_P\}$$

and according to the semantics of action labels in the desired behavior LTS (introduced in Section 3.2 and defined in Appendix A.7), $\cong_U$ is a matching operator between a component action label α and an action label l of the desired behavior LTS. The first three elements of each tuple represent a transition of P_{LTS}. The fourth (fifth) is the set of (identifiers of) *Active Components* (AC_p and $AC_{p'}$, respectively), *i.e.*, the ones that can perform some actions "matching" with a transition outgoing from the state of P_{LTS} specified by the first (third) element of each tuple (*i.e.*, p and p' in the definition).

Fig. 8 shows a possible execution of *PVisit* applied to the desired behavior LTS shown in Fig. 4. Informally, considering a depth-first visit as shown in figure, the allowed action tables are updated as follows. The transition 1 specifies that component $C2$ "moves" the current state of P from $S0$ to $S1$ when performing action $!C1.a$. Moreover, by means of a sub-procedure, we store in the same entry of the allowed action table of $w2$ (*i.e.*, the wrapper that supervises $C2$) also the set of components that can perform at least one action in the new state of P. Actually, since all the components are allowed to perform at least one action in both states $S0$ and $S1$, the fourth and the fifth items of each table entry are set to '*'. The allowed action table of $w2$ and of all the other wrappers are updated by proceeding with the depth-first visit on the transitions 2, 3, and 4, in an analogous way. At the end of the visit, the output of *PVisit* is given by the set of tables shown on the left-hand side of Fig. 8. Note that the table corresponding to $w1$ allows component $C1$ to perform any action without moving the current state of P.

By means of *PVisit* each local wrapper knows its allowed actions that can change the state of P_{LTS}. Moreover, a local wrapper knows also which are the active components that can move and which must be blocked according to the current state of P_{LTS}. Let us assume that a component C_i is going to perform an action contained in its table $W_{C_i}^{UA}$. If it can proceed according to the current state of P_{LTS}, then all the other active components are blocked by sending a blocking message to the corresponding local wrappers. Once C_i has performed the action, all the components that can move in the new state of P_{LTS} are unblocked and they are made aware of the current state of P_{LTS} (see code line 28 of Procedure *Ask* below). Note that if an action of an active component does not change the state of P_{LTS}, it can be performed without exchanging messages among the system components, hence maintaining pure parallelism (this is realized by Procedure *Ask*, code line 31).

Considering the table of updating allowed actions, let *Lookahead*(*state of* P_{LTS}: p) be a procedure that given a state p of P_{LTS}, returns the set of components that are allowed to perform at least one action in the state p. *PVisit* takes in P_{LTS} that predicates on a (sub-)set of the n components to be assembled. The procedure makes use of the following variables: *Active_Components* (AC_p) is the set of components that are allowed to make a move in the current state p; *Next_Components* (NC_p) is the set of components that must be allowed to move once the current state of P_{LTS} has changed to p' (*i.e.*, $NC_p \equiv AC_{p'}$); $W_{C_i}^{UA}$ is the table of updating and allowed actions of the component C_i.

Procedure (*PVisit*(*state of* P_{LTS}: p;)).

```
1:  for each i := 1 to n do
2:    W_Ci^UA := ∅;
3:  end for
4:  Active_Components := Lookahead(p);
5:  Next_Components := ∅;
6:  mark p as Visited;
7:  for each son p' of p do
8:    if the edge (p, p') is not visited then
9:      mark the edge (p, p') as Visited;
10:     Next_Components :=Lookahead(p');
11:     for each C_i ∈ Active_Components allowed to perform
        an action α by the label of the edge (p, p') do
12:       W_Ci^UA := W_Ci^UA ⋃⟨p, α, p', Active_Components,
          Next_Components⟩;
13:       if p' is not visited then
14:         PVisit (p');
15:     end if
16:   end for
17: end if
18: end for
```

Once this procedure has been performed, each local wrapper knows the states of P_{LTS} from which it can allow the corresponding component to perform a specific action. Moreover, once the component performs such an action, it knows also which are the components that must be blocked and which ones must be unblocked in order to respect the behavior specified by P_{LTS}.

To summarize, the setup of *Last Chance* and *Updating and Allowed action* tables is realized by means of the procedures *AVisit* and *PVisit*. They are depth-first visits of A and P_{LTS}, respectively. These procedures are performed at construction-time and, after their execution, A and P_{LTS} can be discarded.

We now describe how local wrappers use, at run-time, the built tables to correctly interact with each other ensuring (i) deadlock-freeness and (ii) the behavior specified by P_{LTS} (*e.g.*, the LTS shown in Fig. 4 in our explanatory example). Before describing it in details, by referring to Fig. 9, let us give some intuitions of the method we use. Note that, while interacting, components may need an ordering among the sent and received messages. This can be realized in several ways and in our implementations we have made use of the standard time-stamp method (see Appendix B). However, for the sake of readability, we prefer not to make explicit this detail in the rest of the paper. The method we use is based on two procedures called *Ask* and *Ack*, respectively, whose pseudo-code is shown below.

Referring again to Figs. 3 and 4, in Fig. 9 an high-level description of the messages that can be exchanged between the wrappers and the components is shown. These messages are exchanged in order to ensure deadlock-freeness and the desired behavior

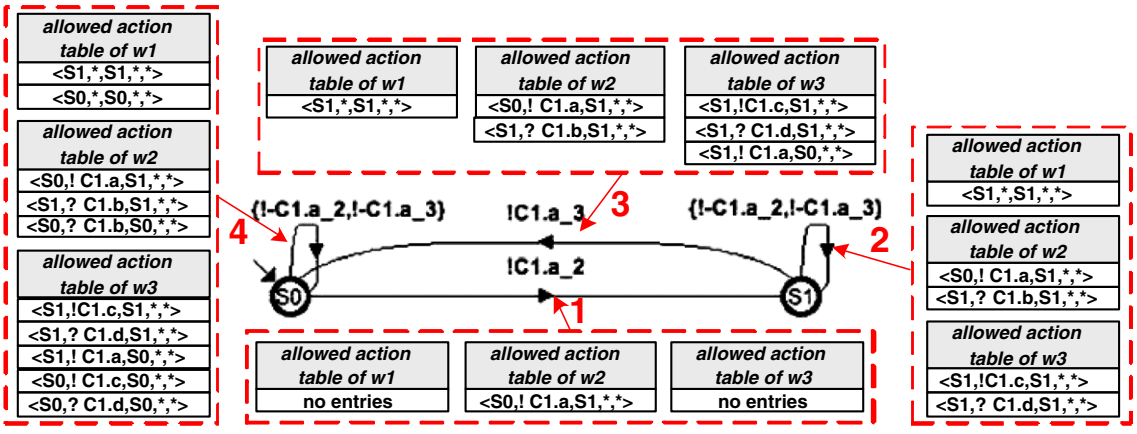


Fig. 8. Procedure *PVisit*: a possible execution.

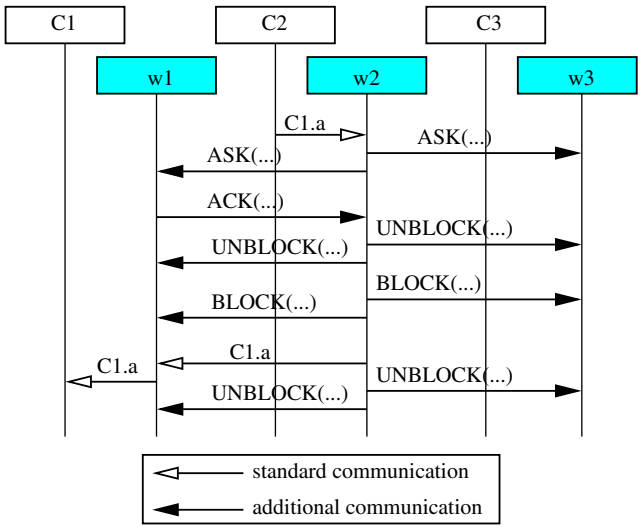


Fig. 9. Procedures *Ask* and *Ack*: a possible execution.

specified by P (see Fig. 4). For the sake of simplicity, the scenario that we show concerns only the sending of the message $!C1.a$ by $C2$ when the system's execution is in a certain global state. When $C2$ wants to send the request $C1.a$, first of all it needs to know whether the global state is $S4$ (*i.e.*, $\langle S2, S0, S1\rangle$), since it knows from its stored table of last chance states that $C1.a$ is a critical action with respect to $S4$. This is realized by means of procedure *Ask* that enquires the wrappers of $C1$ and $C3$ (*i.e.*, the sending of the *ACK* messages towards $w1$ and $w3$ shown in Fig. 9) in order to retrieve the global state. If at least one wrapper, among $w1$ and $w3$, replies that $C1$ or $C3$, respectively, are not in $S4$ of the glue adaptor, then $w2$ will receive an *ACK* message that allows component $C2$ to send the desired request. The *UNBLOCK* messages, after the *ACK* message, say to all the local wrappers that were blocked with respect to the global state $S4$, to proceed. Moreover, before sending the request, $C2$ needs also to know if it is allowed to proceed according to the desired behavior P. This is realized by maintaining at each time a subset of active components corresponding to the states of P in which such components are allowed to perform actions. In our example, $C2$ would not be active if the global state matches with $S1$ in P. In the case the global state matches $S0$ of P, $w2$ has to block all the other active components since the state of P is going to change. This is realized by means of *BLOCK* messages. Once $C2$ has performed its action, $w2$ must make active all the components that can perform some action in the new state by means of *UNBLOCK* messages. The information concerning active components is stored in the tables of allowed actions spread among wrappers, and the exchanged messages are still managed by procedures *Ask* and *Ack*.

More precisely, let C_x be an active component that is going to perform an action α (*i.e.*, in C_x there is a state transition labeled with α, and α is allowed with respect to P_{LTS}, *i.e.*, α appears in $W^{UA}_{C_x}$). The associated local wrapper W_{C_x} checks if α is (i) a critical action (*i.e.*, α appears in $W^{LC}_{C_x}$) or (ii) if α changes the global state with respect to P_{LTS}. If it is neither (i) nor (ii), then W_{C_x} forwards α to the right component. In the case of (i), W_{C_x} enters procedure *Ask* described below in order to ask for the permission to forward α. This is done by checking if for any pair $\langle S, \alpha\rangle \in W^{LC}_{C_x}$ there is at least one local wrapper W_{C_y} whose corresponding component C_y is not in S. In the case of (ii), W_{C_x} enters the procedure *Ask* in order to try to block all the active components and after having performed α, it unblocks the components that can be activated with respect to the new state reached over P_{LTS}.

Procedure (*Ask(action:* α;)).

1: Let C_x be the current component that would perform action α and let S_{C_x} be its current state and p be the current state of P_{LTS};
Let $\langle t_i\rangle^{UA}_x$ be the ith tuple contained in the table $W^{UA}_{C_x}$ and $\langle t_i\rangle^{UA}_x[j]$ be its jth element;
2: *flag_forbidden* := 0;
3: **if** $\exists i \mid \langle t_i\rangle^{UA}_x[1] == p$ && $\langle t_i\rangle^{UA}_x[2] == \alpha$ **then**
4: **if** α appears in some pair of $W^{LC}_{C_x}$ **then**
5: **for** every entry $\langle S, \alpha\rangle \in W^{LC}_{C_x}$ **do**
6: $i := 1$;
7: **while** no "ACK, α" received && $i \leqslant n$ **do**
8: Let $S \equiv \langle S_{C_1}, \ldots, S_{C_n}\rangle$; W_{C_x} asks to local wrapper W_{C_i} if it is in or approaching[6] the state S_{C_i};
9: $i++$;
10: **end while**
11: **if** $i > n$ **then**
12: WAIT for an "ACK, α" message of one enquired component C_y;
13: **end if**
14: **if** $i > n$ **then**
15: $i := n$;
16: **end if**
17: **for** $j := 1$ **to** i **do**
18: send "UNBLOCK, α" to W_{C_i};
19: **end for**
20: **end for**
21: **end if**
22: **if** $\langle t_i\rangle^{UA}_x[1]! = \langle t_i\rangle^{UA}_x[3]$ **then**
23: **for** each component $C_j \in \langle t_i\rangle^{UA}_x[4]$ **do**
24: send "BLOCK" to W_{C_j};
25: **end for**
26: perform action α;
27: **for** each component $C_j \in \langle t_i\rangle^{UA}_x[5]$ **do**
28: send "UNBLOCK, $\langle t_i\rangle^{UA}_x[3]$" to W_{C_j};
29: **end for**
30: **else**
31: perform action α;
32: **end if**
33: **end if**

Note that, by code line 12, the current local wrapper is self-blocked until some other local wrapper gives it the permission to proceed, *i.e.*, an "ACK". The "*UNBLOCK*" messages of code line 18 say to all the local wrappers that were blocked with respect to the enquired forbidden states, to proceed. The "*UNBLOCK*" messages of code line 28 are instead to unblock components due to the change of P_{LTS} state occurred after having performed action α. On the other hand, when a local wrapper receives a request for a permission, after having given such a permission, it is implicitly self-blocked with respect to the set of states it was enquired for. The following procedure describes the "*ACK*" messages exchanging method.

Procedure (*Ack(last chance state: S*; action: α;)).

1: Let W_{C_y} be the local wrapper (performing this Ack) that was enquired with respect to the state S and the action α that C_x would perform.
2: **if** C_y is not in S && W_{C_y} didn't ask for permission to get in S **then**
3: send "ACK, α" to W_{C_x} that allows C_x to perform the action

[6] C_i is performing its *Ask* procedure with respect to an action that leads C_i to S_{C_i}.

```
α;
4:   if C_y would reach S with the next hop then
5:     WAIT for "UNBLOCK, α" from W_Cx;
6:   end if
7: else
8:   once C_y is not in S send "ACK, α" to W_Cx that allows C_x to
     perform the action α;
9:   if no "UNBLOCK, α" from W_Cx has been received then
10:    WAIT for it;
11:  end if
12: end if
```

The "*WAIT*" instructions of code lines 5 and 10 block the current local wrapper in order not to allow the corresponding component to enter a forbidden state. Note that, while the "*UNBLOCK*" message has a one-to-one correspondence, that is, for each message there is a receiver waiting for it, the "*ACK*" message can be sometimes useless. In fact, a local wrapper needs just one "*ACK*" message in order to allow the corresponding component to proceed with the enquired critical action. All the other possible "*ACK*" messages are simply ignored.

5.3. Correctness

We now provide the correctness of our method. Given A and P_{LTS}, we show that our method synthesizes local wrappers that (i) allow the composed system to be free from deadlocks and (ii) allow P_{LTS} to be exhibited.

We prove (i) by focussing on the last chance nodes. Since the synthesis of A is correct as proved in Tivoli and Autili (2006), we can assume that the last chance nodes are correctly discovered by means of the procedure *AVisit* that performs a standard depth-first visit. Thus, our proof can be reduced to show that the local wrappers disallow the system to reach a forbidden trace. Note that, by construction, such a trace can be undertaken only through a last chance node by performing an action that labels one of its outgoing dead edges. Let us assume by contradiction that the component z can perform the critical action c from the last chance state S, and that S has an outgoing dead edge labeled by c_z (see for instance Fig. 6). Since, as already noted, the last chance nodes are correctly discovered, when procedure *AVisit* is visiting S, it correctly stores in W_z^{LC} the tuple $\langle S, c\rangle$. At runtime, whenever the component z would perform action c, W_z checks if c is a critical action by means of code line 4 of its *Ask* procedure. It then starts to ask for permission (at least one "ACK" message is required) from all other components by means of the "while" statement (code line 7 of the same procedure). Each enquired local wrapper W_{C_i}, by the *Ack* procedure, checks if the current state of the corresponding component C_i is in S. If it is, it does not reply to z until it does not change state (code line 8 of the *Ack* procedure). In doing so, until the system state remains S, no local wrapper will reply to W_z. Since W_z is blocked on code line *12* of the *Ask* procedure until no "ACK" message is received, a contradiction follows by observing that action c can be performed by z only at code line 26 of the same procedure.

To prove (ii), let us assume by contradiction that the component x performs the action a when it is not allowed by P_{LTS}, that is, the current state S_P of P_{LTS} has no outgoing edge labeled by a_x. First of all, in order for a component to be active, either its local wrapper has received an "UNBLOCK" message from some other local wrapper (by means of code line 28 of the *Ask* procedure) or the system has just started and W_x^{UA} has some entry with S_0 (the initial state of A) as first element. In both cases, each time a component is active, its local wrapper knows exactly which is the current P_{LTS} state. By construction, x can perform action a if there exists an entry in W_x^{UA} whose first element matches with the current state of P_{LTS} and whose second element matches with a (see code line *3* of the *Ask* procedure). The contradiction follows by observing that such an entry was obtained by visiting P_{LTS} hence, by construction, there must exist an outgoing edge whose label matches with a_x from the node labeled by S_P.

5.4. Overhead

Our approach adds some overhead due to the messages that the local wrapper have to send each other in order to synchronize themselves when required (*i.e.*, additional communication). This happens when a component has to perform an action that either might lead to a forbidden state according to deadlocks or is not allowed according to P_{LTS}. In the worst case, this means sending, for each action, a message to all the other components in order to either ask for permission to perform the current action or communicate that the state of P_{LTS} is changed. Of course, in practical cases, where usually many parallel computations are allowed, the overhead is much smaller and the additional messages do not decrease system performance.

In the following section we introduce the Synthesis tool that implements the presented approach.

6. The Synthesis tool

By referring to Fig. 10, the method implemented by the current version of Synthesis assumes the following data as inputs:

(a) the interface specification of the components to be assembled. It is given as a set of IDL files, one for each component implementing a *server* logic (obviously, for components that implement only a client code there is no IDL). According to "*design by contract*" approaches (Szyperski, 2004), we can assume that each of these IDL files is augmented by the component developer through a commented header. Such an header contains a relative path[7] which externally refers to an XML file encoding the specification of the interaction protocol (see input (c)). For a client, such an XML file is directly provided by the client developer. In our context, a component always respects its interaction protocol specification since it is provided by the developer of the same component, who is aware of the information needed to specify the component interaction protocol.
(b) The LTS-based specification of the desired behavior.
(c) For each component (either client or server), the specification of its interaction protocol with the expected environment. It is an XML file that encodes an *high-level Message Sequence Chart* (hMSC). An hMSC specifies the possible scenarios for message exchanging between the component and its expected environment, and the possible continuations among the scenarios. For a server component, this XML file is referred within its augmented IDL.

These three inputs are then processed in two main steps:

(1) considering inputs (a) and (c), Synthesis automatically derives the component LTSs. As already said, this is done by implementing a revisited version of the algorithm described in Uchitel et al. (2004). However, Synthesis can also take these LTSs directly as input by providing the user with LTS drawing tools. From the component LTSs, Synthesis automatically derives the LTS A of the centralized glue adaptor (*i.e.*, output (d)).
(2) After A has been generated, Synthesis explores both it and the

[7] Note that, this is only an implementation choice and it can easily be changed.

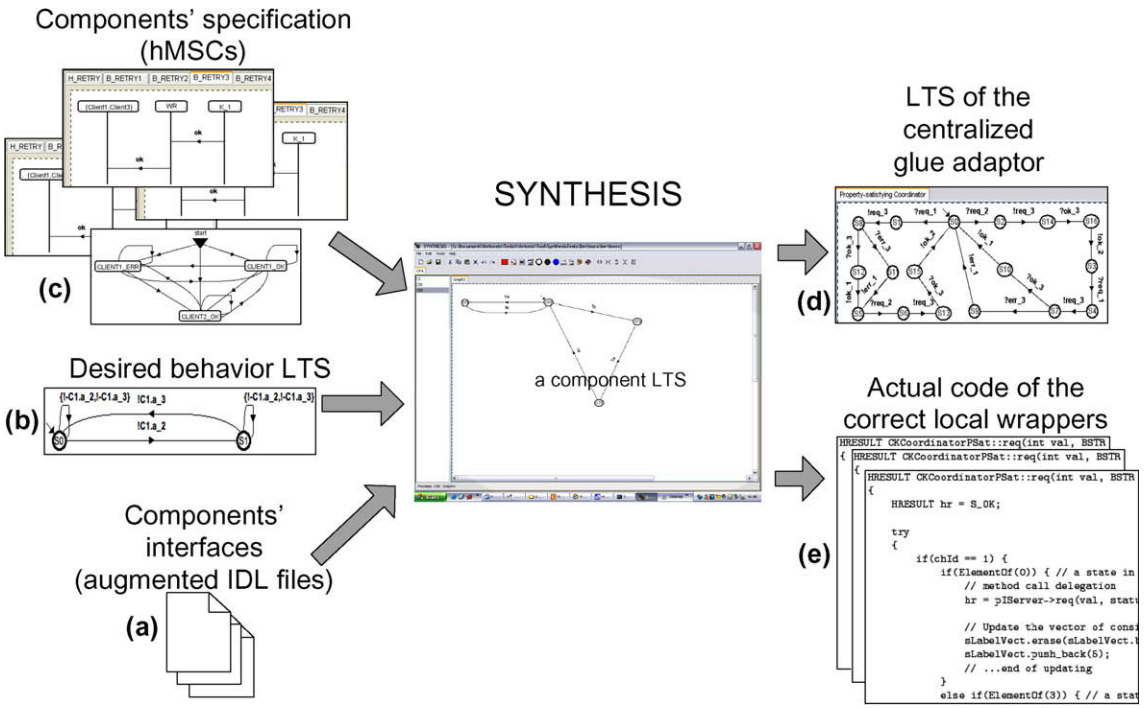


Fig. 10. The SYNTHESIS tool.

LTS of the desired behavior to synthesize the actual code of the correct distributed adaptor (*i.e.*, output (e)). It is implemented as a set of EJB component wrappers (Autili et al., 2007). Each wrapper is developed by using AspectJ that easily supports the wrapper tasks of intercepting the component messages and correctly coordinating them. Note that AspectJ is only one possible implementation choice.

7. Architecture of the SYNTHESIS tool

In Fig. 10 we have shown the input and output data of the SYNTHESIS tool. Now, by means of capital letters, we obtain a direct mapping between Figs. 11 and 10 that allows us to correlate each module with the I/O data it performs.

In the reminder of this section, we briefly describe each SYNTHESIS module.

Module (A), component interface parser: this module contains a superclass (*i.e.*, "IDLParser" entity in Fig. 11) 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, and Java IDL for EJB). In the current version of *SYNTHESIS*, we specialized that class to implement two parsers: one for Microsoft IDL (MIDL) files and the other for Java IDL files.

Module (B), LTS specification of the desired behavior: this module is used to specify the desired behavior for the system to be built in terms of LTSs. The current implementation of this module encodes each LTS as a binary object by exploiting *Java* serialization. A corresponding version based on XML is still work in progress. Each LTS describes component interactions that must be cooperatively guaranteed by the component wrappers to be generated.

Module (C), hMSC specification of the components: this module is used to specify/display/process the interaction behavior of each component with its expected environment in terms of an hMSC specification. In doing that, this module exploits an *ad hoc* library that we have developed to allow creation, validation and manipulation of hMSCs encoded 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 the relation "requires" between the two).

Module (D), builder of the centralized adaptor model: this module is responsible for deriving the model of the centralized glue code. In particular, it is specialized by the "LTS Unificator", "DlockEraser" and "Desired Behavior Analyzer" entities shown in Fig. 11D. These entities respectively implement: (i) the LTS unification algorithm that is for building the LTS of the centralized adaptor, it uses the "LTS Builder" that exploits the module C. Taking as input the hMSC specification of the components, it builds and outputs the component LTSs. According to the hMSCs data format, an LTS is coded in *XML*. (ii) The last chance states identification algorithm. (iii) The allowed actions (with respect to the specified desired behavior) identification algorithm.

Module (E), generator of the component wrappers actual code: this module implements a generator of the component wrappers actual code. It is structured analogously to module A and, hence, it refers to one or more specific development platforms. Currently, it supports the generation of the code implementing either centralized COM/DCOM or distributed EJB adaptors. Note that a component adaptor does not care about checking the value of actual parameters of a component method. In fact, it is merely a "delegator" of method calls, whose forwarding logic respects both deadlock-freeness and the specified desired behavior. This explains why, in drawing the hMSC specification of the components, *SYNTHESIS* does not require a user to specify the actual parameters of a method call.

8. Case study

In this section, we describe our approach at work by means of an industrial case study. The case study concerns the semi-automatic

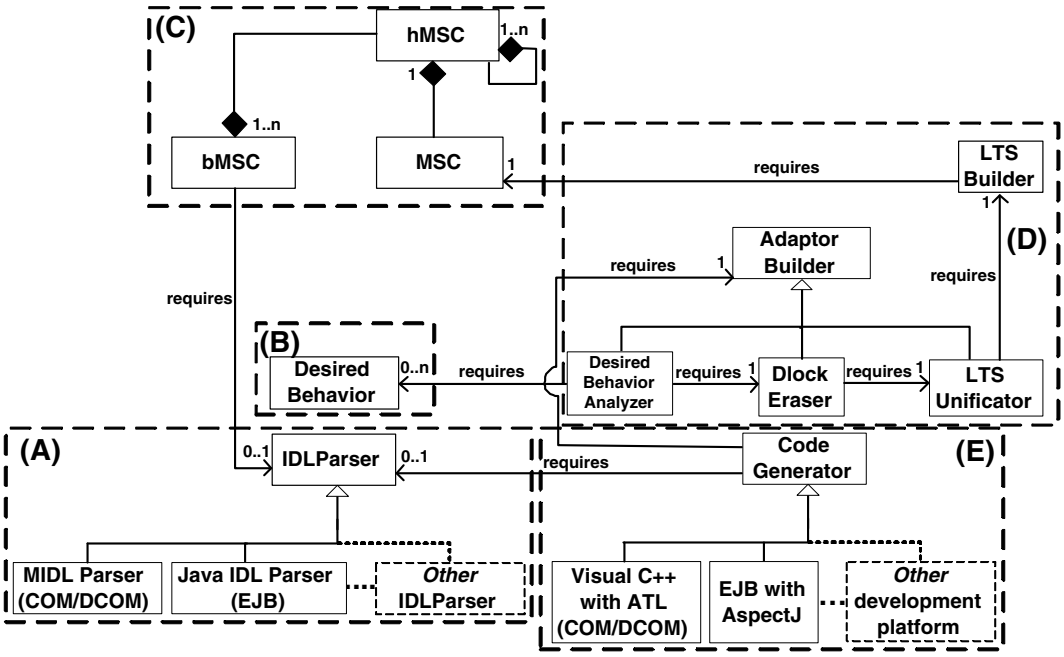


Fig. 11. The SYNTHESIS architecture.

assembly of part of a large distributed system built in the context of the CUSPIS project (CUSPIS).

8.1. The CUSPIS project

In European society, increasing importance is given to the issue of safeguarding, enjoying and supporting *Cultural Heritage*. The European Commission gives high importance to that issue, promoting actions for protection and safeguarding, improving understanding and dissemination of culture and history of the European citizen, making Cultural Heritage increasingly available and accessible.

The CUSPIS project (CUSPIS) combines the Cultural Assets (CAs)[8] infrastructure with the GALILEO and EGNOS infrastructures in order to support the Cultural Heritage safeguarding and protection. To this extent the CUSPIS project focuses on the specification, implementation and deployment of secure information mobility platforms that offer two basic services: Cultural Assets Management (CAM) and Cultural Assets Fruition (CAF).

CAF concerns the dissemination of CAs information everywhere, *e.g.*, people can go around in a museum and receive CAs information on their mobile devices. CAM concerns the secure transport of CAs from a renter (the organization requiring the CAs) to the owner (the organization that holds the CAs).

The CAM process requires three sub-processes: (i) the certificate request, (ii) the certificate generation, and (iii) the monitoring of the CA transport.

In this work, we focus on the CAM service and we show how our approach has been used to automatically implement the certificate generation service out of a set of already implemented black-box components. In Fig. 12, we show the two basic activities that the certificate generation service has to support.

In the first activity (see Fig. 12A) the renter and the owner produce a request certificate that expresses their approval to move a CA from the owner location to that of the renter (*i.e.*, the CA journey). In the following we describe in details all the request certificate fields. The CA_ID field is a signed string that contains the unique identifier of the CA to be moved. *Motivation* is a string that describes the motivation leading to the Cultural Asset journey. The field *renter* (resp., *owner*) contains the X500 name (ITU-T) of the *renter* (resp., *owner*) entity. The field *RenterSignature* (*OwnerSignature*) contains the signature of the fields (*owner*, *Motivation*,*owner*,*Renter*) that is generated with the renter (resp., owner) private key.[9]

In the second phase (see Fig. 12B) the CA owner and a ministry authorized person produce a validation certificate that is used to certify the ministry permission to the CA journey. The *request certificate* field contains the request certificate produced during the first activity. The *owner* and *ministry* fields contain the X500 name of the owner and ministry authorized person, respectively. The *Ministry Signature* (resp., *Owner Signature*) contains the signature of the fields (*request certificate*, *Owner*, *Ministry*) that is generated with the owner (resp., Ministry) private key. In the following section we describe the set of existing components that we have taken into account to automatically and correctly assemble the part of the CUSPIS project that realizes the certificate generation service.

8.2. The existing components and SYNTHESIS at work

In Fig. 13, we show the CUSPIS sub-system that actualizes the certification service.

The component adapter *Ao*, the X500 name server *So*, and the security component *To* reside on the owner host. The component adapter *Am*, the X500 name server *Sm*, and the security component *Tm* reside on the ministry host. The renter certificate client *Cr*, the owner certificate client *Co*, and the ministry certificate client *Cm* can access the owner and ministry hosts through the public network. The clients *Cr* and *Co* interact in order to produce the request certificate. The clients *Co* and *Cm* interact in order to produce the validation certificate. We remark that the request certificate must be always produced before the validation one.

[8] In the context of the CUSPIS project, examples of cultural assets are sculpture, pictures and so on.

[9] SHA with RSA algorithm is used to produce the signature.

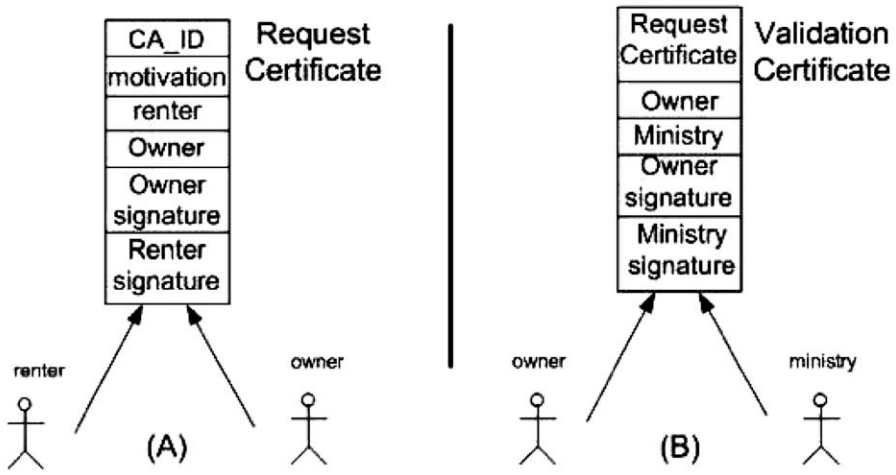


Fig. 12. The certificate process.

In our case study, we have to face two main problems of adaptation. The first problem is a consequence of the use of existing components in a different context from the one they have been originally thought. In particular, these components were developed in a previous project and we want to reuse them in the context of the CUSPIS project because they already realize the required functionalities. The second problem is due to the use of the adaptor and of the X500 server in different hosts, *i.e.*, the ministry and the owner host. These different uses require different adaptations of the same components.

In Figs. 14–17, we show the LTSs of our existing components as they are displayed by the SYNTHESIS tool. The LTSs of the two adapter components *Ao*, and *Am* are shown in Fig. 14. Fig. 15 shows the LTSs of the server components *So*, *Sm*, *To*, and *Tm*; finally the LTSs of the two client components *Cr*, and *Cm*, and the LTS of the client component *Co* are shown in Figs. 16 and 17, respectively. We recall that, a ?*m* (resp., !*m*) denotes a received (resp., sent) message labeled with *m*. The state with an incoming arrow denotes the initial state.

The LTSs of the adapter and server components can be easily understood by looking at Figs. 14 and 15, respectively. The LTSs of the adapters and servers do not need further explanation because the semantics of their transitions is explained, in the following, while discussing the LTSs of the clients.

In the initial state the renter (resp., ministry) client *Cr* (resp., *Cm*) can send the connection request to the server *So* (resp., *Sm*) and, from the state *S1*, it can receive the successful connection notification. After a correct connection, the client (either the renter or the ministry) can send the request *setAdaptor* (to either *Ao* or *Am*), followed by the request *setX500name* (to either *So* or *Sm* according to the previous call of *setAdaptor*). The *setAdaptor* request is used to set the motivation and the CA_ID of the request certificate. The *setX500name* request is used to set the X500 renter (or ministry) name in the validation certificate. The request *releaseX500* is sent from *Cr* (resp., *Cm*) to *So* (resp., *Sm*) in order to release the resource it has acquired. The request *releaseAdaptor* is used to release the *Ao* (resp., *Am*) resource. Note that the *releaseAdaptor* request involves the process of sending the renter (resp., the ministry) signature in order to sign the request certificate.

The owner client *Co* (see Fig. 17) performs almost the same behavior as the one of either *Cr* or *Cm*. The only difference is that

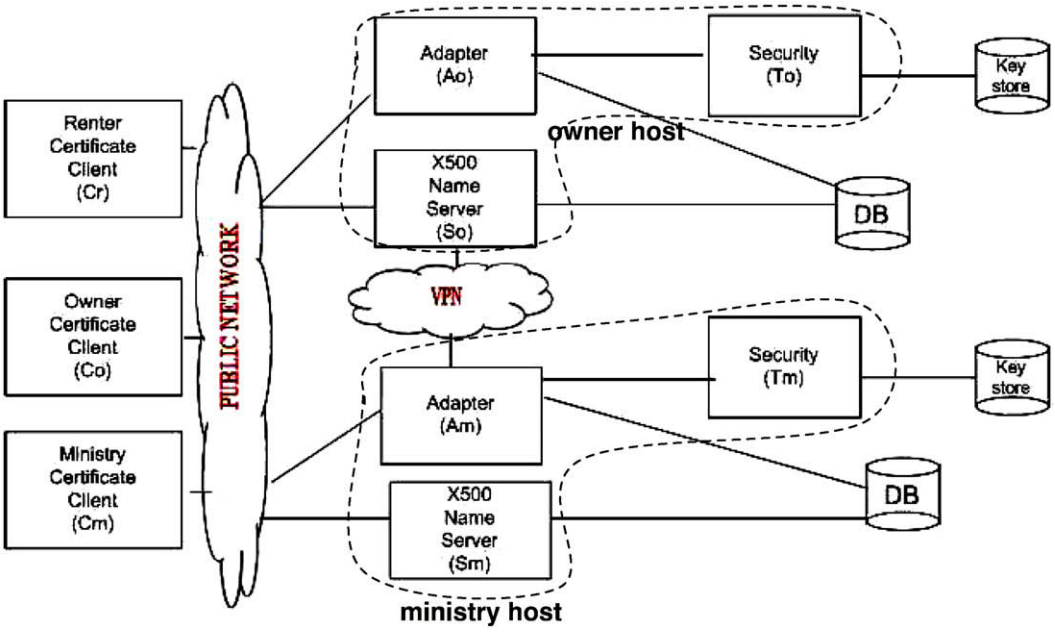


Fig. 13. The CUSPIS system: the certification sub-system.

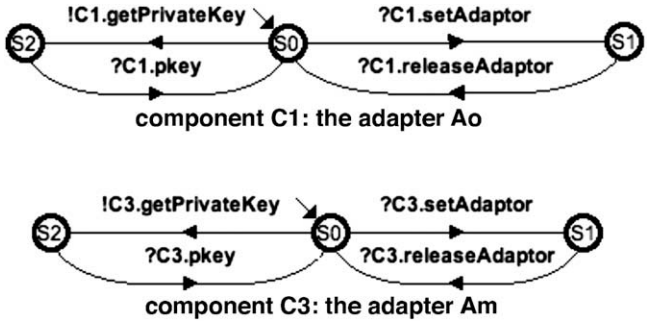


Fig. 14. LTSs of the adapter components.

Co calls a *setX500name* followed by a *setAdaptor*, whereas *Cr* and *Cm* call a *setAdaptor* followed by a *setX500Name*.

In Fig. 18, we show the LTS specified, by using SYNTHESIS, to model the system desired behavior that we wish to guarantee in order to correctly assemble the previously specified components. We denote it as P_{LTS}. By referring to 3.2, we recall that P_{LTS} is an high-level description of a desired behavior that we want to guarantee in the resulting DABA system that is being assembled.

Substantially, P_{LTS} allows the ministry client *Cm* (referred as *C5* within SYNTHESIS) and the owner client *Co* to interact with the ministry server (referred as *C4* within SYNTHESIS) only after both the renter client *Cr* (referred as *C6* within SYNTHESIS) and the owner client *Co* (referred as *C7* within SYNTHESIS) have released the adaptor

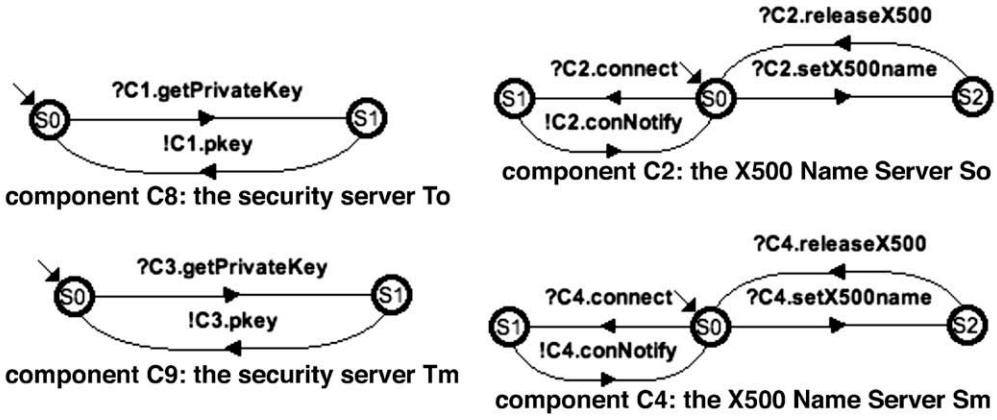
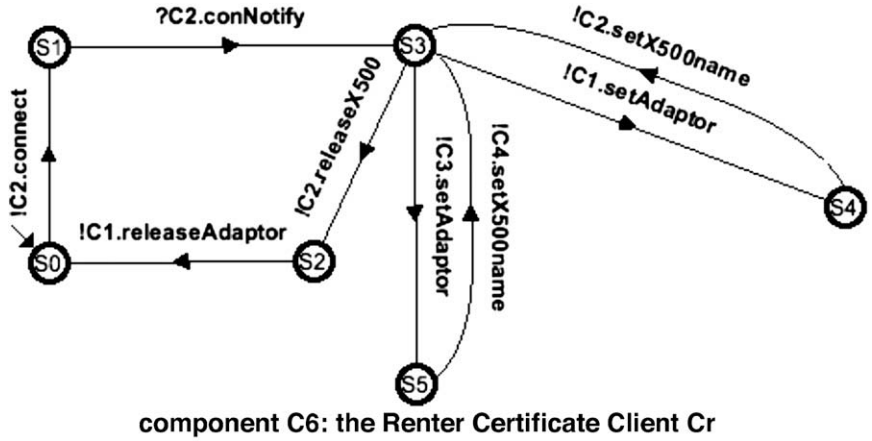


Fig. 15. LTSs of the server components.



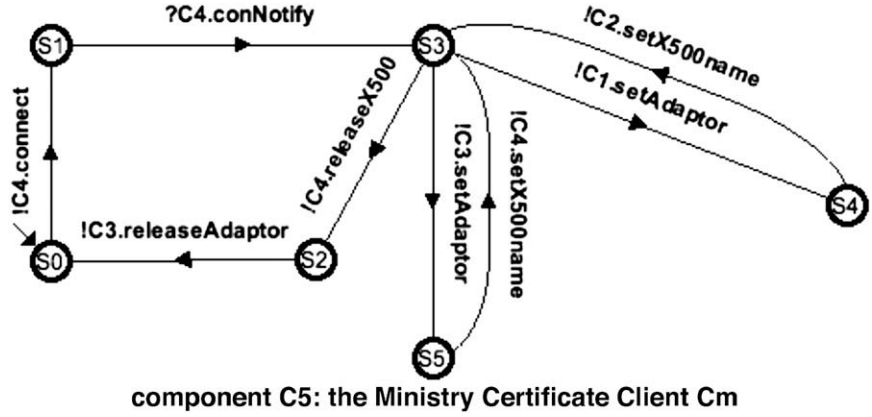


Fig. 16. LTSs of the client components *Cr* and *Cm*.

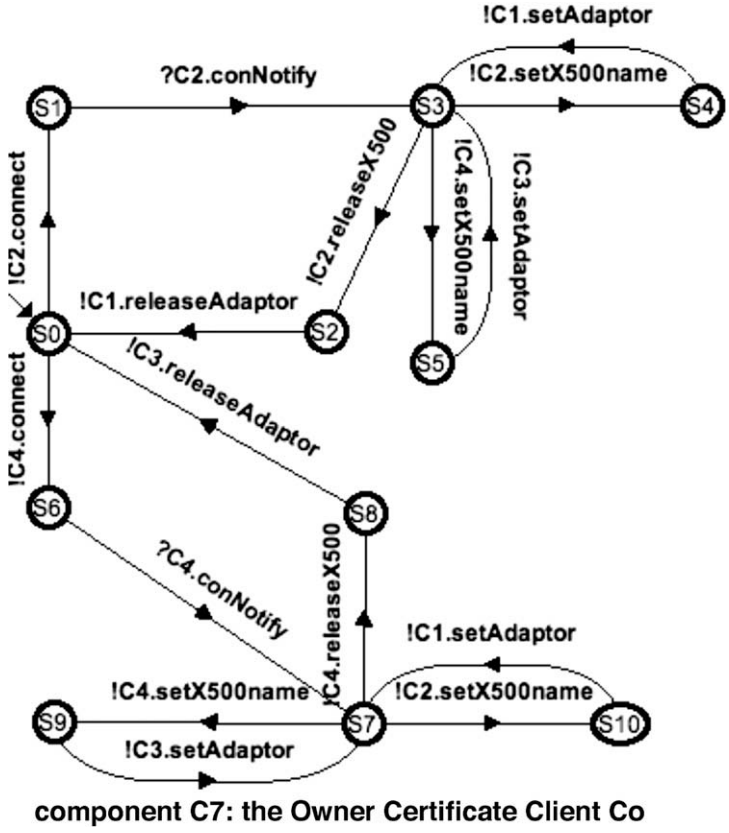


Fig. 17. LTS of the client component *Co*.

resource (state *S*3 shown in Fig. 18). In other words, it models the fact that the request certificate must be always written before the validation certificate.

By referring to the method discussed in Section 5.1, by taking into account the LTSs shown in Figs. 14–17, we automatically derive a model of the centralized glue adaptor *A*. This is done by using SYNTHESIS and performing the approach described in Inverardi and Tivoli (2003). Finally, by taking into account the LTS specification of the desired behavior that the composed system must exhibit (see Fig. 18), we mechanically distribute the correct behavior of *A* in a set of local wrappers each of them for each existing component.

The generation of the LTS modelling the behavior of *A* took 11.5 minutes, by running SYNTHESIS on a *1.83 GHz Intel Core Duo processor*. This LTS has 8031 states and 15332 transitions. Due to the size of the LTS of *A*, its graphical representation within the SYNTHESIS tool is obviously unreadable, hence we do not show it. Despite this, SYNTHESIS returns useful information about the possible deadlocks and its corresponding last chance nodes (hereafter, called also last chance states).

For instance, the LTS of *A* has two sink states whose *IDs* are S4842 and S5204. Beyond these two states, it has also eight, the so-called, *deadlocking* states that are either sink states or states always leading to deadlocking states. Their IDs are S4881, S4994, S5215, S5240, S4841, S4891, S5203, and S5220. Corresponding to these deadlocking states, there are eight last chance states whose IDs are S3553, S3657, S3892, S3931, S3533, S3557, S3883, and S3894. From the states S4881, S4994, S4841, and S4891 only the sink S4842 can be reached. From the states S5215, S524, S5203, and S5220 only the sink S5204 can be reached. By referring to the last chance states mentioned above, in Fig. 19 we show a fragment of the LTS of *A* that concerns the forbidden traces originating from those last chance states. In the figure the deadlocking states are drawn light-gray and the sink ones are drawn dark-gray.

We recall that within the LTS of *A* a state is a tuple of component LTS states. Furthermore, a transition label has the same syntax as a component LTS transition label except for a suffix (*e.g.*, "_5", "_7", "_4") that specifies from (resp., to) which component *A* has received (resp., sent) the message. By referring to Fig. 19, we show below the last chance states tables automatically generated by SYNTHESIS after the execution of the procedure *AVisit* (see Section 5.2). In order to prevent the detected deadlocks, all the components can proceed freely except for *Cm* (*i.e.*, *C5* in SYNTHESIS) and *Co* (*i.e.*, *C7* in SYNTHESIS). Thus, in the following we show F^{LC}_{C5} and F^{LC}_{C7} that are the last chance state tables for the deadlock-preventing local wrappers (*i.e.*, *w5* and *w7*) that supervises *Cm* and *Co*, respectively. These tables are shown by using the same textual format given as output by SYNTHESIS.

```
Last chance states table of w5:
<S3553=<S1,S2,S0,S2,S3,S3,S5,S0,S0>, ?C3.setAdaptor>
<S3657=<S0,S2,S1,S2,S3,S3,S4,S0,S0>, ?C1.setAdaptor>
<S3892=<S1,S2,S0,S2,S3,S3,S9,S0,S0>, ?C3.setAdaptor>
<S3931=<S0,S2,S1,S2,S3,S3,S10,S0,S0>,
?C1.setAdaptor>

Last chance states table of w7:
<S3533=<S1,S2,S1,S0,S3,S5,S3,S0,S0>,
?C4.setX500Name>
<S3557=<S1,S0,S1,S2,S3,S4,S3,S0,S0>,
?C2.setX500Name>
<S3883=<S1,S2,S1,S0,S5,S3,S3,S0,S0>,
?C4.setX500Name>
<S3894=<S1,S0,S1,S2,S4,S3,S3,S0,S0>,
?C2.setX500Name>
```

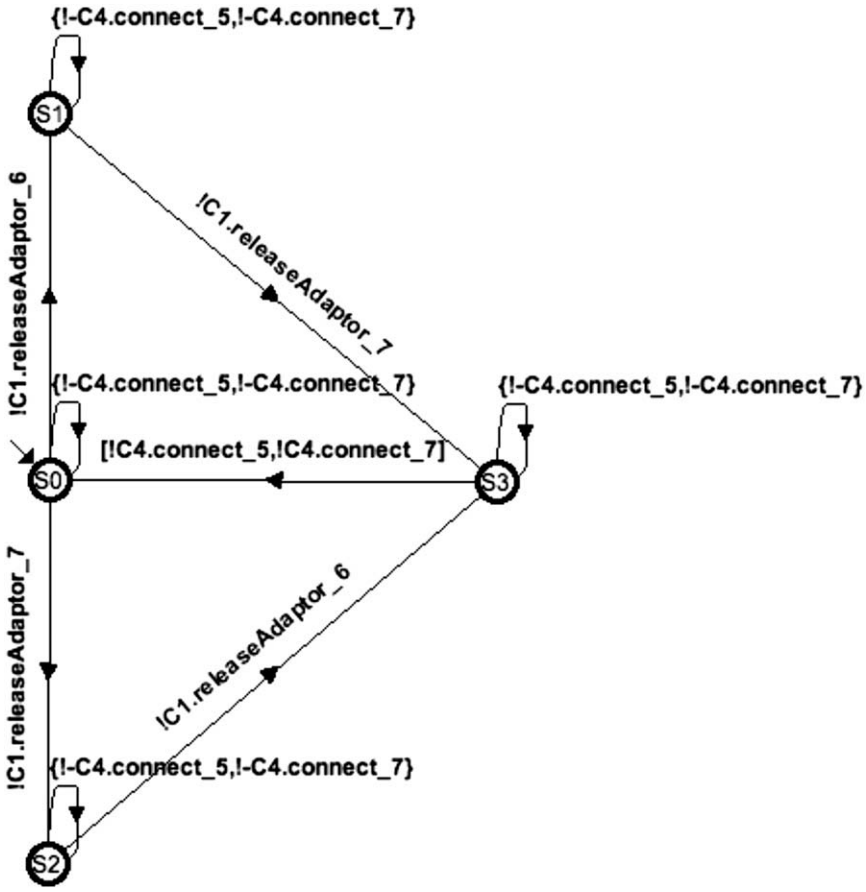


Fig. 18. P_{LTS}: the specification of the system desired behavior that must be enforced.

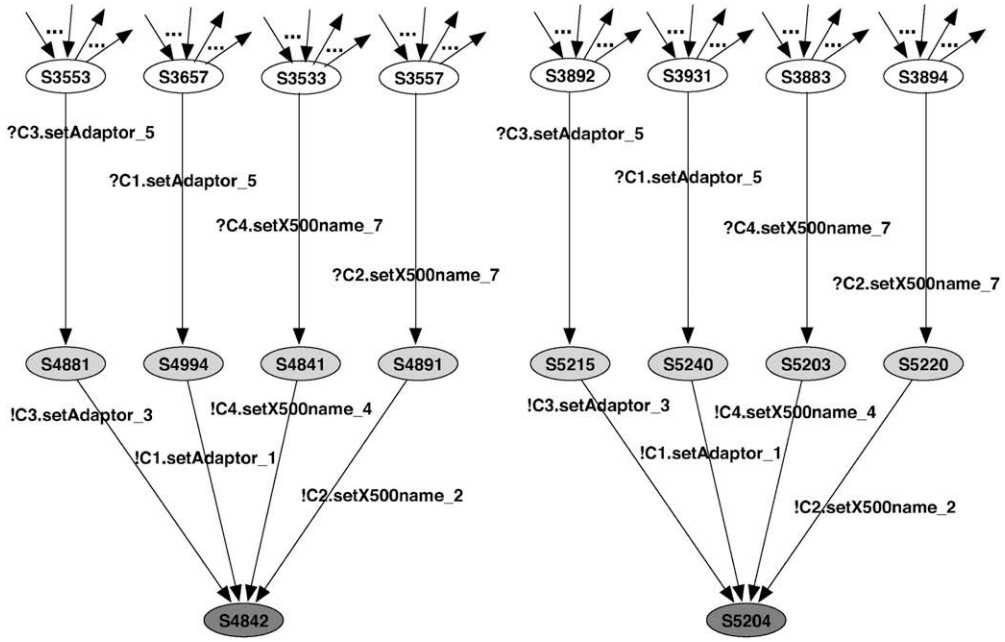


Fig. 19. Forbidden traces in the LTS of *A*.

By exploiting its last chance states table, each time the component *C5* (*i.e.*, *Cm*) performs the action *!C3.setAdaptor* (resp., *!C1.setAdaptor*), *i.e.*, it invokes the method setAdaptor to *Am* (resp., *Ao*), the corresponding local wrapper (*i.e.*, the local wrapper *w5*) has to check that the global state is neither S3553 nor S3892 (resp., neither S3657 nor S3931). By referring to Section 5.2, we recall that this is done by means of the procedure *Ask* (performed by *w5*) and the procedure *Ack* (performed by all the other local wrappers). The *Ask* procedure is used (by *w5*) in order to suitably query all the other local wrappers different from *w5* and, hence, to know

the current global state in terms of a tuple of component states. The *Ack* procedure is used (by all the other local wrappers) in order to either allow the querying component to perform an action or temporarily block it (since the action would violate some behavioral properties, *e.g.*, the deadlock-freeness). The local wrapper *w7* behaves in an analogous way according to its last chance state table.

After performing *AVisit* to derive the last chance state table for each local wrapper, SYNTHESIS performs *PVisit* (see Section 5.2) to derive the updating and allowed actions table for each local wrapper. This is done by taking into account the LTS specification P_{LTS} (see Fig. 18). We recall that these tables are needed to distribute P_{LTS} among the local wrappers.

Following we show the tables of updating and allowed actions used by each local wrapper as generated by the procedure *PVisit*. Denoting by "*" any possible value of a specified scope and by "* – *S*" any possible value of a specified scope except for the values in the set *S*, P_{LTS} is translated by procedure *PVisit* hence generating the following tables of updating and allowed actions. Below we show only the tables related to the local wrappers *w5*, *w6*, and *w7* because for all the other local wrappers the tables are of type <*, *, *, *, *> that means that any action is allowed in any state.

```
Table of updating and allowed actions of w5:
<S0, * - {! C4.connect},S0, *, *>
<S1, * - {! C4.connect},S1, *, *>
<S2, * - {! C4.connect},S2, *, *>
<S3, * - {! C4.connect},S3, *, *>
<S3, ! C4.connect,S0, *, *>

Table of updating and allowed actions of w6:
<S0, * - {! Cl.releaseAdaptor},S0, *, *>
<S0,Cl.releaseAdaptor, S1, *, *>
<S1, *, S1, *, *>
<S2, * - {! Cl.releaseAdaptor},S2, *, *>
<S2, ! Cl.releaseAdaptor, S3, *, *>
<S3, *, S3, *, *>

Table of updating and allowed actions of w7:
<S0, * - {! Cl.releaseAdaptor,C4.connect}, S0, *, *>
<S0, Cl.releaseAdaptor,S2, *, *>
<S1, * - {! Cl.releaseAdaptor, C4.connect},S1, *, *>
<S1, Cl.releaseAdaptor, S3, *, *>
<S2, * - {! C4.connect},S2, *, *>
<S3, * - {! C4.connect},S3, *, *>
<S3, C4.connect,S0, *, *>
```

Note that, when during the runtime, the state of P_{LTS} changes from *S0* to *S1* by means of the action *! C1.releaseAdaptor* performed by *C6* (*i.e.*, Cr), *w6* informs *w5* and *w7* of the new state of *P*_LTS by means of the "UNBLOCK" message of code line 28 of its *Ask* procedure (see Section 5.2). Consequently *w5* and *w7* know that in such a state their supervised components cannot perform *C4.connect* since the entries <S1, C4.connect, *, *, *> are not present in the table of updating and allowed actions of *w5* and of *w7*. The components supervised by *w5* and *w7* can perform *C4.connect* only when the state *S3* of P_{LTS} has been reached hence reflecting the specified desired behavior.

Once the tables of last chance states and of updating and allowed actions are filled, the interactions among local wrappers can start by means of the procedures *Ask* and *Ack* described in Section 5.2.

The actual code shown below is a fragment of the actual code that SYNTHESIS has derived for the local wrapper *w5* that supervises the component *C5* (*i.e.*, the ministry client *Cm*). As it is shown, SYNTHESIS uses *AspectJ* in order to write the code of a local wrapper. In other words, each local wrapper is implemented as an *aspect* by using AspectJ. Beyond coding both the tables of last chance states and of updating and allowed actions, and the procedures *Ask* and *Ack* (that are not shown in the code fragment) SYNTHESIS derives, for each possible component action, an AspectJ *pointcut*. For each pointcut, if it is the case that the pointcut is associated to an action that either can lead to a deadlock (*e.g.*, *setAdaptor* performed through instances of *C1* and *C3*) or might violate the specified desired behavior (*e.g.*, *connect* performed through an instance of *C4*), then both a *before* and an *after* advice is generated. Otherwise, only an after advice is generated. In the former case, in order to establish whether the component can perform a method call or not, before that call is performed (before advice), the local wrapper calls the procedure *AuthorizationRequest( )*. After the method call has been performed (after advice), the local wrapper updates its current state according to the LTS of the component it supervises (*i.e.*, the current state of a local wrapper is used to trace the current state of the supervised component). In the latter case, the current state is updated and the procedure *Ack* is called in order to allow a component, different from the supervised one, to possibly perform the method call associated to the pointcut defined for the after advice.

Procedure *AuthorizationRequest*( ) checks, through the last chance state table of the local wrapper, if the method call associated to the current pointcut can lead to forbidden states. If it is (as in the case of *c3Obj.setAdaptor*(...) and *c1Obj.setAdaptor*(...) associated to the pointcut *b*( ) and *c*( ), respectively), the procedure *Ask* is performed in order to ask the permission to the other component adaptors before performing that method call. If not (as in the case of *c4Obj.connect*(...) associated to the pointcut *a( )*), the local wrapper allows its supervised component to perform the method call without asking anything. In this case, *AuthorizationRequest*( ) also checks (through the updating and allowed actions table of the local wrapper) whether that method call makes P_{LTS} change its state or not. If it does (as in the case of *c4Obj.connect*(...)), *AuthorizationRequest*( ) sends a block message to all the adaptors whose supervised component cannot move (with respect to P_{LTS}) and an unblock message to all the ones whose supervised component was blocked, but now can move because of the new state of P_{LTS}. When, for example, *w5* asked, to some other local wrapper *wx*, for the permission of performing a call to *setAdaptor*(...), *wx* answers *w5* whenever *wx* is or will reach a state different from the one for which *wx* has been enquired by *w5*. Otherwise, *wx* does not answer hence attempting to block *w5*.

Procedure *Ack* is performed to suitably handle some method calls by other components that, performing such method calls, can lead to forbidden states. For example, component *C7* can lead to a deadlock when it calls the method *setX500name*(...) whereas *C5* cannot. For instance, when *C7* calls the method *setX500name*(...) through an instance of either *C2* or *C4* and, hence, *w7* asks for the permission to call such a method, *w5* gives (through procedure *Ack*) the permission to it whenever its state does not participate in a forbidden one. Otherwise, it attempts to block *w7*.

9. Related work

The approach for automatically assembling software components and adapting their externally observable interactions presented in this paper is related to a number of other approaches that have been considered in the literature.

In this section, we firstly discuss valuable work in the literature that, even though not strictly related to our work, provided us with useful background notions concerning the nature and the composition of software components; then, we relate with our previous works and other works closest to our approach.

In Arbab (2005a,b) and Arbab et al. (2006a,b) (and references therein), focussing on what components are and how they are to be constructed, the authors provide the reader with useful notions and subtleties concerning concepts such as component *composition*, *behavior*, *interaction*, *coordination* and *glue code*. The authors introduce a foundation model, called *Abstract Behavior Type*

```
//
// LOCAL WRAPPER FOR THE MINISTRY CERTIFICATE CLIENT "Cm" (i.e., C5 within
// SYNTHESIS). It is an AspectJ class whose aim is to define a pointcut for the
// method calls ''connect'' on C4, ''releaseWrapper'' on C3,''releaseX500'' on C4,
// ''setX500name'' on C4, ''setWrapper'' on C3, ''setX500name'' on C2, and
// ''setWrapper'' on C1 performed by Cm. It also forces Cm to ask the other
// components for the authorization to perform such method calls. Morevoer, it
// is also responsible to reflect the LTS of Cm by accordingly updating the
// value of the static member ''stateW5''.
//
package Local1Wrapper5; import java.net.*; import java.io.*;

public aspect Local1Wrapper5 {
   // Current state
   public static int stateW5 = 0;
   ...

   //
   // Pointcut for the method call ''connect'' on C4. It defines an advice of
   // type ''before'': AuthorizationRequest() will be performed before the
   // execution of the method call ''connect'' on C4 (i.e., it is a
   // synchronization message)
   //
   pointcut a() : call(* c4Obj.connect(..));
   before() : a() {
      AuthorizationRequest();
   }

   //
   // Advice of type ''after'': after method call ''connect'' on C4 has been
   // performed, the current state is updated according to the LTS of Cm
   // (i.e., the LTS of C5)
   //
   after() returning : a(){
      if (stateW5==0) {stateW5 = 3;}
   }

   //
   // Pointcut for the method call ''setWrapper'' on C3. It defines an advice of
   // type ''before'': AuthorizationRequest() will be performed before the
   // execution of the method call ''setWrapper'' on C3 (i.e., it is a
   // synchronization message)
   //
   pointcut b() : call(* c3Obj.setWrapper(..));
   before() : b() {
      AuthorizationRequest();
   }

   //
   // Advice of type ''after'': after method call ''setWrapper'' on C3 has been
   // performed, the current state is updated according to the LTS of Cm (i.e., the
   // LTS of C5)
   //
   after() returning : b(){
      if (stateW5==3) {stateW5 = 5;}
   }

   //
   // Pointcut for the method call ''setWrapper'' on C1. It defines an advice of
   // type ''before'': AuthorizationRequest() will be performed before the
   // execution of the method call ''setWrapper'' on C1 (i.e., it is a
   // synchronization message)
   //
   pointcut c() : call(* c1Obj.setWrapper(..));
   before() : c() {
      AuthorizationRequest();
   }

   //
   // Advice of type ''after'': after method call ''setWrapper'' on C1 has been
   // performed, the current state is updated according to the LTS of Cm
   // (i.e., the LTS of C5)
   //
   after() returning : c(){
      if (stateW5==3) {stateW5 = 4;}
   }

   //
   // Pointcut for the method call ''setX500Name''. It defines an advice of type
   // ''after'': after ''setX500Name'' has been performed, the current state is
   // updated according to the LTS of Cm (i.e., the LTS of C5). It authorizes
   // another component to perform the method call ''setX500Name''
   //
   pointcut d() : call(* c2Obj.setX500Name(..));
   after() returning : d() {
      if (stateW5==4) {stateW5 = 3;}
      Ack();
   }

   // The rest of the code is analogous to what has been done above
   pointcut e() : call(* c4Obj.setX500Name(..));
   after() returning : e() {
      if (stateW5==5) {stateW5 = 3;}
      Ack();
   }

   pointcut f() : call(* c4Obj.releaseX500(..));
   after() returning : f() {
      if (stateW5==3) {stateW5 = 2;}
      Ack();
   }

   pointcut g() : call(* c3Obj.releaseWrapper(..));
   after() returning : g() {
      if (stateW5==2) {stateW5 = 0;}
      Ack();
   }

   ...
}
```

(ABT), for both components and their composition, as a higher-level alternative to *Abstract Data Type*. In their view, component-based systems consist of component instances and their connectors, *i.e.*, *glue code*, both of them modeled by ABTs. Clearly distinguishing between coordination and computation, they coined the term *exogenous coordination* for meaning *coordination from outside*. As a concrete application of the ABT model, the authors describe an exogenous coordination model and language, called Reo. By considering common basic concepts that all exogenous coordination protocols deal with, the Reo language enables the compositional construction of complex coordinators model, called *connectors*, from simpler ones (Arbab, 2002, 2003). We share the idea that coordination languages offer a looser inter-component semantic dependency with respect to the method invocation semantics in object oriented paradigms. However, in our work, we are interested in wrapping ready-to-use black-box components in order to intercept their (externally observable) interaction and to suppress certain messages for enforcing the correct/desired interaction.

In Passerone et al. (2002), a game theoretic strategy is used for checking whether incompatible component interfaces can be made compatible by inserting a converter between them. This approach is able to automatically synthesize the converter. Contrarily to what we have presented in this paper, the synthesized converter is a centralized adaptor.

Our research is also related to Yellin and Strom (1997) in the area of protocol adaptor synthesis. The main idea 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, although the work described in Yellin and Strom (1997) provides the foundational base for protocol specification and protocol adapters by considering both syntactic incompatibilities and protocol mismatches, the described approach does not address the automatic synthesis of the adaptor. Note that also our approach can be easily extended to address syntactic incompatibilities between component interfaces. We refer to Tivoli and Autili (2006) for details concerning such an extension.

In another work by some of the authors (Inverardi et al., 2005), it is showed how to generate a distributed adaptor by exploiting an approach to the definition of distributed Intrusion Detection Systems (IDS). Analogously to the approach described in this paper, the distributed adaptor is derived by *splitting* a pre-synthesized centralized one in a set of local wrappers (each of them local to each component). The work in Inverardi et al. (2005) represents a first attempt for distributing centralized adaptors and it has two main disadvantages with respect to the current approach: (a) the method requires a more complex (in time and space) process for pre-synthesizing the centralized adaptor. In fact, it does not simply model all the possible component interactions (like our centralized glue adaptor), but it has to model the component' interactions that are deadlock-free and that satisfies the specified desired behavior (P_{LTS}). In that approach, in fact, the glue adaptor is generated and, afterwards, a suitable synchronous product with P_{LTS} is performed. This longer process with respect to the current approach might also lead to a final bigger centralized adaptor. (b) The adopted solution realize distribution but not parallelism. The distributed local wrappers realize, in fact, the strict distribution of the obtained centralized adaptor by means of the pre-synthesizing step. This means that, since the centralized adaptor cannot parallelize its contained traces, the interactions of the local wrappers maintain this behavior.

In Sen et al. (2004), the authors show how to monitor safety properties locally specified (to each component). They observe the system behavior simply raising a *warning message* when a violation of the specified property is detected. Our approach goes beyond simply detecting properties by also allowing their enforcement. In Sen et al. (2004) the best thing that they can do is to reason about the global state that each component *is aware of*. Note that, such a global state might not be the actual current one and, hence, the property could be considered guaranteed in an "*expired*" state. Furthermore, they cannot automatically detect deadlocks.

Despite the growing number of research works in the area of component adaptation and assembly (see Canal et al., 2006; Brogi et al., 2004; Passerone et al., 2002; Yellin and Strom, 1997 and references therein), very few tools have been proposed to support automatic synthesis of the *actual* composition code for a set of black-box components. Moreover, many of them do not support *industrial* component technologies and frameworks such as, *e.g.*, Microsoft COM/DCOM, Enterprise Java Beans (EJB).

10. Conclusion and future work

In this paper we have presented an approach to automatically assemble concurrent and distributed component-based systems by synthesizing distributed adaptors. Our method extends our previous work described in Tivoli and Autili (2006) that permitted to automatically synthesize centralized adaptors for component-based systems.

The method described in this paper allows us to derive a distributed implementation of the centralized adaptor and, hence, it enhances *scalability*, *fault-tolerance*, *efficiency*, *parallelism* and *deployment*.

We successfully validated the approach on a portion of an industrial case study. We have also implemented it as an extension of our SYNTHESIS tool (Tivoli and Autili, 2006).

The state explosion phenomenon suffered by the centralized glue adaptor A still remains an open problem. A is required to detect the last chance nodes that are needed to automatically avoid deadlocks. Indeed when the deadlocks can be solved in some other ways (*e.g.*, using timeouts) or P_{LTS} ensures their avoidance, generating A is not needed. Local wrappers may add some overhead in terms of messages exchanged. In practical cases, where usually many real parallel computations are allowed, the overhead is negligible since additional communications are much less then standard ones. As future work, whenever A is required, an interesting research direction is to investigate the possibility of directly synthesizing the implementation of the distributed adaptor without producing the model of the centralized one. Further validation by means of other real-scale case studies would be interesting.

Acknowledgements

Comments and suggestions of anonymous referees are gratefully acknowledged.

Appendix A. LTS-based formalism and formal definitions

In this appendix, we summarize the relevant definitions regarding the specification language that is used (within our approach) to specify the externally observable behavior of a component (hence the trace-based semantics of its interaction) and of the AFA- and CABA-systems' behavior (see Section 3.1). This specification formalism uses LTSs (Keller, 1976).

Moreover, as introduced in Sections 3.2 and 4, a simple extension to the LTS syntax allow the SYNTHESIS user to easily specify the desired interaction behavior that is required for the realization of the system's purposes.

A.1. Labelled transition systems

Let *Act* be the universal set of observable actions, and $Act_\tau = Act \cup \{\tau\}$, where τ denotes an internal action that is not *observable* to a component's environment.

Definition 5 (*LTS*). An LTS L is a quadruple (S, T, D, s_0), where S is a finite set of states, $T \subseteq Act$ is a set of transition labels (*i.e*, actions) called the *alphabet* of L, $D \subseteq S \times (T \cup \{\tau\}) \times S$ is the transition relation and $s_0 \in S$ is the initial state. L is finite if D is finite and L is empty if D is empty. We will make use of the following notation: $g \xrightarrow{\alpha} h \Leftrightarrow (g, \alpha, h) \in D$.

An LTS $L = (S, T, D, s_0)$ is *non-deterministic* if $\exists\ (s, a, s'),(s, a, s'') \in D$: $s' \neq s''$, otherwise L is *deterministic*. It is worth mentioning that the SYNTHESIS tool deals with both deterministic and non-deterministic LTSs hence letting to the user the flexibility of modeling the behavior of a component by means of either a deterministic or, when needed, a non-deterministic LTS.

Definition 6 (*Trace*). Let $L = (S, T, D, s_0)$ be an LTS, $t = \mu_{i+1}\ \mu_{i+2} \dots \mu_n$ is a *trace* of L is iff there exists a sequence of states $s_i, \dots, s_n \in S$ such that $s_i \xrightarrow{\mu_{i+1}} s_{i+1} \cdots \xrightarrow{\mu_n} s_n$, $i \geqslant 0$, $n > i$. The *empty trace* is denoted by ϵ.

The set of all traces of the LTS L starting from a state s_i is denoted as $Tr(L, s_i)$.

Definition 7 (*Normalized trace*). Given a trace $t = \tau^* \mu_{i+1} \tau^* \dots \tau^* \mu_n\ \tau^*$ in $\mathrm{Tr}(L, s_i)$, the *normalized* version of t is the trace $t^\tau = \mu_{i+1} \dots \mu_n$. Note that, the normalized version of τ^* is the empty trace ϵ.

Given the set of traces $\mathrm{Tr}(L, s_i)$ of an LTS L, we denote the corresponding *set of normalized traces* as $\mathrm{Tr}(L, s_i)^\tau$.

A.2. Component behavior

By referring to Definition 7, the externally observable behavior of a component can be defined as follows:

Definition 8 (*Trace-based semantics of the component externally observable behavior*). The *externally observable behavior of a component* modeled by means of an LTS $C_i = (S, T, D, s_0)$ (modeling the interaction of the component with its expected environment) is

$$\mathrm{Tr}(C_i, s_0)^\tau$$

LTSs can be used to define finite state systems (Taubner, 1989). For our purposes we assume that all systems we deal with are finite state systems. Note that, in our context, this is not a restriction. We are dealing with black-box components (that can be seen as *discrete event reactive systems*), each of them exporting through its interface a finite number of "operations". In our model, each operation of a component interface can be seen as a point of interaction of the component with its expected environment (*e.g.*, an observable action in an automaton). If we would model all the possible externally observable component interactions with a "classical" automaton instead of an LTS, we would also have accepting states. Indeed, for our purposes, what matters about a particular component interaction is not whether it drives the automaton in an accepting state (since we cannot detect this due to the black-box nature of the component) but whether the automaton is able to perform the corresponding sequence of actions interactively. Thus, we should consider an automaton in which *every* state is an accepting state (Milner, 1989; Hopcroft and Ullman, 1979) (*i.e.*, an LTS). A consequence is that if an automaton accepts a particular component interaction seen as a sequence of component interface operation invocations (*i.e.*, a trace of actions in our model), then it also accepts any initial part of that interaction/sequence. In other words, due to the finiteness of the set of component interface operations, although all the possible component interactions can be infinite we can always finitely represent them since the language built over the component interface operations (*i.e.*, the model of the component interaction behavior) is prefix-closed (Hopcroft and Ullman, 1979). Prefix-closed languages are generated by prefix-grammars that describe exactly all regular languages. It is well known that regular languages are always accepted by finite-state automata. Furthermore, note that dealing with time is out of the scope of this work, *i.e.*, we do not consider real-time systems (although they might be still modeled using finite-state models). We refer to Tivoli et al. (2007) for a work-in-progress version of SYNTHESIS dealing with real-time systems. Thus, due to our component interaction model and to the fact that we deal with black-box components, it is sufficient to consider finite state systems for dealing with all the systems we are interested in.

In order to model component-based systems, LTSs can be combined using the LTS parallel composition operator. In the literature, several variants of the operator have been defined. The one used here (see Definition 9) has an *interleaving semantics*. That is, if α is an observable action (*i.e.*, $\alpha \neq \tau$) of an LTS L_i, then α is executed simultaneously with the *complementary action* $\bar{\alpha}$ of an LTS L_j (with $i \neq j$) producing an internal action τ at the level of the parallel composition. Synchronization of actions is thus determined by the alphabets of the component LTSs. An action β of an LTS L_i for which no complementary action exists in an LTS L_j (with $i \neq j$), is executed only by L_i and will not be synchronized, hence, producing the same action β at the level of the parallel composition (these actions are called *independent actions*). Analogously, the internal action τ is executed by exactly one component LTS at a time. By referring to the notation used by the SYNTHESIS tool (see for instance Section 4), the receive message `?Cl.a` is a complementary action of the send message `!Cl.a`.

A.3. Component LTSs parallel composition

In the following we formally define the concept of parallel composition of component LTSs. Firstly, for the sake of simplicity, we give the formal definition by just considering two LTSs, then we give the same definition by considering the general case of two or more LTSs.

Definition 9 (*Parallel composition (two LTSs)*). Let $L_1 = (S_1, T_1, D_1, s_0^1)$, and $L_2 = (S_2, T_2, D_2, s_0^2)$ be two LTSs, their *parallel composition* is the LTS $L_1|L_2 = (S, T, D, s_0)$, where

- **States**: $S \subseteq S_1 \times S_2$
- **Root**: $s_0 = (s_0^1, s_0^2)$
- **Labels**: $T = T_1 \cup T_2 \cup \{\tau\}$
- **Synchronization**: $((s_1, s_2), \tau, (s_1', s_2')) \in D \Longleftrightarrow \exists \alpha, \bar{\alpha} \in T_1 \cup T_2 : (s_1, \alpha, s_1') \in D_1 \wedge (s_2, \bar{\alpha}, s_2') \in D_2 \wedge \alpha \neq \tau$
- **Interleaving**: $((s_1, s_2), \beta, (s_1', s_2)) \in D \Longleftrightarrow \exists \beta \in T_1 : \quad (s_1, \beta, s_1') \in D_1 \wedge (\bar{\beta} \notin T_2 \vee \beta = \tau)$ (the rule Interleaving has a symmetric version that is not given since its definition is trivial)
- **Reachability**: $\forall s \in S, \exists t \in \mathrm{Tr}(L_1|L_2, s_0)$ leading to s.

In practice, the parallel composition operator "|" combines the behaviors of two LTSs by synchronizing their *shared/common actions* and *interleaving* their *non-shared* and *internal actions*. Note that *SYNTHESIS* takes as input LTSs that do not contains internal actions. These actions are considered only for parallel composition purposes.

Moreover, the isolated part of the parallel composition that is not reachable from the initial state is ignored, as it has no semantic significance.

Indeed, the parallel composition operator of two LTSs cannot be used to incrementally built the parallel composition of more than

two LTSs since it is not associative, *e.g.*, $(L_1|L_2)|\ L_3$ is not necessarily equal to $L_1|(L_2|L_3)$. Thus, for the general case of more than two LTSs, the following definition must be used. That is, Definition 10 is used, in general, for defining the behavior of an AFA- and CABA-system.

Definition 10 (*Parallel composition (general case)*). Let $L_1 = (S_1, T_1, D_1, s_0^1)$, …, $L_n = (S_n, T_n, D_n, s_0^n)$ be n LTSs, their *parallel composition* is the LTS $L_1| \ldots |L_n = (S, T, D, s_0)$, where

- **States**: $S \subseteq S_1 \times \ldots \times S_n$
- **Root**: $s_0 = (s_0^1, \ldots, s_0^n)$
- **Labels**: $T = \bigcup_{i=1}^{n} T_i \cup \{\tau\}$
- **Synchronization**:
  $((s_1, \ldots, s_n), \tau, (s_1', \ldots, s_n')) \in D \Longleftrightarrow \exists i, j \in \{1, \ldots, n\} : i \neq j, \alpha, \bar{\alpha} \in \bigcup_{t=1}^{n} T_t : \wedge (s_i, \alpha, s_i') \in D_i \wedge (s_j, \bar{\alpha}, s_j') \in D_j \wedge \alpha \neq \tau \wedge \forall k \in \{1, \ldots, n\} : k \neq i, j \rightarrow s_k' = s_k$
- **Interleaving**: $((s_1, \ldots, s_n), \beta, (s_1', \ldots, s_n')) \in D \Longleftrightarrow \exists i \in \{1, \ldots, n\}, \beta \in T_i : (s_i, \beta, s_i') \in D_i \wedge (\forall j \in \{1, \ldots, n\}, j \neq i \rightarrow s_j' = s_j \wedge (\bar{\beta} \notin \bigcup_{k=1, k \neq i}^{n} T_k \vee \beta = \tau))$
- **Reachability**: $\forall s \in S, \exists t \in Tr(L_1| \ldots |L_n, s_0)$ leading to s.

A.4. AFA system

By exploiting Definitions 10,11 follows:

Definition 11 (*AFA-system behavior*). The *behavior of an AFA-system* (see Section 3.1), composed by a set of n components (synchronizing on common actions) that are modeled by the LTSs $C_1 = (S_1, T_1, D_1, s_0^1)$, …, $C_n = (S_n, T_n, D_n, s_0^n)$, is

$\mathrm{Tr}((C_1| \ldots |C_n), (s_0^1, \ldots, s_0^n))$

Its *externally observable behavior* is

$\mathrm{Tr}((C_1| \ldots |C_n), (s_0^1, \ldots, s_0^n))^{\tau}$

A.5. CABA system

Definition 12 (Relabeling function). Let $L = (S, T, D, s_0)$ be an LTS, $L[w_i]$ is the LTS L relabeled by the *relabeling function* w_i (for some $i \in \mathcal{N}$) and $L[w_i] = (S, T[w_i], D[w_i], s_0)$, where

- $T[w_i] = \{\alpha\_i | \alpha \in T\}$
- $D[w_i] = \{(s, \alpha\_i, s') | \alpha \neq \tau, (s, \alpha, s') \in D\} \cup \{(v, \tau, v') |\ (v, \tau, v') \in D\}$

Definition 13 (*Glue adaptor LTS*). Let $C_1 = (S_1, T_1, D_1, s_0^1)$, …, $C_n = (S_n, T_n, D_n, s_0^n)$ be n component LTSs, their *glue adaptor LTS* is the LTS $A_{glue} = (S, T, D, s_0^{\mathrm{glue}})$, where

- **States**: $S \subseteq S_1 \times \ldots \times S_n$
- **Root**: $s_0^{\mathrm{glue}} = (s_0^1, \ldots, s_0^n)$
- **Labels**: $T = \bigcup_{i=1}^{n} T_i[w_i]$
- **Strictly I/O behavior**: $((s_1, \ldots, s_n), ?\alpha\_i, (s_1', \ldots, s_n')), ((s_1', \ldots, s_n'), !\alpha\_j, (s_1'', \ldots, s_n'')) \in D \Longleftrightarrow \exists i, j \in \{1, \ldots, n\} : i \neq j \wedge (s_i, \alpha, s_i') \in D_i \wedge (s_j, ?\alpha, s_j'') \in D_j \wedge \forall k, h \in \{1, \ldots, n\} : k \neq i \wedge h \neq j \rightarrow s_k' = s_k \wedge s_h'' = s_h'$
- **Reachability**: $\forall s \in S, \exists t \in \mathrm{Tr}(L, s_0^{\mathrm{glue}})$ leading to s.

It is worth noticing that the *strictly I/O behavior* of the *glue adaptor LTS* in Definition 13 models the strictly sequential nature of the adaptor, *i.e.*, each received message is forwarded strictly to the right component.

By exploiting Definition 13 it is straightforward to define the behavior of a CABA-system (see Section 3.1) for a set of n components assembled by a glue adaptor.

Definition 14 (*CABA-system behavior*). Given the LTSs $C_1 = (S_1, T_1, D_1, s_0^1)$, …, $C_n = (S_n, T_n, D_n, s_0^n)$ modeling the behavior of the n components, and given the LTS $A_{\mathrm{glue}} = (S, T, D, s_0^{\mathrm{glue}})$ modeling the strictly sequential behavior of the glue adaptor, the *behavior of the CABA-system* is modeled by

$\mathrm{Tr}((C_1[w_1]| \ldots |C_n[w_n]|A_{\mathrm{glue}}), (s_0^1, \ldots, s_0^n, s_0^{\mathrm{glue}}))$

Its *externally observable behavior* is

$\mathrm{Tr}((C_1[w_1]| \ldots |C_n[w_n]|A_{\mathrm{glue}}), (s_0^1, \ldots, s_0^n, s_0^{\mathrm{glue}}))^{\tau}$

A.6. Deadlock

We will refer to sink states of an LTS as *deadlock states*. A deadlock state models the fact that a deadlock has occurred in the associated component/system. An LTS is *deadlock-free* if it does not have deadlock states.

Definition 15 (*Deadlock state*). Let $L = (S, T, D, s_0)$ be an LTS, $s_i \in S$ is a *deadlock state* of L iff $\nexists s_j \in S$ such that $s_i \xrightarrow{\alpha} s_j$. In other words, a *deadlock state* is a sink state.

Definition 16 (*Forbidden trace and forbidden state*). Let $L = (S, T, D, s_0)$ be an LTS and let $t = \mu_{i+1}\ \mu_{i+2} \ldots \mu_n$ be a *trace* in $\bigcup_{s \in S} \mathrm{Tr}(L, s)$ such that $s_i \xrightarrow{\mu_{i+1}} s_{i+1} \ldots \xrightarrow{\mu_n} s_n$, then t is a *forbidden trace* of L iff s_n is a deadlock state and $\forall_{i \leqslant j < n},\ \nexists s_j \xrightarrow{\mu_{j+1}'} s_{j+1}' : \mu_{j+1} \neq \mu_{j+1}' \wedge s_{j+1} \neq s_{j+1}'$. All the states $s_i, \ldots, s_n \in S$ are *forbidden states*.

We denote by FT_L the set of all forbidden traces of an LTS L and by FS_L the set of all forbidden states of L. Note that, FS_L is the set of all deadlock states and of all the states within forbidden traces necessarily leading to deadlock states; a forbidden trace is a trace that starts at a node which has no transitions that can avoid a forbidden state and thus necessarily ends in a deadlock state (*i.e.*, a sink).

Definition 17 (*Last chance state*). Let $L = (S, T, D, s_0)$ be an LTS and let $ft = \mu_{i+1}\mu_{i+2} \ldots \mu_n$ be a *forbidden trace* in FT_L such that $s_i \xrightarrow{\mu_{i+1}} s_{i+1} \ldots \xrightarrow{\mu_n} s_n$, if there exists a trace $t = \mu_i, \mu_{i+1}, \mu_{i+2}, \ldots, \mu_n \in ((\bigcup_{s \in S} \mathrm{Tr}(L, s)) \backslash FT_L)$ such that $s_{i-1} \xrightarrow{\mu_i} s_i \xrightarrow{\mu_{i+1}} s_{i+1} \ldots \xrightarrow{\mu_n} s_n$, then s_{i-1} is a *last chance state* for *ft*.

We denote by LC_L the set of all last chance states of the LTS L.

A.7. Desired behavior

A desired behavior LTS is defined over a specific set of actions that are semantically equivalent to component actions. Let $C_1, \ldots, C_n$ be the components given as input to our method, and let $C_1[w_1] = (S_1, T_1[w_1], D_1[w_1], s_0^1), \ldots, C_n[w_n] = (S_n, T_n[w_n], D_n[w_n], s_0^n)$ be the corresponding relabeled LTSs (see Definition 12 in Appendix A.5). Let $U = \bigcup_{i=1}^{n} T_i[w_i]$ be the universal set of component actions. U is ranged over by $\alpha, \alpha_1, \alpha_2, \ldots$. Unlike actions in a component LTS, each action in U has associated an identifier specifying which component (in the CFA-system) performs that action. For instance, `C3.methodl_1` models the action `C3.methodl` performed by the component `C1`. In the following we formally define the syntax of the action label of the desired behavior LTS. The syntax is formalized by means of a grammar that can be used to generate action labels for a desired behavior LTS.

Definition 18 (*Desired behavior actions syntax*). The universal set $DbAct_U$ of desired behavior actions over U, is the set of action labels generated by the following grammar:

$l ::= \alpha | Neg(\alpha) | ?true\_\# | \{Neg(\alpha_1), \ldots, Neg(\alpha_m)\} | [\alpha_1, \ldots, \alpha_k]$

where *Neg* is a relabeling function over U such that

$Neg(\alpha) = ? - a_1$ if $\alpha \in U$ and $\alpha = ?a_1$
$Neg(\alpha) = -a_1$ if $\alpha \in U$ and $\alpha = a_1$

The syntax of the action labels in $DbAct_U$ is similar to the syntax of the action labels in a relabeled component LTS except for two kinds of actions: (i) a *universal action* (*i.e.*, ?*true*_#) which models any possible component action (*i.e.*, any action in U) and (ii) a *negative action* which models any possible component action except for the negative action itself; for instance, the negative action -`C3.methodl_l` models all the actions in U different from `C3.methodl_l`. Moreover, action labels in $DbAct_U$ can be a simple formula obtained as logical "AND" or "OR" composition of action labels. The "AND" operator can be applied only to negative actions and it is denoted by means of the notation {...}. The "OR" operator can be applied only to regular actions (*i.e.*, component actions) and, for it, the notation [...] is used (see Fig. 4). The semantics of action labels in $DbAct_U$ is defined as follows, by means of a certain notion of semantic equivalence between regular actions and desired behavior actions. Note that, component actions in a relabeled component LTS are a particular case of desired behavior actions (*i.e.*, they are regular actions α of Definition 18).

Definition 19 (Desired behavior actions semantics). Let $\alpha \in U$ and $l \in DbAct_U$, we say that α *shares the meaning of* l or, simply, α *matches* l (denoted by $\alpha \cong_U l$) if there exists a binary relation $\cong_U$ relating α and l (*i.e.*, $(\alpha, l) \in \cong_U$ with $\cong_U \subseteq U \times DbAct_U$) such that

- $\alpha_1 \cong_U \alpha_2 \Leftrightarrow \alpha_1 = \alpha_2$
- $\alpha_1 \cong_U Neg(\alpha_2) \Leftrightarrow \alpha_1 \ncong_U \alpha_2$
- $\alpha \cong_U ?true\_\#$
- $\alpha \cong_U \{Neg(\alpha_1), \ldots, Neg(\alpha_m)\} \Longleftrightarrow \bigwedge_{i=1}^{m} (\alpha \ncong_U \alpha_i)$
- $\alpha \cong_U [\alpha_1, \ldots, \alpha_m] \Longleftrightarrow \bigvee_{i=1}^{m} (\alpha \cong_U \alpha_i)$

A desired behavior LTS (with respect to an universal set U of relabeled component LTS actions) is defined over a set of transition labels that is a sub-set of $DbAct_U$.

Definition 20 (*Desired behavior LTS*). Let $C_1, \ldots, C_n$ be n component LTS, let $C_1[w_1] = (S_1, T_1[w_1], D_1[w_1], s_0^1), \ldots, C_n[w_n] = (S_n, T_n[w_n], D_n[w_n], s_0^n)$ be their corresponding relabeled LTSs, and let $U = \bigcup_{i=1}^{n} T_i[w_i]$; a *desired behavior LTS over* $DbAct_U$ for $C_1, \ldots, C_n$ is a *well-formed* and, possibly, *non-deterministic* LTS $P_{\mathrm{LTS}} = (S_P, T_P, D_P, p_0)$ where S_P is the set of states, T_P is the set of transitions labels such that $T_P \subseteq DbAct_U$, D_P is the set of transitions, and p_0 is the initial state.

A.8. Trace containment check

Our method checks whether enforcing a desired behavior is possible or not through a *trace containment check* between the desired behavior LTS and the adaptor LTS where the deadlocking interactions have been removed.

Definition 21 (*Trace containment under* $\cong_U$). Let $L_1 = (S_1, T_1, D_1, s_0^1)$ and $L_2 = (S_2, T_2, D_2, s_0^2)$ be two LTSs, and let $T_1, T_2 \subseteq DbAct_U$ for some universal set of observable actions U; we say that $\mathrm{Tr}(L_1, s_0^1)^\tau$ is *contained under* $\cong_U$ in $\mathrm{Tr}(L_2, s_0^2)^\tau$ (written $\mathrm{Tr}(L_1, s_0^1)^\tau \subseteq_{\cong_U} \mathrm{Tr}(L_2, s_0^2)^\tau$) if and only if $\forall t \in \mathrm{Tr}(L_1, s_0^1)^\tau : \exists t' \in \mathrm{Tr}(L_2, s_0^2)^\tau : t \cong_U t'$.

By abusing notation, we extend the *action complement operator* to sets of transition labels, traces, and set of traces as follows: let $T \subseteq Act_\tau$ be a set of transition labels, then $\overline{T} = \{\bar{\alpha} | \alpha \in T\} \cup \{\tau \mid \tau \in T\}$; let $t = \beta_1 \ldots \beta_m$ be a trace, then $\bar{t} = \overline{\beta_1} \ldots \overline{\beta_m}$; for the empty trace ϵ we consider $\bar{\epsilon} = \epsilon$; let $L = (S, T, D, s_0)$ be an LTS and let $\mathrm{Tr}(L) = \{t | t \in (T^* \cup \{\epsilon\})\}$, then $\overline{\mathrm{Tr}(L)} = \{\bar{t} | t \in \mathrm{Tr}(L)\}$.

Given a desired behavior LTS $P = (S_P, T_P, D_P, s_0^P)$ and the adaptor LTS $K_{df} = (S, T, D, s)$ where the deadlocking traces have been removed, this check is used to verify whether $\mathrm{Tr}(P, s_0^P)^\tau \subseteq_{\cong_U} \overline{\mathrm{Tr}(K_{df}, s)^\tau}$ or not. This check is implemented by a suitable notion of *refinement* (Milner, 1989). Refinement, in general, formalizes the relation between two LTSs at different level of abstractions. Refinement is usually defined as a variant of *simulation*. In this paper, we use a suitable notion of *strong simulation* (Milner, 1989) to check a refinement relation between two LTSs with observable actions over $DbAct_U$ (*i.e.*, P and K_{df}). To do this, we use the matching operator "$\cong_U$" as action comparison operator of the simulation.

Definition 22 (*Simulation under* $\cong_U$). Let $L_1 = (S_1, T_1, D_1, s_0^1)$ and $L_2 = (S_2, T_2, D_2, s_0^2)$ be two LTSs, and let $T_1, T_2 \subseteq DbAct_U$ for some universal set of observable actions U; a relation $\leqslant_{\cong_U} \subseteq S_1 \times S_2$ is a *strong simulation under* $\cong_U$, or *simulation under* $\cong_U$ for short, where $s \leqslant_{\cong_U} v$ if and only if $\forall s' \in S_1 : \forall l \in T_1 : s \xrightarrow{l} s' \Rightarrow \exists v \in S_2 : v \xrightarrow{l} v' \wedge l \cong_U l' \wedge s' \leqslant_{\cong_U} v'$. We say L_2 simulates under $\cong_U$ L_1, written $L_1 \leqslant_{\cong_U} L_2$, if and only if $s_0^1 \leqslant_{\cong_U} s_0^2$.

Theorem 1 (a trivial variant of the analogous theorem described in Milner (1989)). *Let* L_1 *and* L_2 *be LTSs where* $L_1 \leqslant_{\cong_U} L_2$, *then* $Tr(L_1, s_0^1)^\tau \subseteq_{\cong_U} Tr(L_2, s_0^2)^\tau$.

Appendix B. Happened-before relation and partial ordering

In this appendix, we briefly recall basic notions related to distributed systems and recall the well know *time-stamp method* that we use within the Synthesis approach.

Without loss of generality, we assume that the components to be assembled are uniquely identified and assigned to different processors (residing in different interconnected machines). Note that, this assumption can be maintained to model abstract parallelism when more than one component is assigned to a single processor. For the sake of clarity, in this paper we also assume that each component is single-threaded and, hence, all its send and receive events can be totally ordered to constitute a set of traces (see Appendix A). Note that, this is not a restriction since a multi-threaded component can always be modeled as a set of single-threaded (sub-)components simultaneously executed.[10]

Considering synchronization on common actions (send events and corresponding receive events), interaction among components is modeled by *interleaving* of traces (see Section 4 for a simple example and the Definitions 9, 10 and 13 in Appendix A). This means merging two or more traces such that the independent events (*i.e.*, not common) from different traces may occur in any order in the resulting trace, while the events within the same traces retain their order. A trace resulting from this merge is usually called a *linearization* (Ben-Ari (1990)).

It is worth noting that, in such a concurrent and distributed context, we cannot assume either a single physical clock or a set of perfectly synchronized ones in order to determine whether an event *a* occurs before an event *b* or vice versa. We then need to define a relationship among the system events by abstracting from both the absolute speed of each processor and the absolute time. In this way, we ignore any absolute time scale and assume each event to be executed in a time unit.[11] For example, the sequence of events depicted in Fig. 20a can have different execution times

[10] The new version of Synthesis allows the user to also model multi-threaded components.

[11] This abstraction is acceptable since we are not interested to real-time systems for which the absolute time is relevant.

based on different processor, but we will model it as in Fig. 20b where each event occurs in a prefixed time unit.

By taking into account the *law of causality* (Lamport, 1978) (*i.e.*, a message can be received only after it has been sent), it is possible to define the *happened-before* relation (denoted by $\rightarrow$) on a set of events as follows:

Definition 23 (*Happened-before relation*). Let E_i be the set of events of a component C_i and let $E = \bigcup_{i=1}^{n} E_i$ be the set of all possible system events, then the happened-before relation $\rightarrow \subseteq E \times E$ on the set of events of a system is the smallest relation satisfying the following three conditions:

(1) if e_1 and e_2 are events of the same component and e_1 is executed before e_2, then $e_1 \rightarrow e_2$;
(2) if e_1 is the event corresponding to the sending of a message m by a component and e_2 is the event corresponding to the receiving of m by another component, then $e_1 \rightarrow e_2$;
(3) if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3$.

For each pair of distinct events $e_1, e_2 \in E$, if e_1 does not occur before e_2 and e_2 does not occur before e_1 we will write $e_1 \nrightarrow e_2$ and $e_2 \nrightarrow e_1$. In this case we say that a and b are concurrently executed. Obviously, $e_i \nrightarrow e_i$ for each $e_i \in E$ since e_i cannot occur before itself.

We refer to Lamport (1978) for a detailed discussion about the concepts of concurrency and happened-before relation. Note that, the happened-before relation is only a partial ordering of the events in distributed systems. In particular, when modeling synchronous communication, we can consider send and receive events of the same message to occur simultaneously. Thus, in this case, we might restrict the partial order to only send-events.

B.1. Time-stamps and total ordering

In this section, we briefly describe the well known *time-stamp method* (Lamport, 1978) that we use to implement the happened-before relation within the algorithms described in Section 5. Whereas the events onto a same component can be totally ordered, the relation with the events of different components is not always well defined. By the time-stamp method it is possible to define a global order among the whole events (send/receive) exchanged through the assembled system for each component. In other words, each component can establish a total order among its generated events and the events received from other components.

The idea is to associate a time-stamp to each event. This is just a number that each sending component associates to its messages. Locally, time-stamps are sequentially generated, one for each event. Whenever a receive event e' occurs at a component c, such a component is able to determine a local order among its own events and e'. If c was intended to send a message e with a time-stamp lower than the one associated with e', then e is processed before e', *i.e.*, $e \nrightarrow e'$. Moreover, in order to try to synchronize with the sending component, c will use the received time-stamp plus one as next time-stamp, *i.e.*, the next message that c wants to send will be associated with the updated time-stamp. On the other hand, if the time-stamp associated to e is bigger than the received one, c considers $e' \nrightarrow e$. Finally, if the time-stamp associated with e is equal to the time-stamp associated with e', then there is concurrency. In this case, in order to avoid any ambiguity, an order among components can be a priori fixed. If e' was sent by a component $c' < c$ in the fixed order, than $e' \nrightarrow e$, otherwise $e \nrightarrow e'$, hence obtaining a total order among events. The time-stamp method is well described in Lamport (1978). We use it in what follows since it is a lightweight and easy way to validate mutual exclusion, freedom from deadlock and starvation, and it allows one to state other useful and crucial properties for distributed systems (*e.g.*, minimum overhead in performing synchronizing communication).

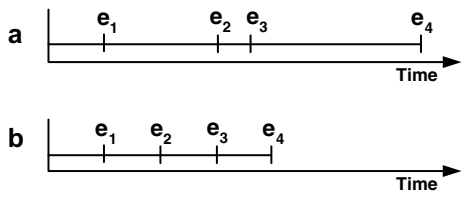


Fig. 20. Time unit and sequence of events.

B.2. Time-stamps implementation

While exchanging messages, when needed, the standard time-stamp method is used in our approach in order to avoid problems of synchronization. In this way, an ordering among dependent messages is established and starvation problems are addressed. Such a method establishes a total order at each wrapper among the sent and received events of the corresponding component. For this purpose we also need an ordering among components. Such an *a priori* fixed order solves concurrency problems arising when two events with associated the same time-stamp must be compared. Whenever a local wrapper of a component C_x receives a message with associated a time-stamp *ts* from the wrapper of a component C_y, it makes use of the following simple procedure in order to update the current time-stamp *TS* that it associates to the events generated by C_x.

The following procedure is used in our implementations in order to update the time-stamp of a component.

Procedure (*UpTS(component:* C_y, timestamp: *ts*;)).

1: **if** (*TS* = *ts* AND *y* < *x*) OR *TS* < *ts* **then**
2: *TS* := *ts* + 1;
3: **end if**

We now provide the pseudo-code related to procedures *Ask* and *Ack* in which the use of the time-stamps and of Procedure *UpTS* has been made explicit.

Procedure (*Ask(action:* α;)).

Procedure.

1: Let C_x be the current component that would perform action α and let S_{C_x} be its current state and p be the current state of P_{LTS};
Let $\langle t_i \rangle_x^{UA}$ be the ith tuple contained in the table $W_{C_x}^{UA}$ and $\langle t_i \rangle_x^{UA}[j]$ be its jth element;
2: *flag_forbidden* := 0;
3: **if** $\exists i | \langle t_i \rangle_x^{UA}[1] == p \&\& \langle t_i \rangle_x^{UA}[2] == \alpha$ **then**
4: **if** α appears in some pair of $W_{C_x}^{LC}$ **then**
5: **for** every entry $\langle S, \alpha \rangle \in W_{C_x}^{LC}$ **do**
6: $i := 1$;
7: *TS* ++;
8: **while** no "ACK, α,*ts*" received && $i \leqslant n$ **do**
9: Let $S \equiv \langle S_{C_1}, \ldots, S_{C_n} \rangle$; W_{C_x} asks to local wrapper W_{C_i} if it is in or approaching[12] the state S_{C_i} with associated *TS*;

[12] C_i is performing its *Ask* procedure with respect to an action that leads C_i to S_{C_i}.

10:
    i ++;
11: **end while**
12: **if** $i > n$ **then**
13: WAIT for an "ACK, α, ts" message of one enquired component C_y;
14: **end if**
15: $UpTS(C_y, ts)$;
16: **if** $i > n$ **do**
17: $i := n$;
18: **end if**
19: **for** $j := 1$ **to** i **do**
20: send "UNBLOCK, α, TS" to W_{C_i};
21: **end for**
22: **end for**
23: **end if**
24: TS ++;
25: **if** $\langle t_i \rangle_x^{UA}[1] = \langle t_i \rangle_x^{UA}[3]$ **then**
26: **for** each component $C_j \in \langle t_i \rangle_x^{UA}[4]$ **do**
27: send "BLOCK, TS" to W_{C_j};
28: **end for**
29: perform action α;
30: **for** each component $C_j \in \langle t_i \rangle_x^{UA}[5]$ **do**
31: send "UNBLOCK, $\langle t_i \rangle_x^{UA}[3]$, TS" to W_{C_j};
32: **end for**
33: **else**
34: perform action α;
35: **end if**
36: **end if**

[Ack(last chance state: S; action: α; timestamp: $ts1$;)]

1: Let W_{C_y} be the local wrapper (performing this Ack) that was enquired with respect to the state S and the action α that C_x would perform.
2: $UpTS(C_x, ts1)$;
3: **if** C_y is not in $S\&\&W_{C_y}$ didn't ask for permission to get in S **then**
4: send "ACK, α, TS" to W_{C_x} that allows C_x to perform the action α;
5: **if** C_y would reach S with the next hop **then**
6: WAIT for "UNBLOCK, α, $ts2$" from W_{C_x};
7: **end if**
8: **else**
9: once C_y is not in S send "ACK, α, TS" to W_{C_x} that allows C_x to perform the action α;
10: **if** no "UNBLOCK, α, $ts2$" from W_{C_x} has been received **then**
11: WAIT for it;
12: **end if**
13: $UpTS(C_x, ts2)$;
14: **end if**

References

Arbab, F., 2002. A channel-based coordination model for component composition. Technical Report SEN-R0203, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, February.

Arbab, F., 2003. Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14 (3), 1–38.

Arbab, F., 2005. Composition by Interaction. Inaugural Lecture, Leiden University, October.

Arbab, F., 2005b. Abstract behavior types: a foundation model for components and their composition. Science of Computer Programming 55, 3–52.

Arbab, F., 2006b. A behavioral model for composition of software components. RSTI – L'objet, Coordination and Adaptation Techniques 12 (1), 33–76.

Arbab, F., Baier, C., Sirjani, M., Rutten, J.J.M.M., 2006a. Modeling component connectors in reo by constraint automata. Science of Computer Programming 61 (2), 75–113.

Autili, M., Inverardi, P., Tivoli, M., Garlan, D., 2004. Synthesis of 'correct' adaptors for protocol enhancement in component-based systems. In: Proceedings of SAVCBS'04 at FSE, pp. 79–86.

Autili, M., Flammini, M., Inverardi, P., Navarra A., Tivoli, M., 2006. Synthesis of concurrent and distributed adaptors for component-based systems. In: Proceedings of the European Workshop on Software Architecture (EWSA). LNCS, vol. 4344, Springer-Verlag, Berlin/Heidelberg. pp. 17–32.

Autili, M., Inverardi, P., Navarra, A., Tivoli, M., 2007. SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, pp. 784–787.

Becker, S., Overhage, S., Reussner, R., 2004. Classifying software component interoperability errors to support component adaption. In: Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C. (Eds.), Component-based Software Engineering, Proceedings of the 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24–25, Lecture Notes in Computer Science, vol. 3054, Springer, pp. 68–83.

Becker, S., Brogi, A., Gorton, I., Overhage, S. Romanovsky, A., Tivoli, M. 2006., Towards an engineering approach to component adaptation. Chapter in Dagstuhl Seminar 04511: Architecting Systems with Trustworthy Components. LNCS, vol. 3938, Springer-Verlang, Berlin/Heidelberg. pp. 193–215.

Ben-Ari, M., 1990. Principles of Concurrent and Distributed Programming. Prentice Hall.

Brand, D., Zafiropulo, P., 1983. On communicating finite-state machines. Journal of the ACM 30 (2).

Brandin, B.A., Wonham, W.M., 1994. Supervisory control of timed discrete-event systems. IEEE Transactions on Automatic Control 39 (2).

Brogi, A., Canal, C., Pimentel, E., 2004. Behavioral types and component adaptation. In: Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST2004).

Canal, C., Poizat, P., Salaün, G., 2006. Synchronizing behavioral mismatch in software composition. In: Proceedings of the International Conference on Formal Methods for Open Object-based Distributed Systems. LNCS, vol. 4037.

Compare, D., Inverardi, P., Wolf, A.L., 1999. Uncovering architectural mismatch in component behavior. Science of Computer Programming (33), 101–131.

Crnkovic, I., Larsson, M., 2002. Building Reliable Component-based Software Systems. Artech House Boston, London.

European Commission 6th Framework Program – 2nd Call Galileo Joint Undertaking: Cultural Heritage Space Identification System (CUSPIS). <http://www.cuspis-project.info>.

Hopcroft, J., Ullman, J., 1979. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley.

Horwich, P., 1990. Wittgenstein and kripke on the nature of meaning. Mind and Language 5, 105–121.

Inverardi, P., Tivoli, M., 2003. Software architecture for correct components assembly. In: Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture. LNCS, vol. 2804. Springer.

Inverardi, P., Mostarda, L., Tivoli, M., Autili, M., 2005. Synthesis of correct and distributed adaptors for component-based systems: an automatic approach. In: Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)- Long Beach, CA, USA.

ITU-T: The X500 naming service: ISO/IEC 9594-1. <http://www.itu.int/home/index.html>.

ITU Telecommunication Standardisation Sector, 1996. ITU-T Reccomendation Z.120. Message Sequence Charts. (MSC'96), Geneva.

Keller, R., 1976. Formal verification of parallel programs. Communications of the ACM 19 (7), 371–384.

Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21 (7), 558–565.

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

Nicola, R.D., Vaandrager, F., 1995. Three logics for branching bisimulation. Journal of the ACM 42 (2), 458487.

Passerone, R., de Alfaro, L., Heinzinger, T., Sangiovanni-Vincentelli, A.L., 2002. Convertibility verification and converter synthesis: two faces of the same coin. In: Proceedings of the International Conference on Computer Aided Design (ICCAD). San Jose, CA, USA.

Ramadge, P.J., Wonham, W.M., 1987. Supervisory control of a class of discrete event processes. SIAM Journal of Control and Optimization 25 (1).

Schmidt, H.W., Reussner, R.H., 2002. Generating adapters for concurrent component protocol synchronisation. In: Proceedings of the Fifth IFIP International Conference on Formal Methods for Open Object-based Distributed Systems.

Sen, K., Vardhan, A., Agha, G., Rosu, G., 2004. Efficient decentralized monitoring of safety in distributed systems. In: Proceedings of the International Conference on Software Engineering (ICSE), Edinburgh, UK.

Szyperski, C., 2004. Component Software: Beyond Object-oriented Programming. Addison-Wesley.

Taubner, D., 1989. Finite representations of CCS and TCSP programs by automata and petri nets. LNCS, vol. 369.

Tivoli, M., Autili, M., 2006. SYNTHESIS: a tool for synthesizing "correct" and protocol-enhanced adaptors. RSTI L'Objet Journal 12 (1), 77–103.

Tivoli, M., Fradet, P., Girault, A., Goessler, G., 2007. Adaptor synthesis for real-time components. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), Member of ETAPS 2007, Braga, Portugal. LNCS, vol. 4424. Springer-Verlang, Berlin/Heidelberg, ISBN 978-3-540-71208-4, pp. 185–200.

Uchitel, S., Kramer, J., Magee, J., 2004. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. ACM Transactions on Software Engineering and Methodology (TOSEM) 13 (1), 37–85.
Yakimovich, D., Travassos, G., Basili, V., 1999. A classification of software components incompatibilities for COTS integration. Technical Report, Software Engineering Laboratory Workshop, NASA/Goddard Space Flight Center, Greenbelt, Maryland.
Yellin, D., Strom, R., 1997. Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems 19 (2), 292–333.
Zaremski, A., Wing, J., 1995. Signature matching: a tool for using software libraries. ACM Transaction on Software Engineering and Methodology 4, 146–170.

Marco Autili received a first-class honors degree in Computer Science on April 2004, and a PhD in Computer Science on April 2008 from the University of L'Aquila, Computer Science Department. Currently, he is a Research Assistant at the Computer Science Department of the University of L'Aquila. In the field of Component Based Software Engineering, his research interests include: Formal Methods to the Automatic Adaptation and Composition of Software Components and their adaptation w.r.t. resource consumption in a given execution environment; Service Oriented Architectures; Formal Requirements Specification.

Leonardo Mostarda received both his Computer Science degree and his Ph.D. from the University of L'Aquila in 2002 and 2006, respectively. In 2006 he participated in the CUSPIS European project at the University of L'Aquila as a post-doctoral researcher. He started in 2007 as a research associate at Imperial College London. His research interests include distributed systems, monitoring systems, and security.

Alfredo Navarra received the master degree in Computer Science at the University of L'Aquila in 2000, and the Ph.D. degree in Computer Science at the University of Rome "La Sapienza" in 2004. From 2003 to 2004, he joint the MASCOTTE project team at the INRIA institute of Sophia Antipolis as PhD student and PostDoc. In 2005, he was PostDoc at the Computer Science Department at University of L'Aquila. In 2006, he joint the LaBRI at University of Bordeaux as PostDoc. In 2007, he joint the Department of Electrical and Information Engineering at University of L'Aquila, and he has become Assistant Professor at the Mathematics and Computer Science Department at University of Perugia. His research interests include algorithms, computational complexity, networking and distributed computing.

Massimo Tivoli received a first-class honors degree in Computer Science on 2001, and a PhD in Computer Science on 2005 from the University of L'Aquila, Computer Science Department. Currently, he is an Assistant Professor at the Computer Science Department of the University of L'Aquila. His research interests include Formal Methods to the Automatic Adaptation and Composition of Software Components, Component Based Software Engineering, Software Architectures, and Service Oriented Architectures.