Towards an architectural approach for the dynamic and automatic composition of software components

Antonio Bucchiarone[1], Patrizio Pelliccione[2], Andrea Polini[1], Massimo Tivoli[3]

[1] Istituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche
via Moruzzi, 1 – 56124 Pisa, Italy
{antonio.bucchiarone, andrea.polini}@isti.cnr.it
[2] Software Engineering Competence Center, University of Luxembourg
6, rue R. Coudenhove-Kalergi, Luxembourg
patrizio.pelliccione@uni.lu
[3] Dipartimento di Informatica
Università dell'Aquila, I-67010
L'Aquila, Italy
tivoli@di.univaq.it

Abstract. Roughly speaking, a component-based software system is an assembly of reusable components, designed to meet the system requirements identified during the architecting phase. Components are specified, designed and implemented with the intention to be reused, and are assembled in various contexts in order to produce a multitude of software systems. The correct functioning of a component-based system depends on both the properties of the single components and the properties of the assembly code. One main problem in component assembly is related to the ability to establish properties on the assembly code by only assuming a limited knowledge of the single component properties. Our answer to this problem is a Software Architecture (SA) based approach in which standard and synthesis-oriented architectural analysis are combined together in order to efficiently and correctly assemble a system out of a set of already implemented components. Our method model-checks the SA of the system respect to desired requirements and assumes the SA as starting point for the synthesis-oriented process. This process is performed to automatically derive a dynamic and correct component assembly. Although in the system life-cycle components change the SA does not change (if the requirements still remain unmodified). Thus, the SA can be used as starting point for deriving adaptors to correctly replace (at run-time) components in the composed system.

1 Introduction

The latest years have been characterized by the increasing pervasiveness of information processing systems. Many new application fields have been explored and many new software systems have been developed. On one hand this testifies

that the trust on software has generally grown and, as direct consequence, it is more and more used in risky activities. On the other hand, this also makes it mandatory to enhance the "-ilities" of the produced software, while assuring a high dependability, or otherwise the consequences can be catastrophic. This trend does not give any sign to be going to finish yet, and as a result the complexity of software systems is continuously rising. At the same time the software developers, to stay competitive, need to cope with the constant reduction of the time-to-market.

The answer to these challenges is being sought on the potential to obtain complex systems by assembling prefabricated and adequate pieces of software called "components". To make real this vision different research lines, within the *Component Based Software Engineering* (CBSE) area, have received great interest in the last decade, trying to address different aspects of "componetization", such as communication and definition of component technologies.

Finally, particularly relevant have been the studies conducted to model this kind of systems and that have raised the interesting subject referred as *Software Architecture* (SA). Component assembling can result in architectural mismatches when trying to integrate components with incompatible interaction behavior [7]. According to [21], the 50% of bugs are detected after component integration, not during component development. In this context, the notion of SA assumes a key role since it represents the reference skeleton used to compose components and let them interact. In the SA domain, the interaction among the components is represented by the notion of software "connector". Beyond the concepts of component and connector there is also another basic element that characterizes an SA, i.e., the properties against which an implementation of the specified SA has to be validated.

Two common approaches can be used to carry on this validation step:

1. *Standard Analysis:* the analysis process is based on checking if the specified properties hold in the assembled system via model checking technique.
2. *Synthesis-oriented Analysis:* a code synthesis technique can be defined in order to generate the "correct" assembly code for the (pre-selected and -acquired) components forming the specified system. This code is derived in order to force the composed system to exhibit only the specified properties.

In a Component Based (CB) setting, standard analysis techniques can be applied either on a model of the system defined as output of a designing step, or on a model of the system derived by unifying models associated to each component forming the system. Such component models should be generally provided by the developer. Both alternatives present some risks when applied to a CB development process. The first option, in fact, assumes that the running version of the system will be completely conforming with that defined in the SA. This conformance refers both to the expected coordination and behavior of each component. Unfortunately this is not always the case and often the real system presents mismatching coordination and component behavior. Instead, the application of static analysis to a model derived by the real implementation

of the system is mainly hindered by the dimensions of the model that in general contains too much low level details.

The application of a Synthesis-oriented approach to directly derive a correct assembly is certainly relevant but one main problem arises. The unification of the different component models is a process that generally suffers the well-known state-explosion phenomenon.

In this work, we describe an SA based approach that combines standard and synthesis-oriented analysis together in order to efficiently and correctly assemble a system out of a set of already implemented components. Moreover, our approach allows the system to be able to evolve, at run-time, respect to architectural updates at component level (i.e., adding, removing and replacing components). In this work we consider only the replacement of components as possible architectural update. At design-time, the architectural model of the system to be assembled is validated with respect to a set of properties of interest and, by taking into account the analysis results, incrementally refined (i.e., standard analysis). At compile-time, an initial version of the assembled system is built. This is done by using the validated component models as properties to be enforced on the interaction behavior of the actual (pre-acquired) components. In doing so, when needed, an adaptor for each actual component is automatically synthesized (i.e., synthesis-oriented analysis). At run-time, if an actual component is replaced by a different one its corresponding adaptor detects this change and, subsequently, it is re-synthesized in order to force the interaction of the changed component to exhibit only the validated behavior. The combination of standard and synthesis-oriented analysis is performed by combining two previously developed approaches from some of the authors. One is implemented in the CHARMY tool [6, 14], which is for performing standard analysis of an SA model. The other one is implemented in the SYNTHESIS tool [9, 18, 19], which is for performing synthesis-oriented analysis. The two approaches take advantage from each other. In fact, since SYNTHESIS suffers the well-known state-explosion phenomenon, it can exploit the architectural specification previously validated through CHARMY to perform adaptation locally to each component rather than at level of global system interactions. This makes the approach implemented by SYNTHESIS more feasible in practice since, in this case, the problem that SYNTHESIS has to face is highly reduced in number of states of the search space. On the other hand, SYNTHESIS adds to CHARMY automation in assembling the designed and validated component-based system. In fact, in CHARMY this task is completely delegated to the developer.

The paper is organized as follows. Section 2 points out the motivations for an SA based approach to the dynamic and automatic composition of components. Section 3 recalls the approaches implemented in CHARMY and in SYNTHESIS. Section 4 describes the method our approach is based on by distinguishing three main phases of its utilization. Section 5 validates the described approach by means of an industrial case study concerning with a cooling water pipe management system. Section 6 discusses related works. Section 7 concludes and discusses future work.

2 An SA based approach to the dynamic and automatic component assembly: motivations

SA emerged in the first nineties as a way of organizing and reasoning on software system in a similar way of what has been done in other more mature engineering disciplines [15]. Indeed, many software companies have understood the importance of SA modeling in order to obtain a better quality software reducing time and cost of realization. However, putting SA in practice, software architects have learned that the SA production and management is, in general, an expensive task. Thus the introduction of SA into an industrial development life-cycle is justified only by an extensive use of these artifacts able to produce adequate benefits, such as the production of a good quality software reducing at the same time realization costs.

To use an SA just as a documentation artifact produces only an SA documentation but this phase is completely untied from the other phases and typically further modifications of the system are not updated to the SA design. The result of this development process is that the SA design quickly become obsolete.

Many works have instead demonstrated the usefulness of an SA definition for discovering systems problems during the first phases of software development [3, 17]. In fact, when the SA is validated with respect to requirements, it can be used as the starting point for any other analysis and exploited also to drive the next phases of the system development.

Building on this trend, in the reminder of the paper we propose a method which model-checks the SA of a component-based system with respect to desired requirements and assumes the SA as starting point for a synthesis-oriented process. This process is performed to automatically derive a dynamic and correct component assembly. Although in the system life-cycle components change the SA does not change (if the requirements still remain unmodified). Thus, the SA can be used as starting point for deriving adaptors to correctly replace (at run-time) components in the composed system.

3 SA Analysis Tools

In this section we briefly recall aspects of CHARMY and SYNTHESIS that are relevant for the approach presented in this paper. Further details about them will eventually be discussed in Section 5 during the presentation of our case study.

3.1 Charmy: a tool for designing and model-checking architectural specifications

CHARMY [6, 14] is a project whose goal is to apply model-checking techniques to validate the SA conformance to certain properties. In CHARMY the SA is specified through state diagrams used to describe how architectural components

behave. Starting from the SA description CHARMY synthesizes, through a suitable translation into Promela (the specification language of the SPIN [8] model checker) an actual SA complete model that can be executed and verified in SPIN. This model can be validated with respect to a set of properties, e.g., deadlock, correctness of properties, starvation, etc., expressed in Linear-time Temporal Logic (LTL) [13] or in its Büchi Automata representation [4]. Instead of writing directly temporal properties that is a task inherently error prone, CHARMY permits to describe temporal properties by using sequence diagrams that are successively translated into a temporal property representation understandable by SPIN. The model checker SPIN, is a widely distributed software package that supports the formal verification of concurrent systems permitting to analyze their logical consistency by on-the-fly checks, i.e., without the need of constructing a global state graph, thus reducing the complexity of the check. It is the core engine of CHARMY and it is not directly accessible by a CHARMY user.

The state machine-based formalism used by CHARMY is an extended subset of UML state diagrams: labels on arcs uniquely identify the architectural communication channels, and a channel allows the communication only between a pair of components. The labels are structured as follows: '['*guard*']'*event*'('*parameter_list*')"/'op_1';'op_2';' $\cdots$ ';'op_n where *guard* is a boolean condition that denotes the transition activation, an *event* can be a message sent or received (denoted by an exclamation mark "!" or a question mark "?", respectively), or an internal operation (τ) (i.e., an event that does not require synchronization between state machines). Both sent and received messages are performed over defined channels *ch*, i.e., connectors. An event can have several parameters as defined in the parameters list. $op_1, op_2, \cdots, op_n$ are the operations performed when the transition fires.

Sequence Diagrams are described using a UML notation, stereotyped so that: (i) each rectangular box represents an architectural component, (ii) each arrow defines a communication line (a channel) between two components. Between a pair of messages we can select if other messages can occur (loose relation) or not (strict relation). Message constraints are introduced to define a set of messages that must never occur in between the message containing the constraint and its predecessor or successor. Messages are typed as *regular messages* (optional messages), *required messages* (mandatory messages) and *fail messages* (messages representing a fault).

3.2 Synthesis: a tool for synthesizing failure-free component adaptors

SYNTHESIS [9, 18, 19] is a tool for assembling component-based systems out of a set of already implemented heterogeneous components by ensuring the correct functioning of the system at level of component interaction protocol.

Its aim is to analyze and prevent interaction mismatches (i.e., deadlocks, livelocks, etc.) that can arise from component composition. It implements an architectural "coordinator"-based approach. The idea is to build applications by

assuming a formal architectural model of the system representing the components to be integrated and the connectors (i.e., communication channels) over which the components will communicate. Using SYNTHESIS the developer can derive in an automatic way, from the COTS components, the code that implements a new component that has to be inserted into the composed system. This new component implements a software coordinator. The coordinator mediates the interaction among components in order to prevent possible integration failures.

For its aims, SYNTHESIS assumes that some specification of the externally *"observable"* behavior of each actual component (forming the system to be assembled) is available in the form of state diagrams. For externally *"observable"* behavior of the component, we mean the behavior of the component in terms of the messages exchanged with its expected environment. Under this assumption SYNTHESIS is able to automatically derive the assembly code (i.e., the coordinator's actual code) for a set of components. This code is derived in order to obtain a deadlock-free system.

4 Method description

Our method can be described by distinguishing three main phases of its utilization as showed in Figure 1. We now look at each phase in detail.

4.1 Design-time phase: validating the system SA

At design-time, our approach assumes that an architectural specification of the system to be assembled is provided in terms of state and sequence diagrams. State diagrams are used to describe how architectural components behave, sequence diagrams to describe the behavioral properties against which the composed system must be validated. By taking into account this initial specification, the goal of the design-time phase is to perform standard analysis to incrementally obtain the correct (with respect to the specified properties) specification of the actual components that should be acquired in order to assembly the specified and validated system. More precisely CHARMY, starting from the state diagrams representing components behavior, synthesizes, through a suitable translation into Promela, an actual SA model that can be executed and verified in SPIN. This model can be validated with respect to a set of properties expressed in extended UML2.0 sequence diagrams and automatically translated in Büchi automata.

At the end of this step we have a system specification (and, hence, also the specification of the components forming it) which respects the properties of interest, expressed in terms of sequence diagrams.

4.2 Compile-time phase: component composition through static adaptation

In order to correctly build component-based systems by practicing CBSE and to fully utilize its well known advantages, the next step after SA-level analy-

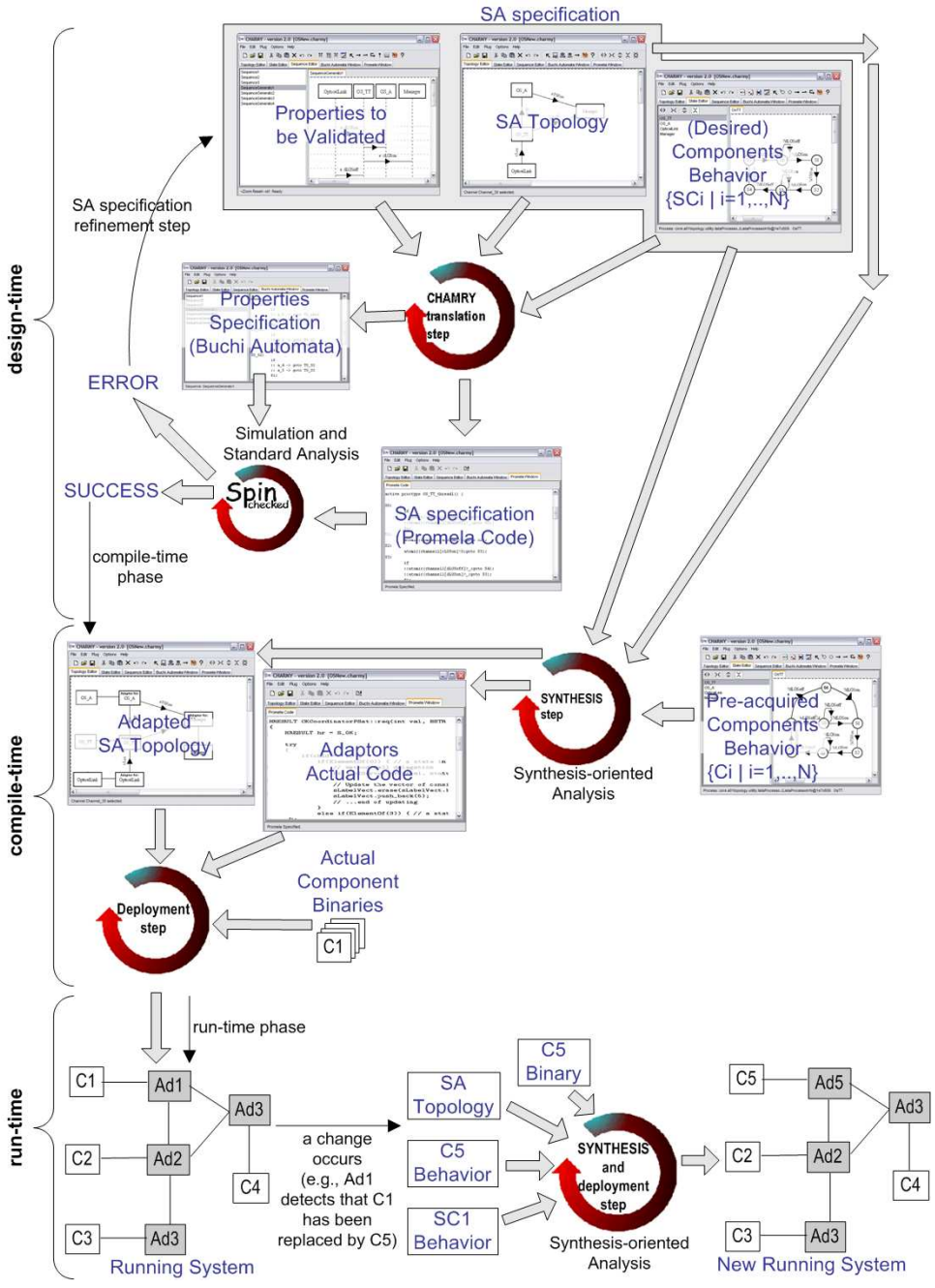

Fig. 1. 3-phase method

sis is to assemble the designed and validated component-based system out of a set of already implemented third-party or *Commercial-Off-The-Shelf (COTS)* components. These third-party components are selected by looking at the functionality that they implement. They have to "contain" the same functionality implemented by the corresponding component in the system SA model. However, in practice, this criterion is not enough since the actual component interaction behavior (i.e., its protocol) might not fit the one as specified in the validated SA model. Thus, *static* (i.e., at compile-time) adaptation can be done to eliminate the resulting mismatches. The compile-time phase of our approach goes in this direction. Once selected and acquired the actual components providing the functionalities of the components in the system SA specification, a set of adaptors (possibly one for each actual component) is mechanically build. This is done by using SYNTHESIS. Thus, we assume to have a state diagrams specification of the actual component interaction behavior. From this specification and from the specification of the corresponding components in the system SA model, the static adaptation phase restricts (or, when needed, even extends) the behavior of the actual component in order to fit the specified one. In other words, the composition of the actual component with the synthesized adaptor behaves as the specified component specified in the system SA model and, hence, a first version of the composed system has been automatically build and it is correct by construction. It is important to note that the state machines of CHARMY and the state machines of SYNTHESIS are exactly the same.

Note that although, in general, the synthesis process suffers the state-explosion phenomenon, our approach makes it feasible. In fact, it exploits the system SA model (previously validated) in order to perform adaptation locally to each component rather than at level of global system interactions. In this way, within our approach, the synthesis process has to face a problem that has a reduced complexity in terms of its "space-size".

4.3 Run-time phase: component composition through dynamic adaptation

Before speaking about dynamic adaptation we introduce some definition in order to well define the system that we are able to manage. By considering the component and connector architectural elements, we will make use of the following definitions [5]:

Weakly-Closed System: A Weakly-Closed System is a system with a fixed number of components.

Closed System: A Closed System is a system with a fixed number of component instances and fixed connectors.

Weakly-Opened System: A Weakly-Opened System is a system with variable number of component instances and with fixed connectors.

Opened System: An Opened System is a system with variable connectors and number of component instances.

Opposite to static adaptation, dynamic adaptation is performed at run-time. In static adaptation, detection and correction can be performed before the sys-

tem is run, hence yielding systems correct by construction. This can be achieved because systems are Closed. If we want to deal with Weakly-Closed, Weakly-Opened or Opened Systems, in which the components and their protocols may change over time, dynamic adaptation is required. The method that we present in this paper does not give a solution for dynamic adaptation of Weakly-Opened and Opened Systems but give a solution for Weakly-Closed ones.

The dynamic adaptation process might be incremental. A first adaptor is built (if needed) at compile-time, and then adaptation has to evolve when things change in the system. This discussion is concerned with the run-time phase of our approach. We do not consider other possible system changes beyond component replacement and modification of component bindings.

During the execution of the system we keep stored both the actual component specification and the modeled component specification. Once an actual component is replaced with a different one, the dynamic adaptation phase (at run-time) re-synthesizes an adaptor for it and deploys this adaptor in the composed system. In this way, the initial version of the system is dynamically converted into a new one by still ensuring the validated properties.

5 Case study: a cooling water pipe system

The case study that we use in this paper to better explain and validate our method is concerned with the automatic assembly of a cooling water pipe management system that collects and correlates data about the amount of water that flows in different water pipes.

The water pipes are placed in two different zones, denoted by *p* and *p1*, and they transport water that has to be used to cool industrial machinery. The zone *p* (*p1*) is monitored by the server *C1* (*C2*). *C1* (*C2*) supports cooperative work and allows the access to a collection of data related to the water pipes that it monitors. *C1* (*C2*) implements the interface *I1* (*I2*). Since some of the water pipes do not include a *Programmable Logic Controller* (PLC) system, the two servers cannot always automatically obtain the data related to the water that flows in those water pipes. Therefore, *I1* (*I2*) provides the method *p* (*p1*) to get an exclusive access to the data collection related to the water that flows in the pipe of the zone *p* (*p1*). This allows a client to (i) read the data automatically stored by the server and (ii) manually update the report related to the water that flows in the pipes, which are not monitored by a PLC. Correspondingly, *I1* (*I2*) provides the method *FreeP* (*FreeP1*) to both publish the updates made on the data collection and release the access granted to it. Moreover, *I2* provides also a method *Connect* to authenticate the clients.

By referring to the method depicted in Figure 1, we want to assemble a client-server cooling water pipe management system formed by *C1*, *C2* as servers and two clients (called *C3* and *C4*). In doing so, we want to automatically ensure deadlock-freeness and other specified behavioral properties. Moreover, we wish to obtain a system which can tolerate component replacement at run-time.

Figure 2 shows the state machines that describe the *desired* behavior of the components *C1*, *C2*, *C3* and *C4*. As we can see in this figure the server *C2* contains also the connection functionality (above mentioned) required to access to the *p1* zone.

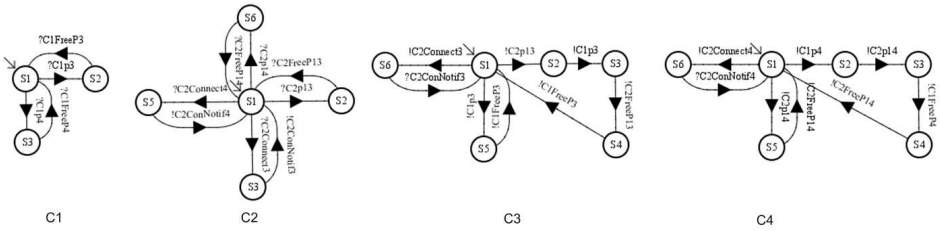

Fig. 2. State machines of the water pipe management system

These state machines are designed by using the CHARMY tool that generates the Promela code needed for the verification with SPIN. At the beginning is not sure that the desired behavior specified by the designer is correct (especially for large systems). Thus, the intention of the designer is to verify the correctness of its model in order to refines it and produce the correct specification of the system that must be assembled. In modeling the desired behavior of the components forming the system the designer distinguishes the method calls performed by *C3* and *C4* towards *C1* and *C2*. This is done by adding a suffix to each message label (i.e., message labels ending with "3" and "4" denote method calls performed by *C3* and *C4*, respectively).

The first verification that we make is the deadlock freeness. To do that we can use the basic checks of SPIN called *invalid end states*, without specifying any property. Unfortunately the result of the verification is not valid and it is caused by the components *C3* and *C4* that ask the access to the zones *p* and *p1* in a complementary way.

To solve the problem we modify the component *C3* as it is shown in Figure 3.A.

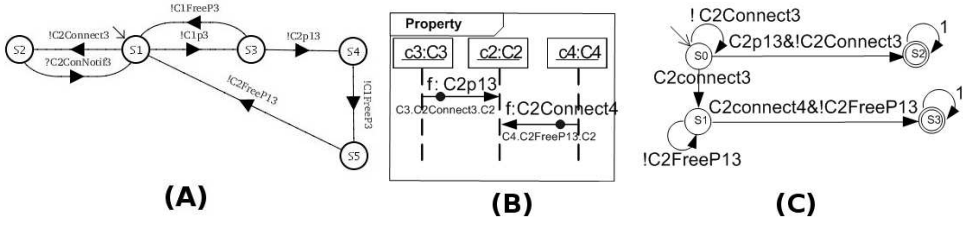

Fig. 3. (A) *C3* modified to ensure deadlock-freeness; (B) sequence diagram and (C) Büchi automaton of the desired property

The second step of verification is to check if the model satisfies the properties of interest. For the sake of simplicity, we make use only of one desired property. This property represents the desired interaction protocol for accessing the information related to the water flowing in the pipe of zone *p1*. That is, it is mandatory for *C3* to be authenticated before accessing to the information related to the zone *p1*. Thus, if *C3* is connected then *C4* cannot connect before that the granting ticket will be discarded. It happens after that C3 publishes its updates on the information related to the zone p1, represented with the action *C2FreeP13*.

The property is described in Figures 3.B and 3.C in terms of its corresponding PSC [2, 16] notation and Büchi automaton, respectively. PSC is a simple and graphical formalism for specifying temporal properties. For the sake of brevity we recall only the elements of PSC required to understand the property in Figure 3.B. As already introduced in Section 3.1 there are three different kind of messages and in this example we make use of two fail messages (messages prefixed by the label "f:"). We recall that fail messages are used to identify messages that should never be exchanged. We make use also of two constraints that impose "restrictions" on the set of messages that can be exchanged before the message considered and after its predecessor (the predecessor of the first message of a PSC is the startup of the system). In [2, 16] is presented also an algorithm, Psc2Ba able to translate PSC to Büchi automata. Psc2Ba is used to generate the Büchi automaton for the PSC representation of the desired property.

Coming back to the desired property, referring to the PSC notation, two are the messages considered: *C2p13* and *C2Connect4*. The first one has *C2Connect3* as constraint of the message that implements the restriction imposed to *C3* to make a connection before gaining the access to the *p1* zone. The second one has the message *C2FreeP13* as constraint. The meaning is that the component *C4* can connect only when the component *C3* leaves the *p1* zone. Charmy and its engine SPIN return a not valid result for this property, essentially caused by the *C2* component that is to simple and without logic, i.e., no order is imposed on the messages that this component exchanges with its environment.

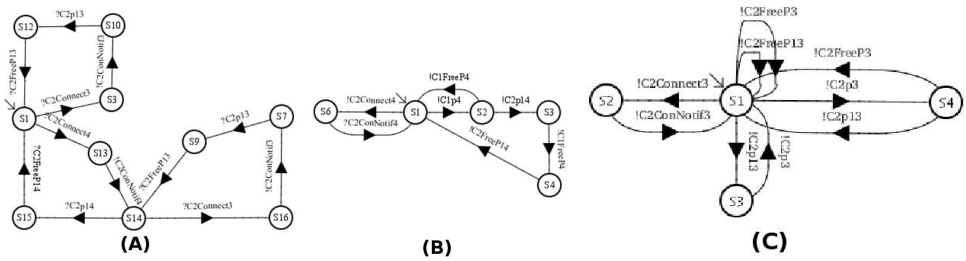

Fig. 4. (A) *C2* and (B) *C4* modified to ensure the desired property; (C) interaction behavior assumed by *AC3*

Thus the interaction behavior of *C2* and *C4* has to be changed. Figures 4.A and 4.B report the modifications made. Now the component *C2* contains explicitly an order for the messages exchanged with its environment, i.e., the connections of *C3* and *C4*, and the access for the same components to the *p1* zone. *C4* has been limited in accessing the *p1* zone in order to respect the constraints imposed by the new version of *C2*.

At this point the design-time phase of our method is terminated and we have obtained a correct specification of the system that we want to assemble. This system is formed by the components *C1* showed in Figure 2, *C3* showed in Figure 3.A, *C2* and *C4* showed in Figures 4.A and 4.B, respectively. The connectors that we consider are simple communication channels connecting each of *C3* and *C4* with both *C1* and *C2*.

Now, let us consider four COTS components that we have selected and acquired in order to assemble the validated system. Let us denote them as *AC1*, *AC2*, *AC3* and *AC4* that correspond to the actual version (available on the market) of *C1*, *C2*, *C3* and *C4*, respectively. We recall that to perform the compile-time phase of our method we assume that the usual interface specification of the COTS components (e.g., IDL specification) has been extended with extra-information related to the interaction behavior of each component with its expected environment. *AC1*, *AC2* and *AC4* behave as specified for *C1*, *C2* and *C4*, respectively. Unfortunately, as showed in Figure 4.C, *AC3* behaves in a different way respect to *C3*.

Note that *AC3* "contains" the component interactions specified by *C3*. That is, *AC3* exhibits all interactions of *C3* plus different ones. Thus, in this case, the compile-time phase of our method uses SYNTHESIS to automatically synthesize an adaptor for *AC3* (i.e., *AdtC3*) whose aim is to restrict the behavior of *AC3* in order to exhibit only the interactions specified by *C3*. That is, *AdtC3* is synthesized in such a way that the parallel composition of it with *AC3* behaves like *C3*. In other words, the implementation of the architectural component *C3* is automatically derived and distributed in two actual components: *AC3* acquired from third-parties and *AdtC3* automatically built by taking into account the behavioral specification of *AC3* and *C3*.

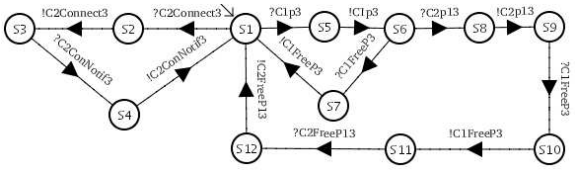

Fig. 5. Behavioral model of *AdtC3*

In Figure 5 we show the state machine automatically synthesized for *AdtC3*. Informally, this state machine is derived by automatically synthesizing partial views of *AdtC3* that reflect the behavior of the adaptor only related to the as-

sumptions made by *AC3* and *C3* on their expected environment. Finally, these partial views are "unified" and the state machine showed in Figure 5 is automatically obtained. Moreover the model of the adaptor is deadlock-free, i.e., it is synthesized in order to prevent possible deadlocks that might occur during the interaction between *AC3* and its environment (whose "reduced-in-space" model is represented by the state machine of *C3* where input actions have been converted into output ones, and viceversa). For the sake of brevity, here, we do not show the synthesis process in detail. Refer to [9] for a detailed description of the deadlock-free adaptor synthesis.

Using the same technique described in [19], from this state machine and by taking into account the information stored in its nodes and arcs, SYNTHESIS is able to derive the actual code implementing *AdtC3*. The compile-time phase concludes by assembling the COTS components and the adaptor together. At this point, a running implementation of the validated system is automatically obtained and it is correct by construction.

Note that if *AC3* "contains" only a sub-set of the set of interactions specified by *C3* (plus different ones) then performing adaptation by only restricting its behavior is not enough. In fact, in this case, the set of interactions specified by *AC3* must be also extended in order to perform the *C3* interactions that are missing in the *AC3* specification. In [20] such a *protocol-enhancing* technique is described and since it is out of the scope of this paper we refer to [20] for it.

During the compile-time phase, the implementation of the adaptor has been enriched with suitable mechanisms that makes the adaptor able to detect and react to the replacement of the component that it controls. A possible solution for this is discussed in Section 7. This is the starting point of the run-time phase of our method. Let us consider a scenario in which *AC3* is replaced (at run-time) by a different component (i.e., *AC5*). *AdtC3* catches the event associated to the replacement of *AC3* and triggers the event associated to the possible off-line synthesis of a new version of it (i.e., *AdtC5*). If *AC5* "contains" the interaction specified by *C3* (plus different ones) then it is not required to re-synthesize *AdtC3* (into *AdtC5*) because, by construction, it is already able to restrict the behavior of *AC5* to perform only the *C3* interactions. Thus, in this case, the system can evolve at run-time without performing any further adaptation. Otherwise, *AdtC5* is synthesized off-line by also using the above mentioned protocol-enhancing technique. During the off-line synthesis (and before that *AdtC3* is substituted by *AdtC5*) *AdtC3* rejects the messages from and towards *AC5*. This is done to prevent a possible failure due to, e.g., the execution of a request on a component which is not the expected one. This makes our adaptation process safe since it is able to prevent failures during the adaptation phase. Obviously, this implies that during the adaptation phase, if a failure occurs, the system might not temporarily progress. Thus, our current approach is not suitable for systems with "hard" timing constraints (e.g., real-time systems).

6 Related work

The architectural approach to the dynamic and automatic composition of software components presented in this paper is related to a large number of other problems that have been considered by researchers over the past two decades. For the sake of brevity we mention below only the works closest to our approach. The most strictly related approaches are concerned with the problem of dynamically composing and adapting software components.

As part of the RAPIDware project, Zhang et al. [22] introduced an aspect-oriented approach to add dynamic adaptation infrastructure to legacy programs in order to enable dynamic adaptation. They separate the adaptation concerns from the functional ones of the program, resulting in a clearer and more maintainable design. We believe that this concept of separation of concerns is crucial to perform adaptation especially when it has to be performed at run-time. In our approach, this concept is implemented by means of the architectural model that we impose on the SA of the system to be assembled. That is, each third-party component cannot directly communicate to the other third-party components in the system but all its interactions must go through its associated adaptor which, in turn, is connected to the other adaptors in the system.

Kulkarni et al. [11, 12] propose a distributed approach to compose distributed fault-tolerant components at run-time. They use theorem proving techniques to show that during and after an adaptation, the adaptive system is always in correct states with respect to satisfying specified transitional-invariants. Their approach, however, does not guarantee the "safeness" of the adaptation process in the presence of failures during the application of the adaptation strategy. Although our approach is not able to solve possible failures during the adaptation phase, differently from their work, we are able to prevent failures during the adaptation phase and, hence, we can guarantee a safe adaptation process.

Appavoo et al. [10] proposed a hot-swapping technique that supports run-time object replacement. In their approach, a quiescent state of an object is the state in which no other process is currently using any function of the object. We argue that this conditions is not sufficient in cases where a critical communication segment between two components includes a series of function invocations. Also, they did not address global conditions for safe dynamic adaptation.

Amano et al. [1] introduced a model for flexible and safe mobile code adaptation, where adaptations are serialized if there are dependencies among adaptation procedures. Their approach supports the use of assertions for specifying preconditions and postconditions for adaptation, where violations will cancel the adaptation or roll back the system to the state prior to the adaptation. Their work focuses on the dependency relationships among adaptation procedures, whereas our work focuses on dependency relationships among components.

7 Conclusions and future work

In this work we proposed an SA based approach for automatically assembling component-based systems out of a set of already implemented components. By

referring to Section 4.3, the component-based systems we deal with are Weakly-Closed. That is, the described approach allows the system to be able to evolve, at run-time, with respect to architectural updates at component level such as component replacement. The models that we use belong to state and sequence diagrams. The combination of standard and synthesis-oriented analysis is performed by combining two previously developed approaches from some of the authors (CHARMY, which is for performing standard analysis of an SA model, and SYNTHESIS, which is for performing synthesis-oriented analysis).

The approach that we proposed promotes engineering approaches that starting from high-level specifications allow for the design and the implementation of UML state and sequence diagrams, hence providing effective tool support for model analysis and code synthesis.

Several aspects need to be better investigate:

- how to manage the process of replacing components and the trigger event for that;
- when a component is replaced we have to be sure that the system still remain in a consistent state.

A possible solution could be the use of exception handling techniques and in particular *Architectural exceptions* that are exceptions that flow between two components. *Fault tolerance* is intended to preserve the delivery of correct services in the presence of active faults. It is generally implemented by error detection and subsequent system recovery. Error detection originates an error signal or message within the system.

Coming back to our context, supposing that a component need to be replaced, this activity could be represented as an exception that triggers the component replacement. System recovery techniques can be used to bring the system in a consistent state before components replacement. For instance, if the component that must be replaced is in execution, system recovery techniques can help the system to reach the state before the component execution.

Another aspect which is concerned with future work is the handling of architectural updates that go beyond the only replacement of components, e.g., adding and removing components. This would allow us to deal with Weakly-Opened and Opened systems.

References

1. N. Amano and T. Watanabe. A software model for flexible and safe adaptation of mobile code programs. *in Proceedings of the international workshop on Principles of software evolution*, ACM Press, 2002.
2. M. Autili, P. Inverardi, and P. Pelliccione. A graphical scenario-based notation for automatically specifying temporal properties. Technical report, Department of Computer Science, University of L'Aquila. Sumbitted for publication, 2005.
3. M. Bernardo and P. Inverardi. *Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods*. SFM-03:SA Lectures, LNCS 2804, 2003.

4. J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. of the International Congress of Logic, Methodology and Philosophy of Science*, pages pp 1–11. Standford University Press, 1960.
5. M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of architectural patterns. In *First European Workshop on Software Architecture - EWSA 2004*, 21-22 May 2004, St Andrews, Scotland.
6. Charmy Project. Charmy web site. http://www.di.univaq.it/charmy, February 2004.
7. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), November 1995.
8. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.
9. P. Inverardi and M. Tivoli. *Software Architecture for Correct Components Assembly - Chapter in: Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture*. Springer, LNCS 2804, Sept. 2003.
10. K. Hui J. Appavoo, C. A. N. Soules, and et al. Enabling autonomic behavior in systems software with hot swapping. *IBM System Journal*, vol. 42, no. 1, 2003.
11. S. Kulkarni and K. Biyani. Correctness of component-based adaptation. *in CBSE7*, May 2004.
12. S. S. Kulkarni, K. N. Biyani, and U. Arumugam. Composing distributed fault-tolerance components. *in DSN*, page pp. W127.W136, June 2003.
13. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.
14. P. Pelliccione. *CHARMY: A framework for Software Architecture Specification and Analysis*. PhD thesis, Computer Science Dept., University of L'Aquila, May 2005.
15. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
16. PSC home page: http://www.di.univaq.it/psc2ba, 2005.
17. D. J. Richardson and P. Inverardi. ROSATEA: International Workshop on the Role of Software Architecture in Analysis E(and) Testing. In *ACM SIGSOFT Software Engineering Notes, vol. 24, no. 4,*, July 1999.
18. Synthesis Project. Synthesis web site. http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.php, September 2004.
19. M. Tivoli and M. Autili. Synthesis: a tool for synthesizing correct and protocol-enhanced adaptors. *L'Object journal*, 12(1), 2005.
20. M. Tivoli and D. Garlan. Adaptor synthesis for protocol-enhanced component based architectures. *WICSA'05 (working session paper)*.
21. J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Softw.*, 14(4):73–83, 1997.
22. J. Zhang, B. H. C. Cheng, Z. Yang, and P. K. McKinley. Enabling safe dynamic component-based software adaptation. *In Architecting Dependable Systems III, LNCS. Springer-Verlag*, 2005.