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

Antonio Bucchiarone*, Andrea Polini
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

Patrizio Pelliccione, Massimo Tivoli
Dipartimento di Informatica
Università dell'Aquila, I-67010
L'Aquila, Italy
{pellicci,tivoli}@di.univaq.it

ABSTRACT

In a component-based software system the 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. However, this ideal scenario is not always the case, e.g., the integration with legacy components. In this context, one main problem in component assembly arises. It is related to the ability to automatically and efficiently (i.e., by reducing the state-explosion phenomenon) synthesize an assembly code for a set of, possibly incompatible, software components. Moreover, this assembly should be able to evolve when things change and to be *correct-by-construction*, i.e., despite the changes, it always ensures a set of properties of interest. In this paper we propose a Software Architecture (SA) based approach in which architectural analysis and code synthesis are combined together in order to efficiently and correctly assemble a system out of a set of already implemented components. The approach can be equally applied to efficiently manage the whole re-factoring of the system when one or more components needs to be substituted, still maintaining the required properties. The specified and validated system SA is used as starting point for the derivation of adaptors required to correctly replace components in the composed system. The approach is applied and validated over an explanatory example concerning with a "cooling water pipe" system.

1. INTRODUCTION

Software component composition is an important objective of software engineering. It promises components reuse and therefore productivity gains, because of shorter time-to-market and improved quality. 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 "componentization", such as communication and definition of component technologies. A component-based (CB) software system is an *assembly* of software components (usually implemented as either third-parties or in-house components), designed to meet the system requirements identified during the "architecting" phase. The studies conducted to model this kind of systems have been relevant and have raised the interesting subject referred as *Software Architecture* (SA). According to [24], 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, which is the system configuration. In other words, component and connectors can be composed together to make up different system configurations. In general, a different configuration guarantees different behavioral properties. Thus, a further element to take into account are the properties against which a SA configuration has to be validated.

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

1. *Architectural Analysis:* the analysis process is based on checking if the specified properties hold in the SA design of the assembled system via, e.g., model checking technique.
2. *Code Synthesis:* a code synthesis technique can be defined in order to generate the "correct" assembly code for the (preselected and -acquired) components forming the specified system. This code is derived in order to force the composed system to exhibit only the specified properties.

Both approaches have advantages and drawbacks. The architectural analysis approach assumes that the running version of the system will be completely conform with that defined in the SA, but unfortunately this is not always the case. The code synthesis approach generally produces a non-tractable model, given the large amount of low level details contained in it, resulting from the consideration of models representing real component implementations. This kind of problem generally leads to the well known state-explosion phenomenon, in which the dimension of the model and number of states in it prevent the applicability of any analysis technique to the whole model. This problem become particularly relevant when run-time refactoring are required, asking for a complete revision of the whole system. As showed in this paper our approach permits instead to limit run-time modifications to a local extent.

In this work, we describe an SA-based approach that combines architectural analysis and code synthesis together in order to ef-

*IMT Graduate School, Via San Micheletto, 3 - 55100 Lucca, Italy

ROSATEA'06 July 17, 2006, Portland, Maine, USA

ficiently and correctly assemble a system out of a set of already implemented components. First, the system's SA is validated and refined respect to a set of properties of interest (i.e., standard analysis). Second, an initial version of the composed system is built by taking into account the models of the components in the SA and the ones of the actual component selected and acquired on the market. The component integration code is automatically synthesized in order to be *correct-by-construction* (i.e., code synthesis). This code is derived in form of a set of adaptors, each of them for each component in the SA. Third, our approach allows the system to be able to evolve, at run-time, respect to architectural updates at component level (e.g., adding, removing and replacing components). These updates lead to modify (by substitution) the adaptors coordinating the components affected by the change. In this work we consider only the replacement of components as possible architectural update. Nevertheless the approach seems promising for the application to a more "extreme" version of run-time dynamicity.

The combination of architectural analysis and code synthesis is performed by combining two previously developed approaches from some of the authors. One is implemented in the CHARMY tool [8, 11] (architectural analysis) and one in the SYNTHESIS tool [22, 23] (code synthesis). The two approaches take advantage from each other. On one hand, CHARMY provides SYNTHESIS with an already validated system's SA. SYNTHESIS can exploit this system's SA to perform adaptation locally to each component rather than at level of global system interactions and, hence, reducing the state-explosion phenomenon. On the other hand, SYNTHESIS adds to CHARMY automation in assembling the designed and validated 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-centric approach to the dynamic and automatic composition of components. Section 3 recalls the approaches implemented in CHARMY and in SYNTHESIS. Section 4 provides an overview of the method our approach is based on by distinguishing three main phases of its utilization. Section 5 discusses in more detail the described approach and validates it by means of an explanatory example concerning with a cooling water pipe management system. Section 6 discusses related works. Section 7 concludes and discusses future work.

2. CONSIDERATIONS ON THE SA-CENTRIC APPROACH

SA emerged in the first nineties as a way of organizing and reasoning on software systems in a similar way of what has been done in other more mature engineering disciplines [18]. 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 software quality 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 [4, 20]. In fact, once the SA has been completely validated, 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 remainder of the paper we propose a method which model-checks the SA of a CB system with respect to desired requirements and assumes the SA as starting point for a code synthesis process. This process is performed to automatically derive a dynamic and correct component assembly. Although in the system life-cycle the 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.

In our approach the applicability of the synthesis phase leads on two basic requirements on the artifacts that we are considering. The first concerns the nature of the SA description. To enable the adaptor synthesis the SA should in fact contain more low level details with respects to the SAs traditionally used for other analysis purpose. Clearly this will have a negative impact on the initial validation phase, as said carried on via model checking, requiring more time and space to be completed. Nevertheless following the approach the validation phase will be carried on once and for all so we do not consider this a major hurdle to the applicability of the approach. The second hypothesis is more general and it is at the base of any synthesis approach, i.e., it is necessary that the component retrieved from the market encapsulates information on the accepted protocol in terms of an automata that can be managed by the chosen synthesis tool.

The added value of our approach, with respect to a classical software development process, is that system reconfigurations concerning with component replacing are applied in an automatic way and can happen at run-time with limited influences on the whole system configuration. The replacement of a component will only have in fact local consequences requiring the replacement of the adaptors directly interacting with the replaced component and still maintaining the validated properties. Moreover, the SA design of a new version of the system (due to a reconfiguration) is always up to date with respect to its corresponding implementation, being continuously integrated with the definitions of the new adaptor and of the the assembled components.

In this paper we do not discuss the consequences that replacements have on the state of a component and on the necessity of transferring the state from the replaced to the inserted component. This is clearly relevant for our approach but certainly out of scope. Thus, we assume that the middleware and component management infrastructure will support such an activity.

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 an explanatory example.

3.1 Charmy: a tool for SA designing and model-checking

CHARMY [8, 11] 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 [10] 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) [16] or in its Büchi Automata representation [6]. Instead of writing directly temporal properties, which is a task inherently error prone, CHARMY permits to describe them by using an extension of UML 2.0 Sequence Diagrams. These diagrams are called *Property Sequence Charts* (PSCs) [3, 19]. CHARMY automatically translates a PSC 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. SPIN 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., simple 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.

PSCs are an extension of UML 2.0 Sequence Diagrams 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 [22, 23] 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, whenever it is possible, 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 a 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 that performs specified[1] coordination policies. For our purposes, in the following, we will use SYNTHESIS only to prevent possible deadlocks in the assembly code and, hence, we do not take into account specified coordination policies.

4. METHOD DESCRIPTION

In this section we describe our method by giving a high-level overview of it. The explanatory example introduced in Section 5 will help us in order to better detail every single aspect of the approach. Our method can be described by distinguishing three main phases of its utilization. We, now, look at each phase.

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 diagrams and PSCs. State diagrams are used to describe how architectural components behave, PSCs 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 PSC notation and automatically translated into 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.

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 analysis 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, when it is possible. Note that, it might be the case that a component available on the market for our purposes does not exist. In this case the only choice that we have is to implement it by scratch and conforming to its specification in the validated SA. The third-party components

[1] In terms of Büchi Automata.

are selected by looking at the functionalities that they implement. They have to "contain" (possibly by interacting one another) the same functionalities implemented by the corresponding component in the system SA model. In general, this criterion is not enough. In fact, other criteria should be considered for a correct component selection, e.g., QoS constraints. However, it is worth mentioning that this work represents an ongoing work and, for now, we only focus to the component protocol (i.e., the sequences of messages exchanged with its expected environment) and we will consider more realistic selection criteria in possible future work. Although we make the assumption to be just interested on the component protocol, it might be the case that we are not able to find, for each component in the validated SA, directly corresponding ones on the market. In fact, the interaction protocol of one or more cooperating actual components might not fit the one of the component specified in the validated SA model. Moreover, the interaction of the actual components can lead to failures (e.g., deadlocks) since these components may have an incompatible interaction behavior. Thus, *static* (i.e., at compile-time) adaptation can be done to eliminate the resulting mismatches (e.g., to prevent component events that lead to deadlocks) and force the behavior of one or more interacting actual components to perform the interactions specified by the architectural component.

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 is automatically build. This is done by using SYNTHESIS. We assume to have a state diagrams specification of the actual components 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(s) in order to fit the specified one and avoid possible deadlocks. This is done by automatically synthesizing a suitable adaptor. This adaptor serves as deadlock-free assembly code for the actual component(s) and its environment (i.e., all the other components in the system). The adaptor mediates the interaction of the actual components connected to it in order to perform the interaction specified by the corresponding architectural component. This is done by using an *exception-handling*[2] mechanism that considers as exceptional events the ones that lead to deadlocks or that do not respect the specified protocol. In other words, the composition of the actual component(s) with the synthesized adaptor is deadlock-free and it behaves as the component specified in the system SA model. At this point, 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 used by CHARMY and 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 specified 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".

[2]It is only one of the possible solutions and requires to be investigated deeply cause of possible side effects that are concerned, in general, with the usage of exception-handling mechanisms.

4.3 Run-time phase: component composition through dynamic adaptation

Before speaking about dynamic adaptation we introduce some definitions in order to well define the kind of systems that we are able to manage. By considering the component and connector architectural elements, we will make use of the following definitions [7]:

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 system 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 components specification and the modeled components specification.

Once an actual component AC is replaced by a different one, e.g., AC', the current adaptor Adt, to which AC were connected, raises an exception due to, e.g., some unexpected event generated by AC'. By referring to Section 4.2, we recall that Adt raises an exception also whenever AC generates an event that would lead to a deadlock or that is not expected with respect to the protocol specified by the architectural component associated to AC. Thus, in order to correctly deal with unexpected events, Adt is able to distinguish between unexpected events generated by AC or by the component introduced in place of AC (i.e., AC'). In the case of an exceptional event has been generated caused by the replacing of the old component with the new one, Adt handles the raised exception by starting the dynamic adaptation phase. Firstly, the dynamic adaptation phase, at run-time, re-synthesizes off-line Adt producing a new version of it denoted by Adt'. This is done by taking into account the specification of the new actual component (i.e., AC'), of the possible actual components that have to cooperate with it and of the architectural component that was associated to AC. In the meanwhile, the dynamic adaptation phase also intercepts all requests performed towards and from AC'. They are temporarily ignored and "bufferized" in a queue that can be globally accessed by

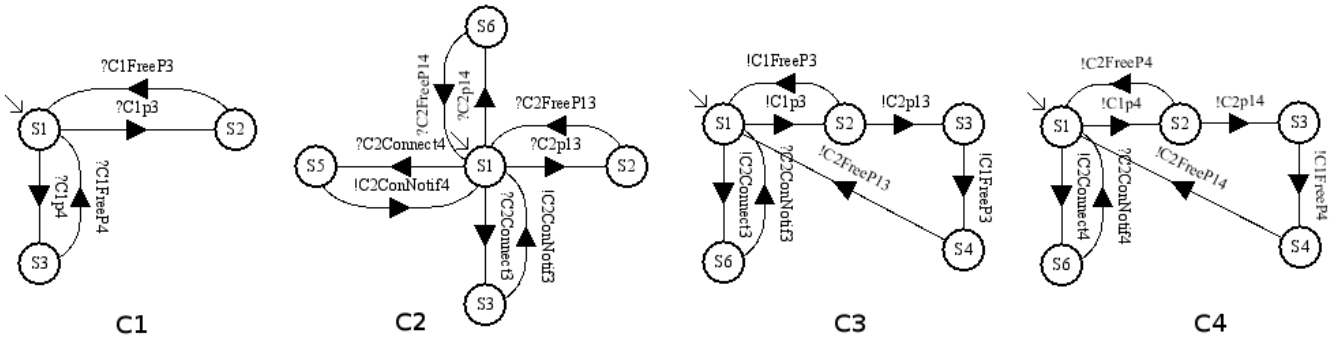

Figure 1: State machines of the water pipe management system

both any adaptor in the system and the process performing the dynamic adaptation phase. Secondly, the part of the system affected by the change is temporarily blocked and Adt' is deployed in the composed system. Finally, the execution of the composed system is triggered and it proceeds by first consuming the queued requests. Once the queue of the pending requests is empty, the system execution proceeds by following the normal flow. In this way, the initial version of the system is dynamically converted into a new one by still ensuring the validated properties. Note that this solution is not suitable for systems that specify "hard" timing constraints, e.g., real-time systems.

5. EXPLANATORY EXAMPLE: A COOLING WATER PIPE SYSTEM

The example that we use in this paper to better explain, detail 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* (resp. *p1*) is monitored by the server *C1* (resp. *C2*). *C1* (resp. *C2*) supports cooperative work and allows the access to a collection of data related to the water pipes that it monitors. *C1* (resp. *C2*) implements the interface *I1* (resp. *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* (resp. *I2*) provides the method *p* (resp. *p1*) to get an exclusive access to the data collection related to the water that flows in the pipe of the zone *p* (resp. *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* (resp. *I2*) provides the method *FreeP* (resp. *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 described in Section 4, 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.

Now, we discuss the application of our method to the above example by describing each phase of it.

5.1 Design-time phase

Figure 1 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.

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 step of verification is to check if the model is deadlock-free and if it satisfies the properties of interest.

About deadlock-freeness, nothing has to be shown because by using CHARMY we automatically verify that the parallel interaction of the component shown in Figure 1 is already deadlock-free.

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 2.B and 2.C in terms of its corresponding PSC [3, 19] 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 2.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

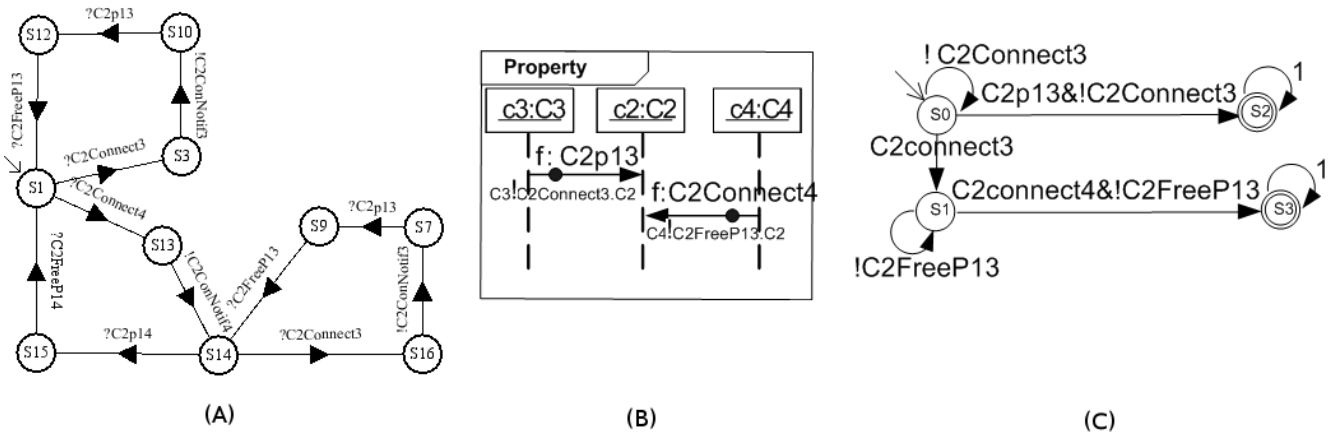

Figure 2: (A) *C2* modified to validate the desired property; (B) PSC **and (C) Büchi automaton of the desired property**

(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 [3, 19] 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.

Thus the interaction behavior of *C2* has to be changed. Figure 2.A reports the modifications made on the component *C2*. 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.

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*, *C3*, and *C4* shown in Figure 1, while *C2* is shown in Figure 2.A. The connectors that we consider are simple communication channels connecting each of *C3* and *C4* with both *C1* and *C2*.

5.2 Compile-time phase

Now, let us consider five COTS components that we have selected and acquired in order to assemble the validated system. Let us denote them as *AC1*, *AC2*, *AC3.1*, *AC3.2* and *AC4*. They "correspond" to the actual version (available on the market) of *C1*, *C2*, *C3* (for *AC3.1* and *AC3.2*) and *C4*, respectively. We recall that to perform the compile-time phase of our method we assume that the interface specification of the COTS components (e.g., the IDL specification) has been extended with extra-information related to the component interaction behavior with respect to the expected environment. Let us suppose that *AC1*, *AC2* and *AC4* behave exactly as specified for *C1*, *C2* and *C4*, respectively. Conversely, suppose that we did not find on the market a component that corresponds exactly to *C3*. The best thing that we did has been finding two components (i.e., *AC3.1* and *AC3.2*) whose interfaces[3] contain, in conjunction, the same interface of *C3*. The interaction protocol specification of these two components is shown in Figures 3.A (for *AC3.1*) and 3.B (for *AC3.2*).

The compile-time phase of our method uses SYNTHESIS to automatically synthesize an adaptor (i.e., *AdtC3*). It has to mediate all interactions between *AC3.1*, *AC3.2* and their environment in order to exhibit only the interactions specified by *C3*. The adaptor is built also to prevent (if possible) deadlocks that may raise from possible mismatching interactions. That is, *AdtC3* is synthesized in such a way that the parallel composition of it with *AC3.1* and *AC3.2* is deadlock-free and behaves as specified by *C3*. In other words, the deadlock-free implementation of *C3* is automatically derived and distributed in three actual components. Two of them are *AC3.1* and *AC3.2* acquired from third-parties. The other one is *AdtC3*, which is automatically built by taking into account the behavioral specification of *AC3.1*, *AC3.2* and *C3*. This aspect concerns the advantage that CHARMY takes from being combined with SYNTHESIS. That is, the component integration code (if possible) is automatically derived correct-by-construction without requiring the developers to work on it.

In Figure 4 we show the state machine (also called automaton) automatically synthesized for *AdtC3*.

For the sake of brevity, here, we do not show the synthesis process in detail. Refer to [23] (and references therein) for a detailed description of the deadlock-free adaptor synthesis.

Informally, the adaptor's automaton is automatically built by considering the following criteria: (i) an adaptor has a strictly sequential input-output behavior. That is, it has to receive a request (resp. notification) and delegates it towards the component expecting to

[3]Here, a component interface is seen as a list of provided/requied methods.

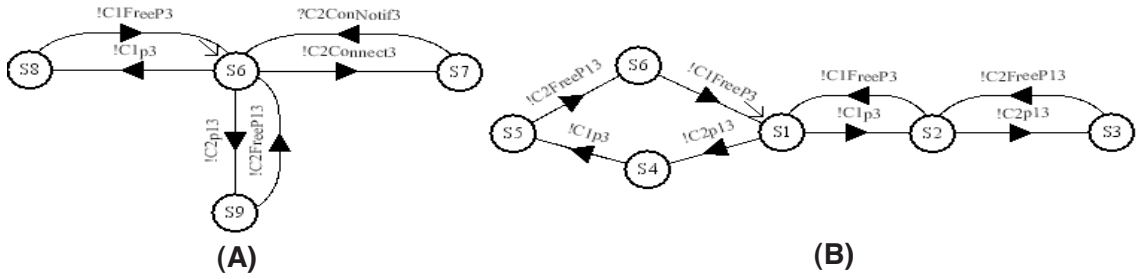

Figure 3: State machines of (A) *AC3.1* and (B) *AC3.2*

receive such a request (resp. notification). For instance, the adaptor shown in Figure 4 can, from its initial state (i.e., *S6*), receive a request of *p3* from *AC3.2* towards *C1* (i.e., the message *?C1p3[2]* from *S6*). After receiving this request, the adaptor delegates it to *C1* (i.e., the message *!C1p3* from *S11*); (ii) at a first stage, the adaptor has to model all possible component interactions, i.e., it is analogous to the *product automaton* of the automata modeling the interaction behavior of the components assembled by the adaptor. As mentioned before, this product automaton must take into account the input-output interaction model of the adaptor. For instance, let us denote by *EnvC3* the automaton equal to *C3* where input actions have been converted into output ones, and viceversa. The adaptor shown in Figure 4, is built by considering the product automaton of *AC3.1*, *AC3.2* and *EnvC3*, and by imposing on it an input-output structure (i.e., one transition on the product automaton becomes two transitions on the adaptor's automaton, that is receive and send transitions). In other words, *EnvC3* models the environment expected by *C3* in order to not block. This aspect concerns the advantage that SYNTHESIS takes from being combined with CHARMY. That is, SYNTHESIS has to solve a problem that has been reduced in space since, for instance, it is enough to consider *EnvC3* as environment for *AC3.1* and *AC3.2* rather than considering the parallel composition of *C1*, *C2* and *C4*. This is true because *C3* has been already validated (by CHARMY) with respect to the interaction with *C1*, *C2* and *C4*. Note that, for large systems, considering as environment a single component rather than the parallel composition of many components represents, in general, a significant optimization. Thus, within our context, the approach implemented by SYNTHESIS (that, in general, suffers the state-explosion phenomenon) is more feasible in practice.

By referring to Figure 4, by using SYNTHESIS, we automatically obtain a first version of the adaptor's model. At this stage the adaptor simply routes component messages and each input it receives is strictly followed by its corresponding output. If there are deadlocks, the adaptor model allows SYNTHESIS to detect them. Each "deadlocking" state is denoted by a dark-gray filled node. A deadlocking state is a sink node or a state leading only to deadlocking states.

In our example, a deadlock occurs in two cases. One case is when *AC3.2* requires the access to, first, the zone *p* and, then, to *p1*. In fact, in this situation *AC3.2* would release the access granted to *p1* but its environment expects to release the access granted to *p*, first. This scenario is modeled by the path from *S6* to *S29* of the adaptor's automaton (see Figure 4). The second case is when *AC3.1* requires the access to the zone *p* and, then, *AC3.2* accesses to *p1* (i.e., the path from *S6* to *S24*). At this point, the only possible message that can be exchanged is the releasing of the access granted to *p* from *AC3.1* (i.e., the path from *S24* to *S26*). After performing this message, all components in the system are blocked waiting for a message that will never be performed. In fact, the environment of *AC3.1* and *AC3.2* expects that the access granted to *p1* will be released but both *AC3.1* and *AC3.2* are unable to do this.

To prevent these deadlocks, SYNTHESIS automatically prunes all the deadlocking paths of the adaptor's automaton (i.e., the path from *S13* to *S26* and the one from *S12* to *S29*).

In this way, by using SYNTHESIS, the model of the deadlock-free adaptor is automatically obtained.

Before deriving the actual code of the deadlock-free adaptor, there is a last check that must be performed. That is, the parallel composition of *AC3.1*, *AC3.2* and the deadlock-free adaptor *AdtC3* has to be "equivalent" to *C3*. This check is required because, after that all the deadlocks have been removed, *AdtC3* might exhibits behaviors that cannot be performed by *C3* and vice versa. Let us denote that parallel composition by S.

Our notion of behavioral equivalence is called *CB-Simulation* and it is based on *stuttering equivalence* [17]. We are based on stuttering equivalence because S, beyond the transitions labeled with *C3*'s actions, has also "*tau*" transitions. As usual, they come from the synchronization with *AC3.1* and *AC3.2* .

For the sake of brevity we refer to [12] for a formal definition of CB-Simulation. Here it is enough to say that, as usual in the automata theory, the aim of the equivalence check is to verify that S simulates *C3* and vice versa, under a suitable notion of simulation (i.e., CB-Simulation). That is, SYNTHESIS automatically checks that S performs[4] all the behaviors of *C3* and vice versa.

By referring to the deadlock-free version (i.e., the one without finite paths) of the adaptor's automaton shown in Figure 4, S has the same automaton where all transitions labeled with actions performed by *AC3.1* and *AC3.2* (i.e., the ones terminating with "[1]" and "[2]") are *tau* transitions. S and *C3* are not equivalent (under our notion of behavioral equivalence defined through CB-Simulation). In fact, if *AC3.1* accesses to *p* before that the composed system has performed any other interaction (i.e., the path from *S6* to *S13*) only one behavior of *C3* can be performed (i.e., the path from *S13* to *S6*). This behavior corresponds to releasing the access granted to *p*. The execution of all other behaviors (i.e., accessing to *p1*) is disallowed. That is, there is a possible execution of *AC3.1* that cannot guarantee the execution of all behaviors specified by *C3*. To prevent this problem, SYNTHESIS automatically

[4]By taking into account a stuttering interpretation.

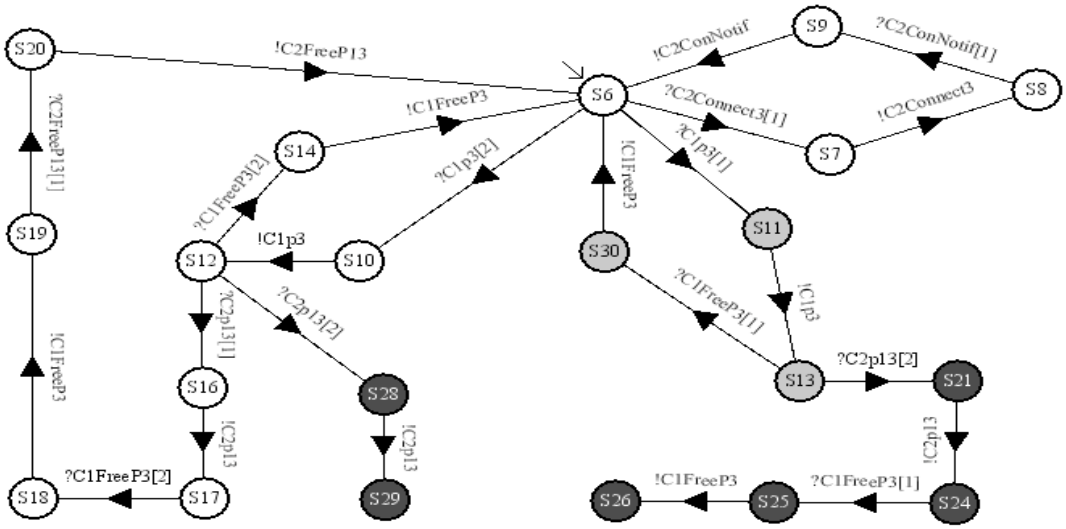

Figure 4: Behavioral model of *AdtC3*

prunes the cycle from *S6* to *S13* and, again, to *S6*. The states of this cycle are denoted by light-gray filled nodes in Figure 4.

After this check, by using SYNTHESIS, we automatically obtain the model (a state machine) of the adaptor that is deadlock-free and allows S to exhibits a behavior equivalent to *C3*.

Using the same technique described in [23], from this state machine, 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 the combined execution of *AC3.1* and *AC3.2* would "contain" only a sub-set of the set of interactions specified by *C3* (plus different ones), then performing adaptation by only restricting its behavior (as we have done previously) is not enough. In fact, in this case, the set of interactions modeled by *AdtC3* must be also extended in order to perform the *C3* interactions that are missing in the specification of *AC3.1* and *AC3.2*. This case occurs when, e.g., the interface signature (i.e., method and parameter names) of *AC3.1* and *AC3.2* does not match with the one of *C3*.

In [23] such a *protocol-enhancing* technique is described and since it is out of the scope of this paper we refer to [23] for a detailed description of it.

Informally, it is based on the original idea, in the area of protocol specification, of Yellin and Strom [25]. Analogously to their work, to semi-automatically enhance the protocol of the adaptor, we require a specification of a *protocol-enhancing policy*, e.g., a set of correspondences of method and parameter names belonging to different component interfaces. Each enhancing policy is used to derive a wrapper component that must be interposed between the adaptor and the components affected by the enhancement. In this way the behavior of the adaptor is enhanced in order to deal with incompatibilities that cannot be solved by simply restricting the composed system behavior.

A similar approach has been adopted also by Bracciali, Brogi and Canal in the area of component adaptor specification [5].

5.3 Run-time phase

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 components that it controls.

A possible solution for this is the use of exception handling techniques and in particular *Architectural exceptions* [13, 21] 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.

This is the starting point of the run-time phase of our method. Let us consider a scenario in which *AC3.1* is replaced (at run-time) by a different component (i.e., *AC3.3*). *AdtC3* catches the event associated to the replacement of *AC3.1* and triggers the event associated to the possible off-line synthesis of a new version of it (i.e., *AdtC3.3*). If *AC3.3* "contains" the interaction specified by *AC3.1* (plus different ones) then it is not required to re-synthesize *AdtC3* (into *AdtC3.3*) because, by construction, it is already able to restrict the behavior of *AC3.3* to perform only the *AC3.1* interactions. Thus, in this case, the system can evolve at run-time without performing any further adaptation. Otherwise, *AdtC3.3* is synthesized off-line by also using (if it is required) the previously mentioned

protocol-enhancing technique. During the off-line synthesis (and before that *AdtC3* is substituted by *AdtC3.3*) *AdtC3* ignores the messages from and towards *AC3.3* since they are stored in a queue of pending requests. When *AdtC3.3* is deployed in place of *AdtC3*, it is forced to first handle the pending requests in the queue and then the composed system execution proceeds by following the normal flow. 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. [26] 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 (that requires to be adapted) 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 components (or adaptors) in the system.

Kulkarni et al. [15] 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. [14] 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. [2] 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 pre-conditions and post-conditions 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.

As mentioned in Section 5, our research is also related to work in the area of protocol adaptor synthesis developed by Yellin and Strom [25]. 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 by means of adaptors. However, they are limited to only consider syntactic incompatibilities between the interfaces of components and they do not allow the kind of interaction behavior that our synthesis approach supports. Moreover, they require a formal specification of the adaptor dictating, for example, a mapping function among events of different components. Although requiring this kind of specification enhances applicability of their approach respect to the one implemented by SYNTHESIS, it is in contrast with our need to be as automatic as possible. In fact even if other kinds of techniques to specify the adaptor are possible, providing the adaptor specification requires to know too many implementation details thus missing part of the goals of the work presented in this paper. However, if we assume to have as input that detailed adaptor specification, SYNTHESIS can be used to deal with the kind of incompatibilities that Yellin and Strom face in their work. In [23], we extended the synthesis process implemented by SYNTHESIS in order to not only restrict the coordinator behavior but also augmenting it in order to consider also such incompatibilities, e.g., interface signature mismatches.

In other work from Bracciali, Brogi and Canal [5], in the area of component adaptation, it is shown how to automatically generate a concrete adaptor from: (i) a specification of component interfaces, (ii) a partial specification of the components interaction behavior, (iii) a specification of the adaptation in terms of a set of correspondences between actions of different components and (iv) a partial specification of the adaptor. The key result is the setting of a formal foundation for the adaptation of heterogeneous components that may present mismatching interaction behavior. Analogously to the work of Yellin and Strom, although this work provides a fully formal definition of the notion of component adaptor, its application domain is different from our. Since, in specifying a system, we want to maintain a high abstraction level, assuming a specification of the adaptation in terms of a set of correspondences between methods (and their parameters) of two components requires to know many implementation details (about the adaptation) that we do not want to consider in order to synthesize the adaptor.

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 architectural analysis and code synthesis is performed by combining two previously developed approaches from some of the authors. One approach is implemented by CHARMY that is for performing architectural analysis of an SA model. The other one is implemented by SYNTHESIS that is for performing code synthesis.

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.

Future work concerns 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. Moreover, to make our method a systematic engineering approach, processes that helps the developer in performing a realistic COTS components selection phase must be investigated (see [1], [9] and references therein). Another interesting aspect concerning future work is the possibility to include in the models not only functional aspects but also extra-functional ones such as timing information. This would extend the applicability of our approach to systems where taking into account the elapsing of time of a component request is a critical task such as embedded real-time systems. Furthermore, a better investigation of the current technologies and mechanisms needed to implement our *self-healing* adaptors must be conducted. Finally, a validation of our approach on a single example is not enough. Moreover the case study described in this paper is not a large-scale one and, hence, we have not been able to conduct a systematic estimation of our tolerance with respect to the state explosion phenomenon. We just experimented that we reduce it with respect to performing directly code synthesis. Thus, future work concerns also the validation of the approach on other case-studies dealing with large-scale systems.

8. ACKNOWLEDGEMENTS

This work is partially supported by the PLASTIC project: Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication. Sixth Framework Programme. http://www.ist-plastic.org.

9. REFERENCES

[1] C. Alves and J. Castro. Cre: A systematic method for cots components selection. In *SBES'01*, 2001.

[2] 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.

[3] M. Autili, P. Inverardi, and P. Pelliccione. A graphical scenario-based notation for automatically specifying temporal properties. In *SCESM'06*, Shanghai, China, May 27, 2006.

[4] 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.

[5] A. Bracciali, A. Brogi, and C. Canal. Systematic component adaptation. *ENTCS*, 66(4), 2002.

[6] 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 1–11. Standford University Press, 1960.

[7] 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.

[8] Charmy Project. Charmy web site. http://www.di.univaq.it/charmy, February 2004.

[9] L. Chung and K. Cooper. Matching, ranking, and selecting components: A cots-aware requirements engineering and software architecting approach. In *MPEC'04*, 2004.

[10] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.

[11] P. Inverardi, H. Muccini, and P. Pelliccione. Charmy: an extensible tool for architectural analysis. In *ESEC/FSE-13*, pages 111–114, New York, NY, USA, 2005. ACM Press.

[12] P. Inverardi and M. Tivoli. *Software Architecture for Correct Components Assembly*. Springer, LNCS 2804, Sept. 2003.

[13] V. Issarny and J. P. Banatre. Architecture-based exception handling. In *HICSS*, 2001.

[14] K. H. 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.

[15] S. S. Kulkarni and K. N. Biyani. Correctness of component-based adaptation. *in CBSE7*, May 2004.

[16] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.

[17] R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

[18] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[19] PSC home page: http://www.di.univaq.it/psc2ba, 2005.

[20] 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*, volume 24, July 1999.

[21] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. C. Filho. Exception handling in the development of dependable component-based systems. *Softw. Pract. Exper.*, 35(3), 2005.

[22] Synthesis Project. Synthesis web site. http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.php, September 2004.

[23] M. Tivoli and M. Autili. Synthesis: a tool for synthesizing correct and protocol-enhanced adaptors. *L'Object journal*, 12(1), 2005.

[24] 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.

[25] D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2), March 1997.

[26] 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.