The Journal of Systems and Software 81 (2008) 2237–2251

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

An architectural approach to the correct and automatic assembly of evolving component-based systems

Patrizio Pelliccione [a], Massimo Tivoli [a,*], Antonio Bucchiarone [b], Andrea Polini [c]

[a] Dipartimento di Informatica, Università dell'Aquila, Via Vetoio snc, Coppito, 67100 L'Aquila, Italy
[b] IMT Lucca Institute for Advanced Studies, Piazza S. Ponziano 6, 55100 Lucca, Italy
[c] Dipartimento di Matematica e Informatica, Università di Camerino, 62032 Camerino, Italy

A R T I C L E I N F O

Article history:
Received 4 December 2006
Received in revised form 26 May 2008
Accepted 27 May 2008
Available online 4 June 2008

Keywords:
Software architecture
Verification and validation
Component based software engineering
Component assembly
Component adaptation

A B S T R A C T

Software components are specified, designed and implemented with the intention to be reused, and they are assembled in various contexts in order to produce a multitude of software systems. However, in the practice of software development, this ideal scenario is often unrealistic. This is mainly due to the lack of an automatic and efficient support to predict properties of the assembly code by only assuming a limited knowledge of the properties of single components. Moreover, to make effective the component-based vision, the assembly code should evolve when things change, *i.e.*, the properties guaranteed by the assembly, before a change occurs, must hold also after the change. Glue code synthesis approaches technically permit one to construct an assembly of components that guarantees specific properties but, practically, they may suffer from the state-space explosion phenomenon.

In this paper, we propose a Software Architecture (SA) based approach in which the usage of the system SA and of SA verification techniques allows the system assembler to design architectural components whose interaction is verified with respect to the specified properties. By exploiting this validation, the system assembler can perform code synthesis by only focusing on each single architectural component, hence refining it as an assembly of actual components which respect the architectural component observable behaviour. In this way code synthesis is performed locally on each architectural component, instead of globally on the whole system interactions, hence reducing the state-space explosion phenomenon.

The approach can be equally well applied to efficiently manage the whole reconfiguration of the system when one or more components need to be updated, still maintaining the required properties. The specified and verified system SA is used as starting point for the derivation of glue adaptors that are required to apply changes in the composed system. The approach is firstly illustrated over an explanatory example and is then applied and validated over a real-world industrial case study.

1. Introduction

Software components composition has been advocated as a software development approach allowing software engineers to manage the continuously raising software complexity, the increasing software dependability requirements, and the reduced time-to-market. In the last decade, researchers and practitioners have put a big effort on trying to make real the software "componentization" vision (see Schmidt et al., 2007 and its previous editions since 2004). In the same period a new discipline, called "Component Based Software Engineering" (CBSE), has been introduced Szyperski, 1998; Heineman and Councill, 2001). A Component-Based (CB) software system is an *assembly* of software components (usually implemented by means of either third-party libraries or in-house components), designed to meet the system requirements that were identified during the analysis phase (Crnkovic, 2002).

The definition of software as a composition of software elements could certainly take advantage from languages and techniques identified within the "Software Architecture" (SA) research domain (Shaw and Garlan, 1996). In particular, the notion of SA assumes a key role in component-based software development since it represents the reference skeleton used to compose components and to let them interact. It is worth noticing that within the SA discipline, the interactions among components are first class citizens and are specifically represented by the notion of software "connector" (Shaw and Garlan, 1996). 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, components and connectors can be composed together to make up different

* Corresponding author.
E-mail addresses: pellicci@di.univaq.it (P. Pelliccione), tivoli@di.univaq.it (M. Tivoli), a.bucchiarone@imtlucca.it (A. Bucchiarone), andrea.polini@unicam.it (A. Polini).

system configurations. As a result, a component based software system can be designed specifying its architecture in terms of components, connectors and its (architectural) configuration.

Indeed, different architectural configurations guarantee different behavioural properties. Therefore, further elements that need to be taken into account are the properties that an SA configuration has to satisfy. Two complementary approaches can be used to guarantee that a software system satisfies a set of properties:

(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 techniques (Clarke et al., 2001).
(2) *Code synthesis:* a code synthesis technique can be defined in order to generate the "correct" assembly code for the (pre-selected and pre-acquired) components forming the specified system. This code is derived in order to force the composed system to exhibit only the specified interactions. In our context, the synthesis techniques we refer to belong to the domains of *discrete controller synthesis* (Brandin and Wonham, 1994; Ramadge and Wonham, 1987), *converter synthesis* (Passerone et al., 2002), and *protocol adaptor synthesis* (Yellin and Strom, 1997; Bracciali et al., 2005; Inverardi and Tivoli, 2003; Bracciali et al., 2002).

Both approaches have advantages and drawbacks.

On the one hand the architectural analysis permits to formally prove that an SA satisfies given properties, under the assumption that the running version of the system will completely conform to its SA. Unfortunately, this is not always the case. As a result the verification step might loose much of its power, being conducted on an SA that might refer to a system that is different from the "implemented" one, *e.g.*, the SA might refer to only an abstraction of the actual system. On the other hand, performing automatic verification of an SA that is too close to its implementation may lead to the state-space explosion phenomenon hence making the analysis useless.

On the contrary, code synthesis approaches technically permit to construct an assembly of components satisfying specific properties. This approach may nevertheless automatically produce a non-tractable model of the assembly. This is due to the, in general, unavoidable need of taking into account all possible interactions of the components in the assembly. That is, to guarantee some specific properties at the level of the whole assembly, it is not always possible to operate locally on each single component. This is particularly true to guarantee specific interaction and safety properties, *e.g.*, deadlock-freedom. This kind of problem generally leads to the well-known state-space explosion phenomenon, in which the dimension of the model and its number of states prevent the applicability of any analysis technique. This problem becomes particularly relevant when run-time re-factoring is required, asking, in general, for a complete revision of the whole system when components need to be added, substituted, or updated.

Hence, if we want to *automatically* assemble a set of possibly *third-party* components to form a system in such a way that it guarantees specific properties of interaction, neither architectural analysis nor code synthesis (taken in isolation) are ideal due to the above discussed limitations. As a consequence, in this work, we propose an SA-based approach that combines architectural analysis and code synthesis in order to *efficiently* and *correctly* assemble a system out of a set of already implemented components. The assembled system must be synthesized in such a way that it can evolve, at run-time, to possible changes (*e.g.*, component replacement).

Firstly, the system's SA is verified and refined with respect to a set of properties of interest (*i.e.*, standard analysis). Successively, an initial version of the composed system is built by taking into account both the models of the *architectural* components in the SA and the ones of the *actual* components. Actual components are selected and acquired on the market in order to implement a verified architectural component. The actual components' assembly code is automatically synthesized (*i.e.*, code synthesis) in order to be *correct-by-construction*, *i.e.*, it adapts the actual components and composes them in order to produce an assembly that behaves equivalently to an architectural component. The assembly code, supporting the implementation of an architectural component, is derived as an adaptor acting as coordinator of the actual components' interaction behaviour.

The approach brings many advantages with respect to a more "traditional" component-based software development process. First, the approach we propose forces a correspondence among the architectural specification and the real implementation, with obvious positive consequences in particular with reference to possible re-factoring phases. The second important characteristic concerns the fact that the synthesis step (acting locally with respect to each single architectural component) deals with "smaller" and, hence, more tractable software models, therefore reducing the effects of the state-space explosion phenomenon. Finally, the system is assembled in order to be modified at run-time, updating or substituting a component with reduced effects on the overall system, and still guaranteeing the properties that were verified during the analysis phase.

Our approach builds on two existing approaches that have been developed by some of the authors. One has been implemented in the CHARMY tool (Charmy Project, 2004; Inverardi et al., 2005) (architectural analysis) and the other in the SYNTHESIS tool (Synthesis Project, 2004; Tivoli and Autili, 2006, 2007) (code synthesis). These two approaches take advantage from each other. On the one hand, CHARMY provides SYNTHESIS with an already verified system's SA. SYNTHESIS can exploit this system's SA to perform adaptation locally on each architectural component rather than at the level of the global system interactions, thus reducing the state-space explosion phenomenon. On the other hand, SYNTHESIS adds to CHARMY automation in assembling and implementing the designed and verified system as third-party components. Indeed, in CHARMY this task is completely delegated to the developer. Our approach shares some problems that are typical to the area of *architectural mismatch detection and recovery*. We will address the relation between the work in architectural mismatch and our work in Section 5.2.

The applicability of the synthesis leads us to make one basic assumption on the artefacts that we are considering. That is, it is necessary that the interface definition of an actual component encapsulates information on the interaction protocol assumed by the component when it interacts with the expected environment (*i.e.*, it is a so-called "behavioural interface"). This information is given in the form of a state machine. According to "*design by contract*" approaches (Szyperski, 1998), we can assume that the IDL file of a component is augmented, by the component developer, through a commented header. Such a header encodes somehow (*e.g.*, by using XML) a state machine that models the observable behaviour performed by the component when it interacts with its expected environment. Note also that, in our context, a component always respects its interaction protocol specification since it is provided by the developer of the same component, who is aware of the information needed to specify the component protocol.

The paper is organized as follows. Section 2 introduces some background notions needed for the understanding of the approach presented in this paper. It also recalls the approaches implemented in CHARMY and in SYNTHESIS. Section 3 describes the method that our approach is based on, by distinguishing four main phases of utilization. The approach is described by means of a running example. Section 4 applies the described approach to a real-scale industrial case study concerning a spacecraft system. Due to the size of the case study, Section 4 just describes the considered spacecraft system and reports the results of the application of our method to that system. Section 5 discusses related work. In Section 6, we discuss the kind of systems that we are able to manage and we summarize the advantages obtained, with our approach, by combining architectural analysis and code synthesis. These advantages have been experienced while applying our approach to the considered case study. Section 7 concludes and discusses future work.

2. Background

In this section we introduce some background notions needed for the understanding of the approach presented in this paper. In particular, in Sections 2.1 and 2.2, we briefly recall those aspects of CHARMY and SYNTHESIS that are relevant for the approach presented in this paper.

2.1. CHARMY: A tool for SA designing and model-checking

CHARMY (Charmy Project, 2004; Inverardi et al., 2005) is a project whose goal is to apply model-checking techniques to validate the SA's conformance to certain properties. In CHARMY the SA is specified through state machines 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 (Holzmann, 2003) model checker) an SA model that can be executed and verified by SPIN. This model can be verified with respect to a set of properties, *e.g.*, deadlock-freedom, correctness of functional properties, starvation, etc., expressed in Linear-time Temporal Logic (LTL) (Manna and Pnueli, 1992) or in its Büchi Automata representation (Buchi, 1960). Instead of writing directly temporal properties, which is an inherently error prone task, CHARMY permits to describe them by using an extension of a subset of UML 2.0 Sequence Diagrams. These diagrams are called *Property Sequence Charts* (PSCS) (Autili et al., 2007). CHARMY automatically translates a PSC into a Büchi automaton. The model checker SPIN is a widely distributed software package that supports the formal verification of concurrent systems. 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 machines: labels on arcs uniquely identify the messages exchanged and each message 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, or an internal operation (τ) (*i.e.*, an event that does not require synchronization between state machines). Sent messages of a component are prefixed by an exclamation mark (*i.e.*, "!"), while received messages are prefixed by a question mark (*i.e.*, "?"). An event can have several parameters as defined in the parameter list. op_1,op_2, $\cdots$,op_n are the operations that are executed when the transition is performed. By exploiting SPIN, the parallel interaction of the components in the designed SA is modelled, in CHARMY, by using the state machine *parallel composition operator* (Keller, 1976). The parallel composition operator combines the behaviours of two state machines by synchronizing their shared/common actions (*i.e.*, sent and received messages with the same label) and interleaving their non-shared and internal actions.

PSCS are an extension of a subset of the UML 2.0 Sequence Diagrams stereotyped such that: (i) each rectangular box represents an architectural component, (ii) each arrow defines a message exchanged among two components. Between a pair of messages we can select whether other messages can (loose relation) or cannot (strict relation) occur. Message constraints are introduced to define a set of messages that may never occur in between the message containing the constraint and its predecessor or successor. The predecessor of the first message of a PSC is the start-up of the system. Messages are typed as *regular messages* (optional messages), *required messages* (mandatory messages), or *fail messages* (messages representing a fault).

2.2. Synthesis: A tool for synthesizing failure-free component adaptors

SYNTHESIS (Synthesis Project, 2004; Tivoli and Autili, 2006; Autili et al., 2007) 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 the level of the component interaction protocol. Its aim is to analyze and prevent interaction mismatches (*i.e.*, deadlocks, livelocks, etc.) that can arise from components 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 third-party (or COTS) components, the code that implements a new component that has then to be inserted into the composed system. This new component implements a software coordinator (also called adaptor). The coordinator mediates the interaction among the components in order to prevent possible integration failures.

SYNTHESIS assumes that a specification of the observable behaviour of each actual component (forming the system to be assembled) is available in the form of a state machine. With observable behaviour of the component, we mean the behaviour of the component in terms of the messages exchanged with its expected environment (*i.e.*, the interaction protocol assumed by the component). Under this assumption, SYNTHESIS is able to automatically derive the correct 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 coordination policies. A coordination policy is a functional property of interaction given in the form of a state machine.

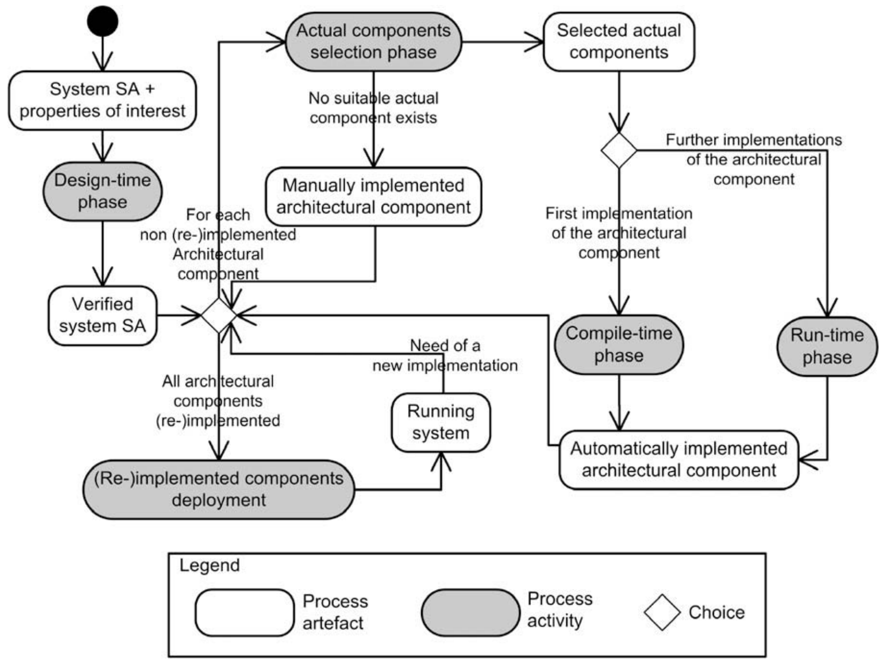
3. Method description

Our method is composed of four main phases organized as shown in Fig. 1. In the following the description of the method assumes that an SA has been modelled by using CHARMY (*System SA + properties of interest* in Fig. 1).

Design-time phase: The first phase concerns the system SA verification. This phase is performed by using CHARMY and it is described in Section 3.1. The input of this phase is an SA and the properties that one wants to check. The output is a system SA specification that respects the properties of interest (*Verified system SA* in Fig. 1).

For each verified architectural component that has not yet been implemented, the *Actual components selection phase* is performed. After that all the architectural components have been implemented, they are deployed (*Re-implemented components deployment* in Fig. 1) hence producing a first running version of the system (*Running system* in Fig. 1).

Actual components selection phase: As already mentioned in Section 1, our method implements each architectural component as an assembly of actual components acquired from a third-party, when possible. This phase aims at selecting third-party components by looking at their interfaces and functionalities. This phase and the selection criteria used to establish which actual components have



System SA + properties of interest
Design-time phase
Verified system SA
For each non (re-)implemented Architectural component
Actual components selection phase
No suitable actual component exists
Manually implemented architectural component
Selected actual components
Further implementations of the architectural component
First implementation of the architectural component
Compile-time phase
Run-time phase
Automatically implemented architectural component
Need of a new implementation
Running system
All architectural components (re-)implemented
(Re-)implemented components deployment
Legend
Process artefact
Process activity
Choice

Fig. 1. Our method.

to be acquired to implement an architectural component are described in Section 3.2. This phase takes as input a verified architectural component and it is performed with respect to a repository of actual components acquired from a third-party (Oreizy et al., 1998) (black-box components). The output is the set of actual components selected as possible candidates for the implementation of the architectural one or an empty set. In case of an empty set, the architectural component is manually implemented (*Manually implemented architectural component* in Fig. 1) since we did not find suitable components that can be assembled to implement the considered architectural component. In this case it is up to the developer to guarantee that the component implementation conforms to its architectural specification, *e.g.*, via verification techniques.

If possible candidates are found (*Selected actual components* in Fig. 1) they could still need some adaptations (*e.g.*, they might provide more functionalities as needed or interaction mismatches might occur). The compile-time phase and the run-time phase will automatically manage that in the first implementation of the architectural component and in its further implementations, respectively.

Compile-time phase: In order to correctly implement the considered architectural component, this phase, described in Section 3.3, automatically produces an assembly of the selected actual components that is correct with respect to the architectural component's observable behaviour (*Automatically implemented architectural component* in Fig. 1).

Run-time phase: When a new implementation of an architectural component is needed (the transition *Need of a new implementation* outgoing from *Running system*), the correct (re-)implementation of the considered architectural component is produced analogously to what is done in the compile-time phase. The run-time phase, which is described in Section 3.4, performs additional operations with respect to the compile-time phase. These operations are the stop of the running system in a consistent state and the transfer of the computational state.

In the following subsections we will detail each single phase that composes our method by making use of a simple explanatory and running example.

This example is concerned with the automatic assembly of a component-based system made of two components `C1` and `C2`.

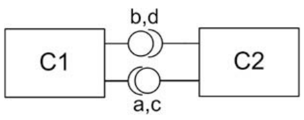


Fig. 2. SA of the running example.

When assembling these components we want to automatically ensure deadlock-freedom and other specified behavioural properties. Moreover, we wish to obtain a system which can tolerate changes at run-time. Fig. 2 shows the SA of this system. The components communicate by means of communication channels that are implicitly defined through provided/required interface matching. As shown in Fig. 2, the provided (*resp.*, required) interface of `C1` matches with the required (*resp.*, provided) interface of `C2`. The components communicate by exchanging messages `a`, `b`, `c`, and `d`. The state machines describing the behaviour of each component and thus the explanation of the communication among these components will be described in Section 3.1.

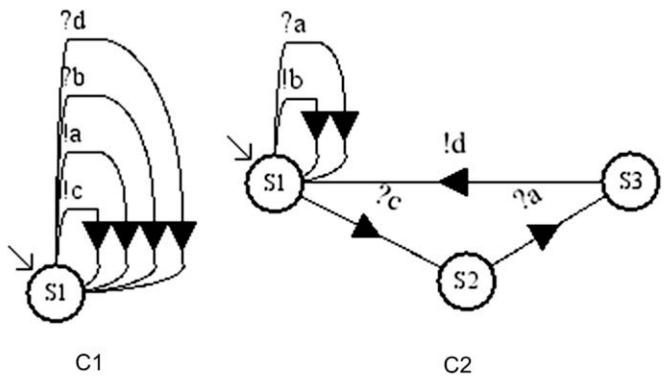


Fig. 3. State machines of the running example.

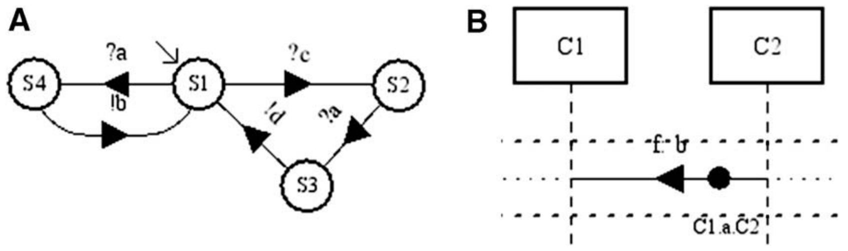


Fig. 4. (A) `C2` modified to validate the desired property, and (B) PSC of the desired property.

3.1. Design-time phase

We recall that, at design-time, our approach assumes that the SA specification of the system to be assembled is provided in terms of state machines and PSCs. Fig. 3 shows the state machines that describe the *desired* behaviour of the components `C1` and `C2` shown in Fig. 2.

These state machines are designed by using CHARMY that generates the Promela code needed for the verification with SPIN. At the beginning it is not sure that the desired behaviour specified by the designer is correct (especially for large systems). Therefore, the intention of the designer is to verify the correctness of the model in order to refine it and produce the correct specification of the system that must be assembled.

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

Regarding deadlock-freedom, nothing has to be shown because by using CHARMY we automatically verify that the parallel interaction of the components shown in Fig. 3 is already deadlock-free.

For the sake of simplicity, we consider only one desired property. This property represents a desired interaction protocol for `C1` and `C2`. That is, a failure occurs if message `b` is exchanged (between `C2` and `C1`) before `a` has been exchanged.

The property is described in Fig. 4B in the form of a PSC diagram. By referring to Section 2.1, in this example we use one fail message (the message prefixed by the label "`f:`"). We recall that fail messages are used to identify messages that should never be exchanged. We use also one constraint to impose a "restriction" on the set of messages that can be exchanged before the considered message and after its predecessor. Coming back to the desired property, referring to the PSC notation, `b` is the considered message. It has `a` as its constraint. The constraint implements the restriction imposed on the interaction among `C1` and `C2` in order to let the exchange of `b` happen only after the exchange of `a`.[1] CHARMY and its engine SPIN return a "`not valid`" result for this property, essentially caused by either `C1` or `C2` that, with respect to the specified desired interaction, are too simple and without logic, *i.e.*, no order is imposed on the exchange of `a` and `b` messages.

Therefore, the interaction behaviour of, *e.g.*, `C2` has to be changed. Fig. 4A reports the modifications we made on `C2`. Now, `C2` explicitly contains an order for the `a` and `b` messages exchanged with `C1`.

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 architectural components `C1` (shown in Fig. 3) and `C2` (shown in Fig. 4A). The connectors that we consider are simple communication channels connecting `C1` with `C2`.

3.2. Actual components selection phase

Within our approach, actual components are selected and acquired on the market in order to implement a verified architectural component. 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 conform to its specification in the verified SA. For our purposes, the third-party components are selected by looking at their interfaces and functionalities. That is, the component selection criterion that we consider is that the actual components have to "contain" (possibly by putting their execution in parallel) the same functionalities implemented by the corresponding architectural component.

Furthermore, a mapping must exist between the different component interfaces. This mapping can be empty, meaning that it is not required since the message names of the architectural component interface match with the ones of the actual components' interface (*i.e.*, a syntactical and semantic one-to-one correspondence exists). It can be a trivial mapping, meaning that there exists a one-to-one correspondence between messages of the architectural component interface and the ones of the actual components' interface except for their names (*i.e.*, a semantic one-to-one correspondence exists). In this case, the needed message relabeling is realized by means of trivial component wrappers that directly delegate messages without any particular message routing logic. In the most general case that mapping can also be complex, meaning that there exists a many-to-many correspondence between messages of different components. In that case, the component wrappers implementing such a mapping also have a complex message routing logic. In general, this interface mapping cannot be built automatically and it requires to develop (by hand) component wrappers solving, *e.g.*, syntactical mismatches, as done in previous work by some of the authors (Autili et al., 2004; Tivoli and Autili, 2006). Since in this work we focus on automatically preventing interaction protocol mismatches, we consider this problem out of the scope of this paper and, hereafter, we will assume that component messages syntactically and semantically match since either they already match or suitable component wrappers have been previously developed by the system assembler (*i.e.*, a possible user of our approach).

In the literature, one can find semi-automatic approaches for automatically solving mismatches at the level of the interface's syntax (Yellin and Strom, 1997; Bracciali et al., 2005; Canal et al., 2006). They use LTSs and a means to define syntactical correspondences (Yellin and Strom, 1997; Bracciali et al., 2005), *e.g.*, a set of *synchronous vectors* (Arnold, 1994; Canal et al., 2006) that is assumed to be given as input to the approach. The knowledge that is required to give synchronous vectors or, in general, syntactical mappings as input is the same as the one required for developing a suitable component wrapper. Therefore, the previously considered assumption is not a limitation of our work with respect to the work described in Yellin and Strom (1997), Bracciali et al. (2005), Canal et al. (2006).

Other criteria should be considered for a more accurate component selection process (Ghosh et al., 2005; Wallnau et al., 2001), *e.g.*, criteria that are based also on QoS constraints. However, in this work we only focus on the component interaction protocol and we will consider more complex selection criteria as possible future work.

[1] The meaning of the label `C1.a.C2` in the constraint of the failure message `b` in Fig. 4B is that the sender of message `a` is `C1` while the receiver is `C2`.

Although we make the assumption to be only interested on the component protocol, it might be the case that we are not able to find, for each component in the verified SA, directly corresponding ones on the market (although there is a valid interface mapping among them). In fact, the interaction protocol of one or more cooperating actual components might not "fit" the one of an architectural component. Therefore, once we have selected a set of actual components as possible candidates for implementing an architectural component, we check if the parallel execution (Keller, 1976) of the selected actual components, *A*, fits the observable behaviour of an architectural component *I* (*i.e.*, the interaction protocol specified through its state machine). This is done by automatically building the state machine *A* and by testing two conditions: (i) *A* does not contain the observable behaviour of *I*; (ii) *A* either contains or is equivalent to the observable behaviour of *I*. In case (i), there is nothing that we can do since the assembly *A* cannot implement *I* in any way, *i.e.*, *A* is not a behavioural subtype of *I* (George et al., 2006; Oreizy et al., 1998). Therefore, we have to proceed manually implementing the architectural component. In case (ii), as described in Section 3.3, we use SYNTHESIS to automatically produce a centralized adaptor to be composed with *A*. The adaptor guarantees that *A* (plus the same adaptor), once put in execution, will exhibit only the behaviour of *I* (obviously if *A* exactly behaves as *I* the adaptor construction is trivial). The behaviour containment check is performed automatically by using the CADP toolbox (Garavel et al., 2002) in order to check if *A* simulates *I* under the well-known *weak-trace equivalence* (Park, 1981).

Now, continuing our explanatory example, let us consider three third-party components that we have selected and acquired in order to assemble the verified system. Let us suppose that we found on the market the component `AC1` that behaves exactly as specified for `C1`. On the contrary, suppose that we did not find on the market a component that corresponds exactly to `C2`. The best thing that we could do is to find two components (*i.e.*, `AC2.1` and `AC2.2`) whose interfaces[2] contain, in conjunction, the same interface as `C2`. The interaction protocol specification of these two components is shown in Fig. 5 in the form of state machines. The states drawn with a thicker border are the so-called *quiescent states*. For a formal definition of a quiescent state, we refer the reader to Kramer and Magee (1990). In our context, it is sufficient to say that a component is in a quiescent state whenever it has completed all component interactions required to perform some complex task and it has not yet started component interactions required for a new complex task. Quiescent states play a key role in performing the third phase of our method that will be described in Section 3.4.

By using the `EXP.OPEN` tool of the CADP toolbox, we can automatically build the state machine of the parallel composition of `AC2.1` and `AC2.2`. This state machine is shown in Fig. 6 as it is displayed by CADP.[3] The state labelled with 0 is the initial state.

In order to use CADP to build the state machine of the parallel composition of `AC2.1` and `AC2.2`, we have implemented an extension of CHARMY that allows one to export the component state machines into `.aut` files. The `.aut` notation represents one of the file formats supported by the CADP toolbox. Now, by using the `BISIMULATOR` tool of the CADP toolbox we can check whether or not the state machine shown in Fig. 6 simulates the state machine of `C2` (exported from CHARMY and given as input to CADP) under weak-trace equivalence. `BISIMULATOR` returns `TRUE` as result of the trace containment check, thus confirming that `AC2.1` and `AC2.2` are good candidates for implementing `C2`.

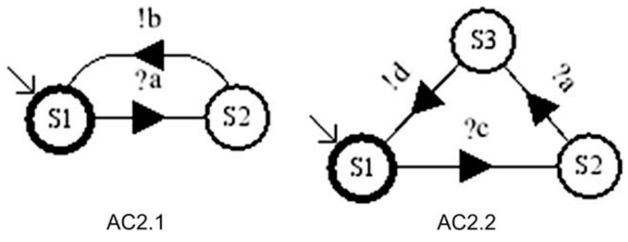


Fig. 5. State machines of `AC2.1` and `AC2.2`.

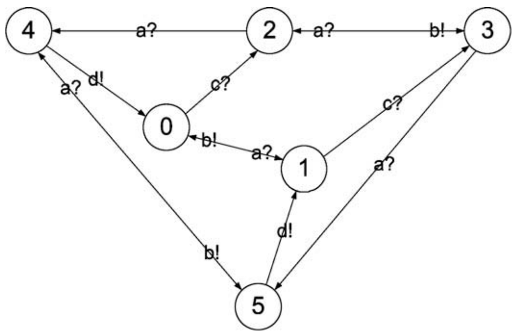


Fig. 6. State machine of the parallel composition of `AC2.1` and `AC2.2`.

3.3. Compile-time phase: component composition through static adaptation

The compile-time phase of our method uses SYNTHESIS to automatically synthesize an adaptor that, in our example, we denote as `AdtC2`. `AdtC2` is built to implement `C2` as an assembly of `AC2.1` and `AC2.2` (the adaptor represents the glue code). Furthermore, it prevents safety violations that may raise from possible interaction mismatches (*e.g.*, deadlock, violation of the `C2` observable behaviour and, hence, possibly, violation of the verified property). That is, `AdtC2` is synthesized in a way that the parallel composition of it with `AC2.1` and `AC2.2` is deadlock-free and behaves as specified by `C2`.

In Fig. 7 we show the state machine of `AdtC2` that has been automatically synthesized by using SYNTHESIS. The thicker state is the quiescent state of the adaptor. It is worth mentioning that a state of the adaptor state machine is a tuple of states of the actual component state machines. A quiescent adaptor state is a tuple of quiescent component states.

For the sake of brevity, we do not show the application of SYNTHESIS in detail here, but we describe only the steps needed for the comprehension of the approach that we are presenting.

Informally, the adaptor state machine is automatically built by considering the following criteria: (i) an adaptor has a strictly sequential input-output behaviour. That is, each message it receives is sent to the right component (that expects to receive that message). For instance, the adaptor shown in Fig. 7 can receive, from its initial state (*i.e.*, `S1`), a request of `a` from the environment of `C2`. After receiving this request, the adaptor delegates it to `AC2.1` (*i.e.*, the message `!a_AC2.1` from `S2`); (ii) at a first stage, the adaptor has to model all possible component interactions, *i.e.*, it is analogous to the *product automaton* (Keller, 1976; Arnold, 1994) of the state machines modelling the observable behaviour 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 `Env(C2)` the state machine of `C2` where sent messages have been converted into received ones, and vice versa. This is an *ideal* environment for `C2`

[2] Here, a component interface is seen as a list of provided/required methods.
[3] Note that in CADP the sent and received messages are denoted analogously to what is done in CHARMY. The only difference is that the symbols '?' and '!' are message label suffixes instead of prefixes.

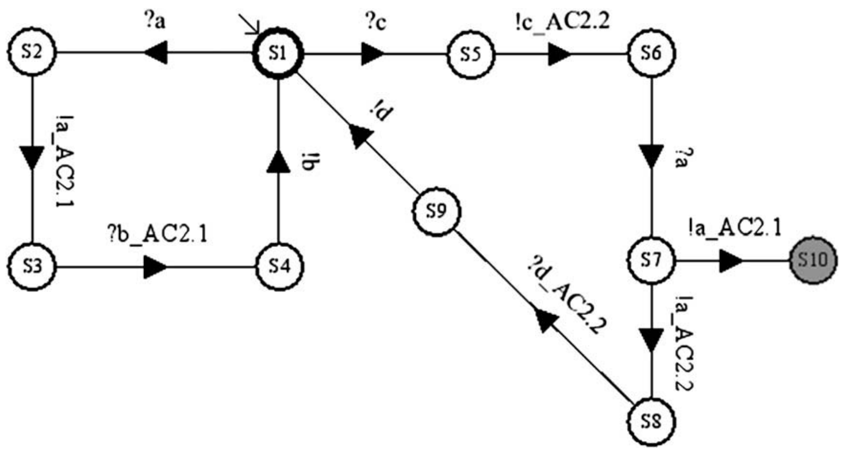


Fig. 7. Behavioural model of AdtC2.

since it completely preserves the C2 observable behaviour. In other words, Env(C2) models the less permissive environment expected by C2 in order not to block.

The adaptor, shown in Fig. 7, is built by considering the product automaton of AC2.1, AC2.2, and Env(C2) (module a suitable message relabelling of AC2.1 and AC2.2), and by imposing on it an input-output structure (*i.e.*, one transition of the standard product automaton becomes two transitions of the adaptor's state machine, *i.e.*, receive and send transitions).

By referring to Fig. 7, at this stage, the adaptor simply routes component messages and each message it receives is strictly sent to the right component. If there are deadlocks in the interaction among the actual components (AC2.1 and AC2.2) and the ideal environment of an architectural component (Env(C2)), SYNTHESIS detects them. Each "deadlocking" state is denoted by a dark-gray filled node. A deadlocking state is a sink state or a state leading only to deadlocking states.

In our example, a deadlock occurs when the message a from Env(C2) is sent to AC2.1 after that the message c from Env(C2) has been sent to AC2.2 (see the paths from S1 to S6 and from S6 to S10 shown in Fig. 7). Indeed, this deadlock models a safety violation concerning the possibility, for AC2.1 and AC2.2, to violate the observable behaviour specified for C2. For instance C2 behaves in a way that after the exchange of c followed by a, it is mandatory to exchange d. This implicit desired interaction can be violated if we implement C2 by composing AC2.1 and AC2.2 in an uncontrolled way (*i.e.*, without an adaptor/coordinator). To prevent this violation, SYNTHESIS automatically prunes all the deadlocking paths of the adaptor state machine (*i.e.*, the transition from S7 to S10). In this way, the state machine 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 AC2.1, AC2.2, and the deadlock-free adaptor AdtC2 has to simulate C2 under weak-trace equivalence. Let us denote that parallel composition by S. This check is required because, after all the violations have been prevented, C2 might exhibit behaviours that cannot be performed by S. Note that checking if C2 simulates S is useless since it is guaranteed by the construction of AdtC2. Analogously to what has been done for the actual component selection phase (Section 3.2), by using CADP, it is possible to automatically perform that check. To do this, we have implemented an extension of SYNTHESIS to support the exporting of the component state machines into .aut files.

By referring to the deadlock-free version (*i.e.*, the one without finite paths) of the adaptor state machine shown in Fig. 7, S has the same state machine where all the transitions labeled with actions performed by AC2.1 and AC2.2 (*i.e.*, the ones terminating with "AC2.1" and "AC2.2", respectively) are τ transitions. CADP allows us to confirm that S simulates C2 under weak-trace equivalence.

Using the same technique as described in Tivoli and Autili (2006); Autili et al., 2007, from the state machine of the deadlock-free AdtC2, SYNTHESIS is able to derive the actual code implementing the deadlock-free adaptor. The compile-time phase concludes by assembling the third-party components and the adaptor together. At this point, a running implementation of the verified system is automatically obtained and it is correct by construction.

The synthesized adaptors play an important role in supporting the run-time replacement of architectural components (as described in Section 3.4). In deriving the actual code of an adaptor, SYNTHESIS implements suitable mechanisms to bring the adaptor execution in a consistent state before architectural components replacement (Kramer and Magee, 1990). Furthermore, since the adaptor code is white-box, we can assume that a general-purpose *adaptor manager* can be built (Wang et al., 2006). The adaptor manager is used to orchestrate the system evolution at run-time.

3.4. Run-time phase: Architectural component replacement through dynamic adaptation

The dynamic adaptation phase of our method does not consider other possible system changes beyond architectural component replacement that could mean actual component addition, suppression, and replacement. In other words, at run-time, we can automatically generate new implementations of the SA that has been designed and verified during the design-time phase of the process. During the execution of the system, we keep stored both the state machines of the actual components $\{AC_i\}$ and the state machines of the architectural components $\{C_i\}$. Let us consider an architectural component C_i implemented (by means of the compile-time phase) as an assembly of actual components $AC_{i_1}, \ldots, AC_{i_n}$ and an adaptor Adt_i. When a developer decides to update the implementation of C_i with a new assembly made of the actual components $AC_{j_1}, \ldots, AC_{j_m}$, the adaptor manager performs off-line[4] the adaptor construction process described in Section 3.3. This is done to derive a new adaptor, Adt_i', that serves as assembly code for the components $AC_{j_1}, \ldots, AC_{j_m}$ in order to correctly implement C_i.

[4] The current version of the system is still running.

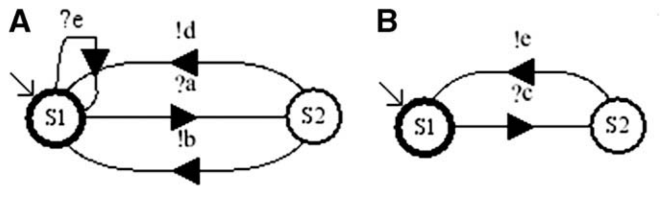


Fig. 8. State machines of (A) `AC2.3` and (B) `AC2.4`.

Once Adt'_i has been built, it is time (for the adaptor manager) to stop the execution of Adt_i in order to substitute it with Adt'_i, and to deploy the new actual components $AC_{j_1},\ldots,AC_{j_m}$ in place of $AC_{i_1},\ldots,AC_{i_n}$. Indeed, the adaptor manager invokes a `stop` method of Adt_i. The code of the `stop` method has been automatically derived by SYNTHESIS in order to prevent Adt_i to instantaneously stop, and make it wait to achieve a quiescent state, *i.e.*, Adt_i will stop only when a quiescent state has been reached. In other words, by looking at the components quiescent states, it is possible to establish the system states in which a change can be applied without disturbing the system execution. As done in Section 3.3, in our method the system engineer specifies, for each actual component, the set of quiescent states.[5] The quiescent state specification is used by Adt_i (during the execution of its `stop` method) in order to identify a quiescent state and, hence, establish when to stop.

Once Adt_i stops, it implicitly blocks received messages. At this point, the adaptor manager deploys $AC_{j_1},\ldots,AC_{j_m}$ and Adt'_i. Then, the adaptor manager uses the component reflection facilities to extract the current state of Adt_i and of $AC_{i_1},\ldots,AC_{i_n}$, and to transfer these states to Adt'_i and $AC_{j_1},\ldots,AC_{j_m}$, respectively. The state transfer is based on a state mapping function, furnished by the software engineer, that takes into account the results of the reflection and the state machines of the new actual components. Further investigation will be able to fully automate this step as in the case of the *state mapping algorithm* described in Vandewoude and Berbers (2005). This algorithm uses the "direct state transfer" technique requiring that a new component provides the necessary functionalities to import and interpret the state of its predecessor. Finally, the adaptor manager removes Adt_i and $AC_{i_1},\ldots,AC_{i_n}$ from the system, and starts the new version of the system by activating the execution of Adt'_i that starts to process component messages.

In this way, the initial version of the system is dynamically converted into a new one by still ensuring the verified properties. Note that this solution is not suitable for systems that specify "hard" timing constraints, e.g., real-time systems.

By continuing our explanatory example, we want to update the implementation of the architectural component `C2`. To do this, let us consider a scenario in which `AC2.1` and `AC2.2` must be replaced (at run-time) by two different components, *i.e.*, `AC2.3` and `AC2.4`. The state machines of `AC2.3` and `AC2.4` are shown in Fig. 8A and B, respectively.

The adaptor manager starts the off-line synthesis of a new version of `AdtC2` (*i.e.*, `AdtC2New`). Then, it calls the `stop` method on `AdtC2`. The adaptor `AdtC2` (see Fig. 7) will stop its execution only when it will reach a quiescent state (*i.e.*, `S1`). During the period in which `AdtC2` is stopped, it blocks the messages from and towards `AC2.1` and `AC2.2`. The adaptor manager uses reflection to retrieve the state of `AC2.1`, `AC2.2`, and of `AdtC2`. The system engineer applies the state mapping function (previously built off-line) to transfer those states to `AC2.3`, `AC2.4`, and `AdtC2New`, respectively. When the old components are removed the new ones are added, and `AdtC2New` is deployed in place of `AdtC2`, `AdtC2New` starts to consume the blocked messages hence following the normal execution flow.

[5] Although the actual components are black-box, the system engineer can identify the quiescent states by looking at the components' documentation and state machines.

4. Case Study: the SA.X project

In this section we report the main results that we obtained by applying our method to a real-world project based on an industrial-scale spacecraft system. In doing this, we also point out the size of the generated models. This size should not be considered as an evaluation criterion. It is reported just to give a little bit of evidence of the applicability of our approach. The case study originates from a collaboration between the *Dipartimento di Informatica, Università dell'Aquila*, and the *German subsidiary of Terma* located in Darmstadt (Cardone et al., 2005; TERMA Corporate, 2006). The initial goal of the collaboration was to formally verify SA.X with CHARMY (SA.X stays for *Software Architecture* for *XASTRO* which is a project developed in Terma). In this paper we borrowed, with minor modifications, the XASTRO system in order to use it in the context of dynamic evolution of component-based systems. The required modifications have been conducted maintaining the original size and complexity of the system. However, by exploiting the method described in Section 3, at design-time (see Section 4.1) we are allowed to consider a high level architectural description of the system that is reduced in complexity and size. The real size and complexity of the system will be taken into account in the compile-time phase (see Section 4.2). During this phase the verified architectural description has been automatically refined by implementing each component of the verified SA with a set of actual components acquired from a third-party.

The SA.X-TC_Chain system is an abstraction of the Commanding Chain of the SCOS_2000 spacecraft control system framework (SCOS2000, 2002; SCOS2000, 2002). The SA.X-TC_Chain acts as a communication bridge between a user (hereafter denoted by *User*), the Network Control and Telemetry Router System (denoted by *NCTRS*), and the Parameter InterFace (denoted by *PIF*). SA.X-TC_Chain allows one to create, dispatch and release a tele-command (TC), and to perform some checks during the release phase, up to the reception of messages by the Space Craft (see Fig. 9).

A TC is an atomically executable entity with a certain number of attributes called parameters. The parameters concern not only argument values, but also some directives used during the TC release end execution, such as release time (*e.g.*, *ASAP*, *Time-tagged*, *WAIT*), execution time (for time-tagged TC), or conditions related to the TC validation and verification criteria. A TC, when emitted, might not be processed immediately and, hence, it is firstly loaded in a stack structure waiting to be processed later.

4.1. Design-time phase

In Fig. 10, we show the SA of the SA.X-TC_Chain system that is composed by four different components: *User*, *PIF*, *TC_Chain*, and *NCTRS*. *NCTRS*, *PIF*, and *User* communicate only with *TC_Chain*. The interface required by *User* defines six messages: *TC_Load*,

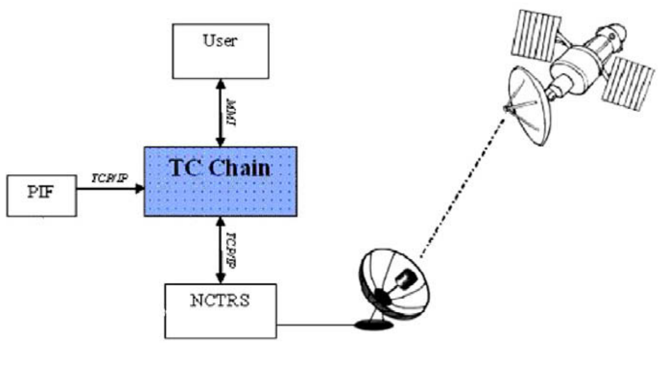


Fig. 9. The SA.X-TC_Chain.

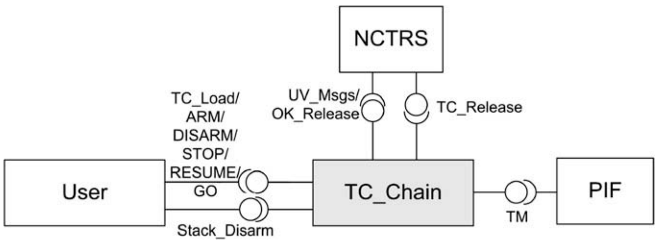


Fig. 10. The SA.X-TC_Chain Software Architecture.

ARM, *DISARM*, *STOP*, *RESUME*, and *GO*. The provided interface of *User* defines the *Stack_Disarm* message. The required interface of *NCTRS* defines *UV_Msgs*, and *OK_Release* while the provided one defines *TC_Release*. *TM* is the message defined by the required interface of *PIF*. The provided (*resp.*, required) interfaces of *TC_Chain* match with the required (*resp.*, provided) interfaces of the components connected to it.

User loads one or more TCs (*i.e.*, *TC_Load*) onto the stack, and then arms the TC on top of the stack (*i.e.*, *ARM*). Finally, it asks the *TC_Chain* to dispatch the loaded TCs (*i.e.*, *GO*). The TC on top of the stack can be stopped (*i.e.*, *STOP*) or disarmed (*i.e.*, *DISARM*). The *Stack_DISARM* action is a directive sent by *TC_Chain* and it serves to disarm all the loaded TCs. *RESUME* restarts and disarms the most recently loaded TC. *TC_Load*, *ARM*, *GO*, *STOP*, *RESUME*, and *DISARM* messages are requests sent by *User* to *TC_Chain*. *Stack_Disarm* is a receive event. *PIF* permits to set the parameters of a TC (*i.e.*, *TM*). *TM* is sent by *PIF* and received by *TC_Chain*.

In order to dispatch a TC, *NCTRS* receives a request for a TC release from *TC_Chain* (*i.e.*, *TC_Release*). Then, *NCTRS* acknowledges *TC_Chain* that the request of a TC release has been processed (*i.e.*, *OK_Release*), and sends information concerning the TC release status, such as "successfully released", "unsuccessfully released", or other information (*i.e.*, *UV_Msgs*). Note that processing a TC release does not assure that the TC will be executed. For instance, in case of an unsuccessful release, the TC will be not activated by *TC_Chain*. Since *TC_Chain* communicates with all the other components by means of the messages described above, its modelled behaviour does not need further explanations.

With respect to this SA model, we verified two properties. *Property 1* expresses that, after a *TC_Load* followed by a *GO*, if *STOP* is not exchanged then *TC* must be released. *Property 2* specifies the following requirement: after a *TC_Load*, the system fails if a *TC_Release* happens and it has not been preceded by *GO*.

In order to model-check the properties we selected the *Liveness (cycles/sequences)* and *Apply Never Claim* options in the *Basic Verification Options* SPIN panel, and we specified the actual amount of physical memory available (*Physical Memory Available* option in the *Advanced Verification Options* panel) to 1000 Mb. The experiment has been performed on a Pentium 1.73GHz with 2GB of RAM.

The verification of *Property 1* took some seconds with 496 internal states and using 2.302 MB of memory. SPIN has reported no errors, which gives us the guarantee that the defined SA complies with the considered requirement. *Property 2* has been verified by taking 408 internal states and almost the same memory as *Property 1*. Also in this case, SPIN has reported no errors.

While running the design-time phase of our approach we experienced that by combining architectural analysis and code synthesis, at design-time we can consider a very high-level SA model. Otherwise, if we would have used only CHARMY then we should have considered the more low-level SA model shown in Fig. 11. This is due to the fact that, without the support provided by the code synthesis, in order to facilitate the developers tasks, the SA model should be reasonably close to its implementation (see Fig. 11), as it is for a standard use of CHARMY in a real-scale context. For evaluation purposes, we tried to verify the SA model shown in Fig. 11 against *Property 1* and *Property 2*. Due to the memory size limit, CHARMY stopped its verification process without giving an answer (*i.e.*, the answer was successful but the explored state-space was only a portion of the total one, thus invalidating the answer).

4.2. Actual component selection and Compile-time phases

When looking for actual components that can implement our architectural components we found suitable components for *User*,

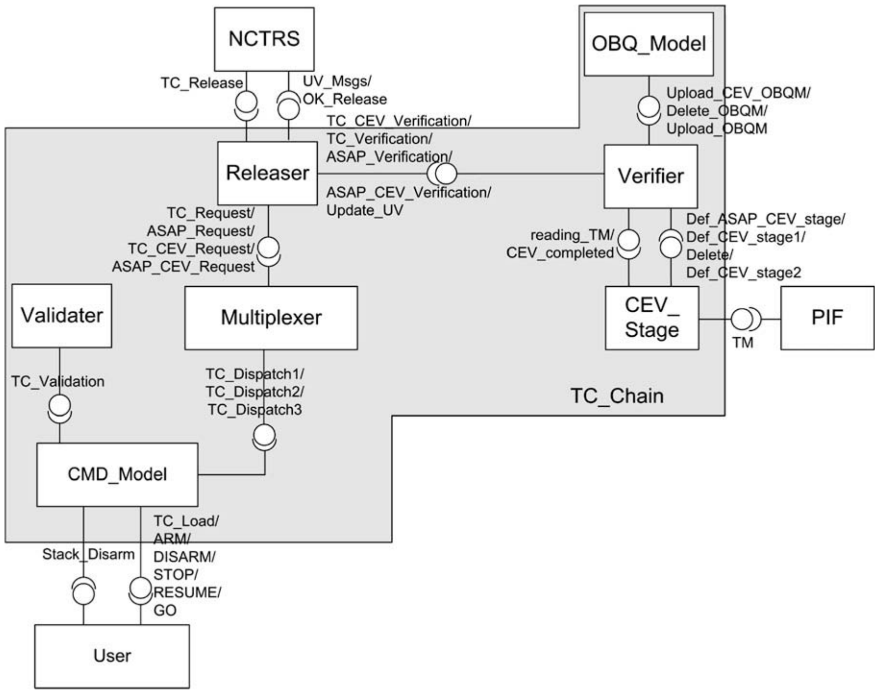

NCTRS
TC_Release
UV_Msgs/ OK_Release
OBQ_Model
Upload_CEV_OBQM/ Delete_OBQM/ Upload_OBQM
TC_CEV_Verification/ TC_Verification/ ASAP_Verification/
ASAP_CEV_Verification/ Update_UV
Releaser
Verifier
TC_Request/ ASAP_Request/ TC_CEV_Request/ ASAP_CEV_Request
reading_TM/ CEV_completed
Def_ASAP_CEV_stage/ Def_CEV_stage1/ Delete/ Def_CEV_stage2
Validater
Multiplexer
CEV_ Stage
PIF
TM
TC_Validation
TC_Dispatch1/ TC_Dispatch2/ TC_Dispatch3
TC_Chain
CMD_Model
Stack_Disarm
TC_Load/ ARM/ DISARM/ STOP/ RESUME/ GO
User

Fig. 11. SA.X-TC_Chain actual components.

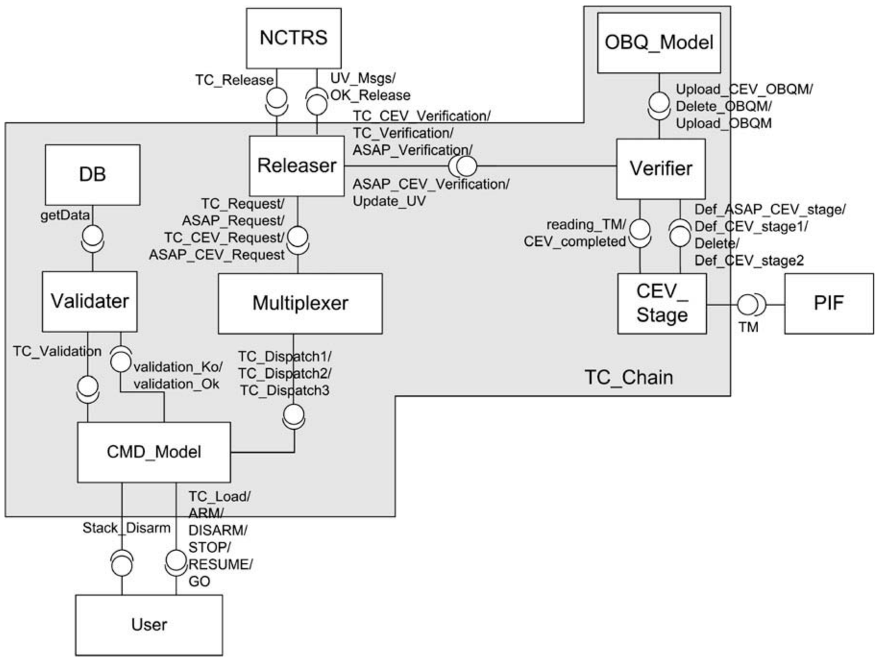


Fig. 12. Actual components of the new version of SA.X-TC_Chain.

NCTRS, and *PIF* but for the *TC_Chain* we found seven components that, hopefully, put in parallel behave as desired. As explained in the previous sections, SYNTHESIS has been used to automatically build the assembly code for those seven components and their environment. This assembly code has been derived deadlock-free and behaviourally equivalent to the behaviour of the architectural component *TC_Chain*.

In Fig. 11, we show the SA model of the system automatically implemented by considering *TC_Chain* as an assembly of those seven components (plus the synthesized glue code, *i.e.*, the adaptor).

The description of the seven actual components is not crucial for the purposes of the case study and, hence, for the sake of brevity, it is omitted.

The adaptor construction process performed by SYNTHESIS took 17 min. The adaptor state machine has 9593 states, and 460 deadlocking states. The memory usage has been 37 MB. The deadlock prevention procedure, subsequently performed, took 2.5 s and the state machine of the resulting deadlock-free adaptor has 9133 states. The memory usage has been 10 MB. Finally, the weak-trace equivalence check performed by using CADP took 2 s. It returned `TRUE` meaning that the protocol equivalence of *TC_Chain* has been preserved. This means that the selected seven components plus the deadlock-free adaptor represent a correct-by-construction implementation of *TC_Chain*.

While running the compile-time phase of our approach we experienced that by combining architectural analysis and code synthesis, at compile-time we can use SYNTHESIS locally with respect to each architectural component by reducing the state-space explosion phenomenon. Otherwise, if we had used only SYNTHESIS then we should have considered exactly all components of the SA model shown in Fig. 11 plus *Property 1* and *Property 2*. This is due to the fact that, without the support provided by the architectural analysis, we cannot consider a (reduced in size) ideal environment against which to assemble the actual components, but we have to build it by putting in parallel all the architectural components that differ from the one that we want to implement plus the verified properties. For evaluation purposes, we tried to build an adaptor for all the components shown in Fig. 11 with respect to guaranteeing *Property 1* and *Property 2* (*i.e.*, the standard approach in SYNTHESIS). Due to the memory size limit, SYNTHESIS exited unsuccessfully without generating an adaptor.

Note also that the situation considered in this case study is a limit situation in which one component is substituted by seven different components. We have chosen to do that to show that SYNTHESIS, although it suffers from the state-space explosion phenomenon, can deal with quite large systems and that the approach we are presenting in this paper is flexible. In general we expect to manage smaller reconfigurations.

4.3. Run-time phase

In Fig. 12, we show the SA of the new version of SA.X-TC_Chain. The applied update concerns the TC validation functionality. Now, *Validater* performs a more accurate validation check. Namely, it exploits additional information stored in *DB*. After *Validater* has performed the validation, it sends the validation result to *CMD_Model*. Accordingly, *CMD_Model* is updated as well.

The adaptor manager starts the generation of the new required adaptor. It took 92 min. The adaptor state machine has 17,837 states and 583 deadlocking states. The memory usage has been 62 MB. The deadlock prevention procedure took 6.8 s by generating 17,254 states. The memory usage has been 18 MB. Note that, as explained in Section 3, the adaptor generation phase is performed off-line, *i.e.*, without the need of stopping the system.

5. 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 that are closest to our approach. The most strictly related approaches concern the problem of dynamically composing and adapting software components, and the area of protocol adapters. We organize the description of these works into three main research areas: component frameworks to the run-time reconfiguration, described in Section 5.1, dynamic adaptation of component-based systems, discussed in Section 5.2, and protocol adapters, discussed in Section 5.3.

5.1. Component frameworks to the run-time reconfiguration

Jadda (Java Adaptive component for Dynamic Distributed Architectures) (Falcarin and Alonso, 2004) is a framework that relies on architecture specification to support dynamic reconfiguration. It uses xADL[6] and no formal support is provided for constraining dynamic reconfiguration. Jadda's support for ad-hoc reconfiguration is accomplished via a console that is used to submit an xADL file with the specified change. Jadda is limited to ad-hoc reconfiguration with no formal support. It thus lacks an automatic support to guarantee the change consistency with respect to the system properties of interest.

In Fractal - Specification, a component is both a design and a run-time entity with explicit provided and required interfaces. Each component is composed of a finite number of other components, which are under the control of the controller of the enclosing component. The Fractal framework is a Java software framework that supports component-based programming according to the Fractal model. It is an open framework that comprises a *core* and several *increments*. The core defines the minimal concepts and the APIs necessary for Fractal-based component programming. Increments define additional concepts and the APIs which extend the core framework to allow for different forms of component composition, configuration, and administration. A Java implementation of the Fractal framework has been developed. It is called Julia (Fractal - Julia) and offers three different forms of configuration: static, dynamic, and partially dynamic. Although, in order to reuse SYNTHESIS as easily as possible, we applied our approach in the context of both COM/DCOM (Tivoli and Autili, 2006) and EJB (Autili et al., 2007) systems, we are currently investigating the usage of Julia as a possible implementation choice for our "evolving" glue adaptors.

SOFA (Bures et al., 2006) is a component model with a number of advanced features. It allows for dynamic evolution of architectures at run-time. The controlled evolution of the SA is driven by well-defined evolution patterns. These patterns are supported by the runtime environment which handles reconfigurations accordingly. A *factory* pattern is used to create and add a new component. A *removal* pattern instead is used to destroy a component that was previously created. In SOFA/DCUP (Plasil et al., 1997, 1998) (Dynamic Component UPdating), each component defines one component manager (CM) and one component builder (CB), which are responsible of managing the associated component. A component may have several implementation objects and/or sub-components that provide its functionality. A component is divided into two parts: a *permanent part* and a *replaceable part*. Therefore, it provides two kinds of operations, control operations and functional operations. Adapting a component means replacing its replaceable part by a new version at run-time. When a sub-component of a global component has to be adapted, the whole component is affected, and its replaceable part is re-deployed, therefore, the entire application has to be re-deployed. DCUP does not provide any degree of automation, *i.e.*, all the adaptation operations must be done by the administrator.

Batista et al. (2005) propose a meta-framework called "Plastik" which supports the specification and creation of component-based systems by facilitating and managing the run-time reconfiguration of such systems while ensuring integrity across changes. This meta-framework is an integration of an architecture description language (an extension of ACME/Armani Garlan et al., 2000; Monroe, xxxx) and a reflective component framework called OpenCOM (Coulson et al., 2004). Plastik generates component systems that can be dynamically reconfigured either through programmed changes or through ad-hoc changes. The run-time level is based on the OpenCOM framework. Components are encapsulated units of functionality and deployment that interact with their environment (*i.e.*, the other components in the system) exclusively through interfaces. By default, components are written in C++. The OpenCOM framework supports a set of so-called *reflective meta-models* which facilitate reconfiguration of systems by permitting different system aspects to be inspected, adapted, and extended at run-time. A *run-time configurator* is responsible for managing the OpenCOM run-time layer. Although there are many common aspects between the work described in this paper and the OpenCOM framework, one main difference is that OpenCOM essentially considers components as white-box entities and, hence, it does not allow the handling of third-party and black-box components, which is a key aspect in our approach.

The OSGI Platform (OSGi) provides standardized primitives that allow applications to be constructed from small, reusable and collaborative components (implemented in Java). These components can be composed into an application and deployed. The OSGI technology provides a Dynamic (or Service-Oriented) Software Architecture with functions to change the components composition dynamically without requiring restarts. The basic unit of deployment is a *bundle*, which provides services and which can depend on and uses other services. A bundle can be seen as a primitive component and its services as interfaces of the component. Moreover the OSGI framework allows bundles to select an available implementation at run-time through the framework service registry. Bundles register new services, receive notifications about the state of services, or look up existing services to adapt to the current capabilities of the device. This aspect of the framework makes an installed bundle extensible after deployment: new bundles can be installed for adding features or existing bundles can be modified and updated without requiring the system to be restarted. Once a bundle is started, its functionality is provided and services are exposed to other bundles installed in the OSGI Service Platform. For each bundle installed in the OSGI framework, there is an associated *Bundle Object* (OSGi) that is used to manage its life cycle.

All the presented approaches are based on specific component models. In our approach we do not make any supposition on the component model of the application to be adapted, thus providing a solution that is independent of a specific model.

5.2. Dynamic adaptation of component-based systems

As part of the RAPIDware project, (Zhang et al., 2005) 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 (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 and Biyani (2004) 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

6 http://www.isr.uci.edu/projects/xarchuci/.

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. (2003) propose 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 processes are currently using any function of the object. We argue that this condition is not sufficient in cases where a critical communication segment between two components includes a series of function invocations. Also, they do not address global conditions for safe dynamic adaptation.

Amano and Watanabe (2002) introduce 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.

Mechanisms used to interconnect components and their ability to cope with architectural mismatches is another related research area. An architectural mismatch occurs when the assumption that a component makes about another component, or the rest of the system, does not match. In Gacek's Ph.D. dissertation (Gacek, 1998) a static mismatch detection method has been proposed. The overall idea of this work is that by analyzing the characteristics of the architectural elements to be integrated and the styles from which these elements were derived, the system architects are able to localize architectural mismatches during development time. Finally, the Architect's Automated Assistant (AAA)[7] tool has been used to detect mismatches during component composition. The limit of this approach, in comparison with ours, is that it is so specific to the context of software development that it cannot be used for run-time detection of error caused mismatches. de Lemos et al. (2003) starting from the previous work have presented how these mismatches can be tolerated during run-time at the architectural level. They have applied general principles of fault tolerance to deal with architectural mismatches. With respect to our work, it is only a preliminary analysis showing a number of particular mismatch tolerance techniques that can be developed depending on the application, architectural styles, types of mismatches, redundancies available, etc. The applicability of this approach to real systems is still an open issue and the authors are trying to define in a more rigorous way the applicability of the approach and its set of general mismatch tolerance techniques.

5.3. Protocol adapters

Our research is also related to work in the area of protocol adaptor synthesis developed by Yellin and Strom (1997). The main idea of this work 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 behaviour 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 with 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 knowing 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 Tivoli and Autili (2006), we extended the synthesis process implemented by SYNTHESIS in order not to only restrict the coordinator behaviour but also to augment it in order to consider also such incompatibilities, *e.g.*, interface signature mismatches.

Spritznagel et al. in Spitznagel and Garlan (2003) propose an approach to specify wrappers, independent of any particular context of use, and use this specification to understand things such as the impact of its use, its effects on the communication protocol between components, compositional properties, etc. The wrappers that they consider are connector wrappers that are primarily designed to affect the communication between components. They define a connector wrapper formally as a protocol transformation and adopt an approach based on process algebras (*e.g.*, FSP). To describe a connector protocol in FSP, they use an approach similar to Allen (1997)); a connector is defined as a set or processes. Moreover three classes of properties are analyzed (*e.g.*, soundness, transparency and compositionality). Our notion of an adaptor is similar to the notion of a "complex" connector defined in Spitznagel and Garlan (2003). In fact, although in our approach connectors are simple communication channels, an adaptor can be seen as a first-class extension of a connector providing coordination facilities.

In work by Bracciali et al. (2002), 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 behaviour, (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 behaviour. 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 ours. Since, when 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.

Other strictly related approaches are in the "*scheduler synthesis*" research area. In the discrete event domain they appear as "*supervisory control*" or "*discrete controller synthesis*" problem (Brandin and Wonham, 1994; Ramadge and Wonham, 1987) addressed by Wonham, Ramadge et al. In very general terms, these works can be seen as an instance of a problem similar to the problem treated in our approach. However, the application domain of these approaches is sensibly different from the software component domain. Dealing with software components introduces a number of further problematic dimensions to the original synthesis problem. In the scheduler synthesis approaches the possible system executions are modelled as a set of event sequences and the system specification describes the desired executions. The role of the supervisory controller is to interact with the system in order to meet system specification. The aim of these approaches is to restrict the system behaviour so that it is contained in a desired behaviour, called the *specification*. To do this, the system is constrained to perform events only in strict synchronization with another system,

[7] http://sunset.usc.edu/available_tools/AAA/.

called the *supervisor* (or *controller*). This is achieved by automatically synthesizing a suitable supervisor with respect to the system specification. In contrast to our method, there is one main assumption to deal with deadlocks: in order to automatically synthesize a *supervisor* which avoids deadlocks, they need to consider a specification of the deadlocking behaviours of the base system (*i.e.*, the event sequences that might cause deadlocks). This is a problem because, for large systems, the designers might not know the deadlocking behaviours simply because they might be unpredictable.

Promising formal techniques for the compositional analysis of component-based design have been developed in de Alfaro and Heinzinger (2001), Passerone et al. (2002). The key of these works is the modular-based reasoning that provides a support for the modular checking of behavioural properties. In de Alfaro and Heinzinger (2001), De Alfaro and Henzinger use an automata-based approach to capture both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The formalism supports automatic compatibility checks between interface models, and thus constitutes a type system for components interaction. The purpose of this work is different from ours. The authors check that two components have compatible interfaces if a legal environment that lets them correctly interact exists. Each legal environment is an adaptor for the two components. They provide a consistency check only among components interfaces. That is, they do not deal with automatic synthesis of component interface adaptors (*i.e.*, automatic synthesis of legal environments). However, in Passerone et al. (2002) De Alfaro, Henzinger, Passerone and Sangiovanni-Vincentelli use a game-theoretic approach for checking whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. This approach is able to automatically synthesize the converter. In contrast to the work presented in this paper, with respect to deadlock-freedom, the specification of the converter's requirements is assumed to be correct. Thus, if for example the specification would erroneously introduce deadlocks, they would not be prevented by the converter that it is synthesized in order to be completely compliant to its requirements specification. In other words, a *deadlock preventing* specification of the requirements to be satisfied by the adaptor has to be provided by delegating to the user the non-trivial task of specifying it.

6. Discussion

In this section we discuss the kind of system architectural configurations that we are able to manage. Furthermore, we recall the advantages experienced by combining CHARMY and SYNTHESIS and by performing the case study described in Section 4.

By considering the component and connector architectural elements, let us consider the following definitions (Caporuscio et al., 2004):

Closed system: A Closed System is a system with a fixed number of component instances and fixed connectors.
Weakly-closed system: A Weakly-Closed System is a system with a fixed number of component instances.
Weakly-opened system: A Weakly-Opened System is a system with variable number of component instances and with fixed connectors.
Open system: An Open System is a system with variable connectors and variable number of component instances.

Our method assures that the architectural component interactions verified on the SA at design-time, are preserved in every generated implementation of the system. The price to be paid for this is that, at the architectural level, we can only deal with Closed Systems although at the implementation level we can deal with Closed, Weakly-Closed, Weakly-Opened and Open Systems. It is up to the software engineer that designs the system to decide the right trade-off between the *granularity* of the SA design and the *adaptability* of its implementation. This is done by choosing the desired abstraction level that the SA to be verified should have. The SA could be very close to its implementation (as it is in the case of the running example described in Section 3). In this case, on the one hand, it is possible to assure that the SA implementation preserves component interactions specified at a low-level of abstraction; on the other hand, the adaptability of the implemented system is low (*i.e.*, the implemented system is Closed). Conversely to this, the SA could be very far from its implementation (as it is in the case of the case study of Section 4). In this case, the SA implementation preserves only component interactions specified at a high-level of abstraction (*e.g.*, it is not possible to define properties concerning the interaction of a set of actual components implementing an architectural one) but its adaptability is very high meaning that the implemented system ranges from Weakly-Closed to Open Systems.

To make the approach described in this paper as automatic as possible, we have implemented an integration of CHARMY and SYNTHESIS in order to make both tools coexist in the same framework and uniform their state machine notations. By using this integrated framework, we have experienced that combining architectural analysis and code synthesis together brings some advantages with respect to use only one of those approaches in isolation.

Note that, although, in general, the synthesis process suffers from the state-space explosion phenomenon, our approach makes it feasible. In fact, it exploits the system SA model (previously verified) in order to perform adaptation locally to each specified architectural component rather than at the level of the global system interactions. In this way, in our approach, the synthesis process has to face a problem that has a reduced complexity in terms of its state space. 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. As described in Section 3.3, in order to automatically build the adaptor state machine, SYNTHESIS can consider as environment for the architectural component to be implemented its ideal environment, rather than of the parallel composition of all other architectural components in the system SA plus the specified properties. Note that, for large systems, considering as environment a single component instead of the parallel composition of many components (plus the specified properties) represents, in general, a significant optimization. Therefore, in our context, the approach implemented by SYNTHESIS (that, in general, suffers from the state-space explosion phenomenon) is more feasible in practice.

The advantage that CHARMY takes from being combined with SYNTHESIS is that, when possible, the component glue code required for the SA implementation is automatically built correct by construction without requiring the developers to work on it. This obviously reduces the development time as well as the time spent to verify that the system implementation conforms to its requirements.

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 6, the component-based systems we deal with can range from Closed to Open systems. That is, the described approach allows the system to be able to evolve, at run-time, with respect to architectural updates at the actual component level such as component replacement, addition, or suppression. The models that we use are state machines and sequence diagrams. The combination of architectural analysis and code synthesis is performed by combining two approaches that were previously developed by some of the authors. One approach is implemented by CHARMY for performing architectural analysis of an SA model. The other one is implemented by SYNTHESIS for performing code synthesis. CHARMY and SYNTHESIS are integrated in the same framework. In order to make the approach described in this paper fully automatic, our framework uses also some tools of the CADP toolbox.

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

Future work concerns the selection criteria used to select and acquire the actual components from the market. To make our method a systematic engineering approach, processes that help the developer in performing a more accurate COTS components selection phase must be investigated (see Alves and Castro, 2001; Chung and Cooper, 2004 and the 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 in which taking into account the elapsing of time of a component request is a critical task, such as embedded real-time systems. Finally, *direct* automatic state transfer techniques (see (Vandewoude and Berbers, 2005) and the references therein) must be investigated in order not to have to handle the state transfer process manually, hence providing automatic support also for this phase that may be error-prone and tedious.

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.

The work is also partially supported by ARTDECO (Adaptive infRasTructure for DECentralized Organizations), an Italian FIRB (Fondo per gli Investimenti della Ricerca di Base) 2005 Project.

The authors wish to thank TERMA GmbH, which inspired the case study, Mariarosaria Cardone who modelled and analyzed the original version of the SA.X-TC Chain system, and Henry Muccini that was the responsible of the collaboration in the University of L'Aquila.

Comments and suggestions of anonymous referees are gratefully acknowledged.

References

Allen, R., 1997. A formal approach to software architecture. Carnegie Mellon, School of Computer Science, Issued as CMU Technical Report CMU-CS-97-144.
Alves, C., Castro, J., 2001. Cre: A systematic method for cots components selection. In: SBES'01.
Amano, N., Watanabe, T., 2002. 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.
Appavoo, K.H.J., Hui, K., Soules, C.A.N., Wisniewski, R.W., Da Silva, D.M., Krieger, O., Auslander, M.A., Edelsohn, D.J., Gamsa, B., Ganger, G.R., McKenney, P., Ostrowski, M., Rosenburg, B., Stumm, M., Xenidis, J., 2003. Enabling autonomic behavior in systems software with hot swapping. IBM System Journal 42 (1).
Arnold, A., 1994. Finite Transition Systems. International Series in Computer Science. Prentice-Hall.
Autili, M., Inverardi, P., Tivoli, M., Garlan, D., 2004. Synthesis of 'correct' adaptors for protocol enhancement in component-based systems. In: Proceedings of SAVCBS'04 Workshop at FSE.
Autili, M., Pelliccione, P., Inverardi, P., 2007. Graphical scenarios for specifying temporal properties: an automated approach. Automated Software Engineering journal 14 (3), 293–340. Sep.
Autili, M., Inverardi, P., Navarra, A., Tivoli, M., 2007. SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In: International Conference on Software Engineering (ICSE2007) – Formal Tool Demos Track, Minneapolis (USA), May.
Batista, T.V., Joolia, A., Coulson, G., 2005. Managing Dynamic Reconfiguration in Component-Based Systems. EWSA.
Bracciali, A., Brogi, A., Canal, C., 2002. Systematic component adaptation. ENTCS 66 (4).
Bracciali, A., Brogi, A., Canal, C., 2005. A formal approach to component adaptation. Journal of Systems and Software 74 (1), 45–54.
Brandin, B.A., Wonham, W.M., 1994. Supervisory control of timed discrete-event systems. IEEE Transactions on Automatic Control 39 (2).
Buchi, J.R., 1960. On a decision method in restricted second order arithmetic. In: Proc. of the International Congress of Logic, Methodology and Philosophy of Science. Standford University Press, pp. 1–11.
Bures, T., Hnetynka, P., Plasil, F., 2006. SOFA 2.0: balancing advanced features in a hierarchical component model. In: SERA.
Canal, C., Poizat, P., Salaün, G., 2006. Synchronizing behavioural mismatch in software composition. In: FMOODS, vol. 4037, LNCS.
Caporuscio, M., Inverardi, P., Pelliccione, P., 2004. Formal analysis of architectural patterns. In: First European Workshop on Software Architecture – EWSA, 21–22 May 2004, St. Andrews, Scotland.
Cardone, M., 2005. Experiencing Architectural Analysis in Industrial Contexts. Master's thesis, Computer Science Department, University of L'Aquila, Italy, December.
Charmy Project, 2004. Charmy web site. <http://www.di.univaq.it/charmy>.
Chung, L., Cooper, K., 2004. Matching, ranking, and selecting components: A cots-aware requirements engineering and software architecting approach. In: MPEC'04.
Clarke, E.M., Grumberg, O., Peled, D.A., 2001. Model Checking. The MIT Press, Massachusetts Institute of Technology.
Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J., 2004. A component model for building systems software. In: IASTED Conference on Software Engineering and Applications.
Crnkovic, I. et al., 2002. Anatomy of a research project in predictable assembly. In: 5th ICSE Workshop on Component Based Software Engineering. ACM. May.
de Alfaro, L., Heinzinger, T., 2001. Interface automata. In: ACM Proc. of the joint 8th ESEC and 9th FSE. ACM Press. Sep.
de Lemos, R., Gacek, C., Romanovsky, A., 2003. Architectural Mismatch Tolerance. LNCS 2677. Springer-Verlag.
Falcarin, P., Alonso, G., 2004. Software Architecture Evolution through Dynamic AOP. LNCS 3047. Springer-Verlag.
Objectweb. Fractal – Julia. <http://fractal.objectweb.org/julia/index.html>.
Objectweb. Fractal – Specification. <http://fractal.objectweb.org/specification/index.html>.
Gacek, C., 1998. Detecting Architectural Mismatches During Composition. Center for Software Engineering, University of Southern California, Los Angeles, CA 90089.
Garavel, H., Lang, F., Mateescu, R., 2002. An overview of CADP 2001. EASST Newsletter, 4. <http://www.inrialpes.fr/vasy/cadp>.
Garlan, D., Monroe, R., Wile, D., 2000. ACME: Architectural Description of Component-based Systems. In: Leavens, G.T., Sitaraman, M. (Eds.), Foundations of Component-based Systems. Cambridge University Press, pp. 47–68.
George, B., Fleurquin, R., Sadou, S., 2006. A component-oriented substitution model. In: ICSR 2006: 9th International Conference on Software Reuse, LNCS 4039, Turin, Italy, June 12–15.
Ghosh, S., Kelly, J.L., Shankar, R.P., 2005. Enabling the Selection of COTS Components, COTS-Based Software Systems, LNCS 3412.
Heineman, G.T., Councill, W.T. (Eds.), 2001. Component-Based Software Engineering. Addison-Wesley.
Holzmann, G.J., 2003. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley. September.
Inverardi, P., Tivoli, M., 2003. Software architecture for correct components assembly - chapter. In: Formal methods for the design of computer. communication and software systems: Software architecture. LNCS 2804. Springer. September.
Inverardi, P., Muccini, H., Pelliccione, P., 2005. Charmy: an extensible tool for architectural analysis. In: ESEC/FSE-13. ACM Press, New York, NY, USA, pp. 111–114.
Keller, R., 1976. Formal verification of parallel programs. Communications of the ACM 19 (7), 371–384.
Kramer, J., Magee, J., 1990. The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. 16 (11), 1293–1306.
Kulkarni, S.S., Biyani, K.N., 2004. Correctness of Component-based Adaptation, in CBSE7, May.
Manna, Z., Pnueli, A., 1992. The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc.
Monroe, R., 1998. Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163, Carnegie Mellon University.

Oreizy, P., Medvidovic, N., Taylor, R.N., 1998. Architecture-based runtime software evolution. In: ICSE '98: Proceedings of the 20th international conference on Software engineering. IEEE Computer Society, Washington, DC, USA, pp. 177–186.

Open Services Gateway Initiative (OSGi). <http://www.osgi.org/>.

Park, D., 1981. Concurrency and Automata on Infinite Sequences. In: Theoretical Computer Science. LNCS, vol. 104. Springer-Verlag, pp. 167–183. March.

Passerone, R., de Alfaro, L., Heinzinger, T., Sangiovanni-Vincentelli, A.L., 2002. Convertibility verification and converter synthesis: two faces of the same coin. In: Proceedings of the ICCAD.

Plasil, F., Balek, D., Janecek, R., 1997. DCUP: Dynamic Component Updating in Java/CORBA Environment. Dep. of SW Engineering. Charles University, Prague, Tech. Report No. 97/10.

Plasil, F., Balek, D., Janecek, R., 1998. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In: Proceedings of ICCDS'98, IEEE CS Press.

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

Schmidt, W.H., Crnkovic, I., Heineman, G.T., Stafford J.A., (Eds.), 2007. Component-Based Software Engineering. 10th International Symposium, CBSE 2007, Medford, MA, USA, July 9–11, 2007, Proceedings, Springer 2007. ISBN 978-3-540-73550-2.

SCOS2000, 2002. Commanding Architectural Design Document. Technical report, European Space Agency.

SCOS2000, 2002. Commanding Software Requirements Document. Technical report, European Space Agency.

Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an emerging Discipline. Prentice Hall, Englewood Cliffs, NJ.

Spitznagel, B., Garlan, D., 2003. A Compositional Formalization of Connector Wrappers. The 2003 International Conference on Software Engineering (ICSE'03), Portland, Oregon, USA, May 3–10.

Synthesis Project, 2004. Synthesis web site. <http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.php>.

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

TERMA Corporate Web Site, 2006. <http://www.terma.com>.

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

Vandewoude, Y., Berbers, Y., 2005. Component state mapping for runtime evolution. In: Proceedings of the 2005 International Conference on Programming Languages and Compilers.

Wallnau, K.C., Hissam, S.A., Seacord, R.C., 2001. Building Systems From Commercial Components. SEI Series in Software Engineering. Addison-Wesley.

Wang, Q., Shen, J., Wang, X., Mei, H., 2006. A component-based approach to online software evolution: Research articles. J. Softw. Maint. Evol. 18 (3), 181–205.

Yellin, D., Strom, R., 1997. Protocol specifications and component adaptors. ACM Trans. on Programming Languages and Systems 19 (2), 292–333. March.

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

Zhang, J., Cheng, B.H.C., Yang, Z., McKinley, P.K., 2005. Enabling safe dynamic component-based software adaptation. In: Architecting Dependable Systems III. LNCS. Springer-Verlag.

Patrizio Pelliccione is an assistant professor at the University of L'Aquila, Computer Science Department. He got his PhD degree in the University of L'Aquila, computer science department. The research topics are mainly in Software Architectures, Software Architectures Analysis, Component-based systems, Fault-tolerance, Middleware, Model checking, Formal Methods. In its research activity Patrizio collaborated with several industries such as Selex Marconi telecommunications, Ericsson, Siemens, TERMA, etc. Patrizio is chair of the ERCIM international workshop on Software Engineering for Resilient Systems (SERENE), is editor of a book: Software Engineering of Fault Tolerant Systems, and is reviewer of several workshops, conferences and journals.

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

Antonio Bucchiarone received his first master degree in Computer Science from the University of L'Aquila (Italy) in April 2003 and the second in Information Technologies from University of Pisa (Italy) in October 2005. He is finishing him PhD in Computer Science and Engineering at IMT School of Lucca (Italy) and since 2004 he is a collaborator of Formal Methods and Tools Group at ISTI-CNR of Pisa (Italy). Antonio's research interests are in the Computer Science, Software Engineering and Formal Methods Areas. His PhD thesis topic is on Dynamic Software Architecture-based development and analysis techniques of Global Computing Systems. In particular, his main research interests are in the field of Dynamic Software Architecture, Service Oriented Architecture, Formal Methods, Component-Based Systems, Model-Checking and Testing.

Andrea Polini received a first-class honors degree in Computer Science in 2000 from University of Pisa and a PhD in Computer Engineering in 2004 from Scuola Superiore Sant'Anna - Pisa. Currently, he is an Assistant Professor at the Computer Science Department of the University of Camerino. His research interests are mainly related to Verification and Testing of Complex Software Systems in particular in relation to Component Based Software Systems and Service Oriented Applications.