Dissertation

An architectural approach to the automatic composition and adaptation of software components

Massimo Tivoli

June 2005

Advisor: Prof. Paola Inverardi

© Massimo Tivoli, 2005. All rights reserved
ABSTRACT

Composition and adaptation of software components is an important issue in Component Based Software Engineering (CBSE). Building a system from reusable or Commercial-Off-The-Shelf (COTS) components introduces a set of problems, mainly related to compatibility and communication. On one hand, components may have incompatible interaction behavior (e.g., deadlocks, livelocks, incompatible method names, mismatch between the order in which messages are expected/exchanged and so on). This might require one to restrict the system’s behavior to avoid undesired interactions among components. On the other hand, it might be necessary to enhance the current communication protocol in order to deal with architectural updates (e.g., inserting, replacing or removing components), missing functionality or to improve quality attributes such as reliability. This might require one to augment the system’s behavior to introduce more sophisticated interactions among components. We address these problems by means of an architectural approach implemented in our SYNTHESIS tool which allows one for both detecting/avoiding incompatible interactions and enhancing the communication protocol by synthesizing a suitable coordinator. Taking into account the specification of the system to be assembled, the specification of the desired behaviors that the composed system has to exhibit and the specification of the protocol enhancements, SYNTHESIS automatically derives, in a compositional way, the glue code for the set of components forming the system. The synthesized glue code implements a software coordinator which avoids incompatible interactions (with respect to the specified desired behaviors) and provides a protocol-enhanced version of the composed system. Our SYNTHESIS tool can be applied to a variety of commonly used component-based middleware (e.g., Sun's Java Beans, J2EE, CORBA, Microsoft COM/DCOM, Microsoft .NET). So far, we validated and applied SYNTHESIS for assembling Microsoft COM/DCOM components. The code synthesized by the current version of SYNTHESIS refers to Microsoft Visual Studio with Active Template Library (ATL) as reference development platform. The approach presented in this thesis and implemented in the SYNTHESIS tool - which is also presented here - has to be considered as a support for developing component-based systems out of a set of already implemented heterogeneous components by ensuring the correct functioning of the assembled system at level of component interaction protocol. Moreover, it also provides the developers with a means for evolving/maintaining the assembled component-based system by allowing them to add new functionality and, in the same time, continue to maintain (or to be consistent with respect to) the behavior of the original system.
ACKNOWLEDGMENTS

First of all, I want to thank my advisor Prof. Paola Inverardi. She is great for many reasons. In particular, I really think that she is one of the most "beautiful minds" in the areas of Software Engineering and Computer Science. I think that everything I have learned is due to her ability to open my mind and to teach me what she knows. I hope to have learned what - or even only half of what - she expected from me. Moreover, although she is born to be a leader, she really appreciates all people that collaborate with her. She takes care of everybody that works under her supervision by maintaining a peaceful and familiar environment in the department. She really has a great heart beyond to be a very smart person.

Secondly, I thank my sweety Daniela. I refer to her as "the beautiful Daniela". She is the one, the best. I thank Daniela for her infinite patient in supporting me during each moment of my PhD program, from the most happy moments to the most ugly ones. Moreover, she had the ability to always give me the best suggestions in order to solve my existential doubts. She is also capable to make me feel closer to her although I am thousands kilometers far away. In some way, thanks to her psychological support, the work of this thesis is a her work too. Daniela, I love you!

Then, I thank my family (i.e., my mother Anna, my father Gianni, my brother Alberto, my aunt Maria Pia and my grandma Antonia) that has always supported both me and each choice I made without making my decisions more difficult than they really were. In particular, mom and Alberto have been able to make me look at myself as a person better than I really am. This made me able to really believe in myself. Daddy, you are a very strong person and I know you love me and you are proud of me although you never say that. Dad, thank you to have supported my studies by making a lot of renunciations. Aunt, thank you for your simplicity and innocence; you really are a rare person. Grandma, although since few months you are not in this world anymore, I always think about you and I talk to you very often and I can heart your valuable answers. This thesis is all dedicated to you! Thanks all, I am really lucky to have a family like this.

Finally, let me thank the Computer Science department stuff and all my colleagues that are also my best friends. All of you, guys, are marvelous. We are a great group; I am not only speaking about Computer Science; I am also speaking about life, real friendship and good feelings. I really hope to continue to work with all of you for the rest of my life. In particular, let me spent few (but very important and sincere) words about Marco (i.e., "il sindaco"). Marco, working with you has really improved my research and my life. You are really smart and "sweet-heart".

I also thank Prof. David Garlan since he supported me and he made me able to collaborate with him and his group when I have been at Carnegie Mellon University. I thank Prof. Ivica Crnkovic which gave me the possibility to be involved in a Post-doctoral program at Malardalen University.
# Table of Contents

Abstract i

Acknowledgments iii

Table of Contents v

1 Introduction 1

1.1 List of publications .................................................. 3

1.2 Funding acknowledgements ........................................ 5

2 Setting the context 7

2.1 A Classification of component mismatches ......................... 7

2.1.1 Signature-level adaptation ........................................ 8

2.1.2 Protocol-level adaptation ......................................... 9

2.1.3 Semantic-level adaptation ....................................... 11

2.2 Our context ............................................................. 12

3 Background 15

3.1 The reference architectural style .................................... 15

3.2 Calculus of Communicating Systems (CCS) ....................... 17

3.3 Configuration formalization ........................................ 18

3.4 Buchi automata ........................................................ 19

3.5 Microsoft COM/DCOM .............................................. 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1 Overview</td>
<td>19</td>
</tr>
<tr>
<td>3.5.2 The role of Windows registry</td>
<td>20</td>
</tr>
<tr>
<td>4 Automatic synthesis of “correct” and protocol-enhanced adaptors</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Automatic synthesis of deadlock-free adaptors</td>
<td>22</td>
</tr>
<tr>
<td>4.1.1 Method description</td>
<td>23</td>
</tr>
<tr>
<td>4.1.2 Method formalization</td>
<td>24</td>
</tr>
<tr>
<td>4.1.3 Deadlock detection and recovery</td>
<td>30</td>
</tr>
<tr>
<td>4.1.4 Correctness and completeness</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Dealing with normalization</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Dealing with signature-level mismatches</td>
<td>42</td>
</tr>
<tr>
<td>4.3.1 Method description</td>
<td>43</td>
</tr>
<tr>
<td>4.3.2 Method formalization</td>
<td>43</td>
</tr>
<tr>
<td>4.4 Automatic synthesis of failure-free adaptors</td>
<td>46</td>
</tr>
<tr>
<td>4.4.1 Method description</td>
<td>46</td>
</tr>
<tr>
<td>4.4.2 Method formalization</td>
<td>46</td>
</tr>
<tr>
<td>4.4.3 Correctness and completeness</td>
<td>50</td>
</tr>
<tr>
<td>4.4.4 Failure-free coordinator synthesis algorithm</td>
<td>53</td>
</tr>
<tr>
<td>4.5 Automatic synthesis of protocol-enhanced adaptors</td>
<td>53</td>
</tr>
<tr>
<td>4.5.1 Method description</td>
<td>54</td>
</tr>
<tr>
<td>4.5.2 Method formalization</td>
<td>56</td>
</tr>
<tr>
<td>4.5.3 Checking enhancement consistency</td>
<td>61</td>
</tr>
<tr>
<td>5 SYNTHESES: a tool for synthesizing “correct” and protocol-enhanced adaptors</td>
<td>65</td>
</tr>
<tr>
<td>5.1 SYNTHESES’s method description</td>
<td>65</td>
</tr>
<tr>
<td>5.1.1 First phase: initial coordinator synthesis</td>
<td>65</td>
</tr>
<tr>
<td>5.1.2 Second phase: from the initial coordinator to the protocol-enhanced coordinator</td>
<td>66</td>
</tr>
<tr>
<td>5.2 The SYNTHESES tool</td>
<td>68</td>
</tr>
<tr>
<td>5.2.1 First phase</td>
<td>69</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Module A: component interface parser</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Module B: bMSC and HMSC specification of the CFA</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Module C: component assumption generator</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Module D: büchi automata specification of the coordination policies</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Module E: builder of the initial coordinator model</td>
</tr>
<tr>
<td>5.2.7</td>
<td>Module F: generator of the initial coordinator actual code</td>
</tr>
<tr>
<td>5.2.8</td>
<td>Second phase</td>
</tr>
<tr>
<td>5.2.9</td>
<td>Module G: wrapper assumption generator</td>
</tr>
<tr>
<td>6</td>
<td>Case studies</td>
</tr>
<tr>
<td>6.1</td>
<td>Automatic coordinator synthesis for a COTS group-ware system</td>
</tr>
<tr>
<td>6.1.1</td>
<td>CW systems: state of the art</td>
</tr>
<tr>
<td>6.1.2</td>
<td>A semi-synchronous CW system</td>
</tr>
<tr>
<td>6.2</td>
<td>Automatic component assembly for a Product Data Management cooperative system</td>
</tr>
<tr>
<td>6.2.1</td>
<td>ThinkTeam architecture</td>
</tr>
<tr>
<td>6.2.2</td>
<td>ThinkTeam/Application integration schema</td>
</tr>
<tr>
<td>6.2.3</td>
<td>SYNTHESIS for integrating ThinkTeam and CAD</td>
</tr>
<tr>
<td>6.3</td>
<td>Automatic synthesis of adaptors for a cooling water pipe system</td>
</tr>
<tr>
<td>6.3.1</td>
<td>SYNTHESIS at work</td>
</tr>
<tr>
<td>7</td>
<td>Related Work</td>
</tr>
<tr>
<td>7.1</td>
<td>Supervisory control theory for synthesizing schedulers</td>
</tr>
<tr>
<td>7.2</td>
<td>Interface theories for converter synthesis</td>
</tr>
<tr>
<td>7.3</td>
<td>Compositional model-checking of software components</td>
</tr>
<tr>
<td>7.4</td>
<td>Protocol adaptor synthesis</td>
</tr>
<tr>
<td>7.5</td>
<td>Wrapper formalization</td>
</tr>
<tr>
<td>7.6</td>
<td>Behavioral types and component adaptation</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
</tr>
<tr>
<td>8.1</td>
<td>Future work</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

References 119
CHAPTER 1

INTRODUCTION

Component composition and adaptation is an important issue in Component-Based Software Engineering (CBSE). Actually, while composing/assembling reusable or Commercial-Off-The-Shelf (COTS) components is a standard practice in modern software development, constraining the composition of components to respect a particular software architecture and some form of adaptation are often unavoidable to achieve the desired interoperability. The integration of legacy systems is the obvious example of unavoidable heterogeneity that needs to be faced. The impossibility of modifying large client applications is another major reason for the need of using some form of adaptation to integrate software. The need for adaptation is even stronger in the emerging area of service-oriented computing where interaction mismatches must be solved to ensure the correct connectivity among different web services.

One of the main problems in the area of component composition and adaptation is related to the ability to establish properties of the assembled system by assuming only a limited knowledge of the single component properties. Our answer to this problem is a software architecture-based approach in which the software architecture imposed on the assembly allows one for detection and recovery of black-box integration anomalies and for adding extra-functionality. Notably, in the context of component-based concurrent systems, COTS or (in general) third-party component integration may cause deadlocks or other software anomalies within the system \([10, 35, 39, 66]\). Building a system from reusable or COTS \([66]\) components introduces a set of problems mainly related to communication and compatibility. Many of these problems arise because of the nature of COTS components. They, usually, are black-box and developers, reusing them, have no method of looking inside the box. This limitation is coupled with an insufficient behavioral specification of the component which, therefore, does not permit one to understand the component interaction behavior in a multi-component setting.

Often, components may have incompatible interaction behavior. Component composition can result in architectural mismatches when trying to integrate components with incompatible interaction behavior \([28]\). This might require one to restrict the system’s behavior to avoid undesired interactions among components. For example, one may restrict the system behavior to the set of deadlock-free interactions or, in general, to a specified set of desired interactions. Moreover, it might be necessary to enhance the current communication protocol. This might require one to augment the system’s behavior to introduce more sophisticated interactions among components. These enhancements might be needed to achieve dependability, to add extra-functionality and/or to deal properly with system’s architecture updates (e.g., aggregating, inserting, replacing and removing components).

Thus, if we want to assure that a component-based system validates specified behavioral properties, we must take into account the components’ interaction behavior. In this context, the notion of software architecture assumes a key role since it represents the reference skeleton used to compose components and let them interact. In the software architecture domain, the interaction among the components is represented by the notion of software connector.

In this thesis, we formalize an architectural approach to the automatic composition and adaptation of software components and discuss its implementation and validation. Our aim is to analyze and fix dynamic behavioral problems that can arise from component composition. Moreover, we also want to be able to
enhance the current communication protocol in order to augment the system’s behavior by adding more sophisticated interactions among components.

We propose 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 over which the components will communicate [60]. We will consider the special case of a generic layered architecture in which components can request services of components below them, and notify components above them. We compose a system in such a way that it is possible to check whether and why the system exhibits integration failures. We derive, in an automatic way, from the COTS (black-box) components, the code that implements a new component that has to be inserted in the composed system. This new component implements a software coordinator. The coordinator mediates the interaction among components in order to avoid possible integration failures.

For the aims of this work, we assume that some specification of the externally “observable” behavior of each component (forming the assembled system) is available in the form of a Calculus of Communicating Systems (CCS) process [50]. Thus, in the reminder of the thesis, when we consider a set of black-box components, we will associate to each black-box component a CCS process representing the externally “observable” behavior of the component. This is the behavior of the component in terms of the messages exchanged with its environment. It is possible to automatically derive these CCS processes by assuming a partial specification of the assembled system. This partial specification is given in the form of basic Message Sequence Chart (bMSC) and High Level MSC (HMSC) specifications [3, 70, 71, 72]. The automatic derivation of the behavioral specification for each component can be performed by applying our implementation of the algorithm described in [70, 71, 72]. Moreover, we assume a specification of the desired behaviors that the composed system should exhibit. Under these two assumptions we have developed a framework that automatically derives the assembly code for a set of components. This code, which represents the initial coordinator’s actual code, is derived in order to obtain a failure-free system (i.e., a deadlock-free system that performs only the specified desired interactions).

Subsequently, taking into account the specification of possible protocol enhancements, our framework automatically derives, in a compositional way, the enhanced glue code for the set of components. The enhanced glue code implements a software coordinator that avoids not only incompatible interactions but also provides a protocol-enhanced version of the composed system. More precisely, this enhanced coordinator is defined as a composition of new coordinators and components that are assembled with the initial coordinator in order to enhance its protocol. Each new component represents a wrapper component. A wrapper intercepts the interactions defined by the initial coordinator’s protocol in order to apply the specified enhancements without modifying the initial coordinator and the components in the system. The new coordinators are needed to assemble the wrappers, the initial coordinator and the rest of the components forming the composed system in such a way that the whole assembly is deadlock-free. It is worthwhile noticing that, in this way, our approach is compositional. That is, we can treat the enhanced coordinator as a new composite initial coordinator and enforce new desired interactions as well as apply new enhancements. This allows us to perform a protocol enhancement as a composition of modular protocol enhancements by improving on the reusability of the synthesized glue code. When we apply the specified protocol enhancements to produce the enhanced coordinator, we might re-introduce incompatible interactions avoided by the initial coordinator. That is, the enhancements do not hold the desired behaviors specified to produce the initial coordinator. In this thesis, we also show how to check if a protocol enhancement retains the desired behaviors enforced through the initial coordinator. This is done, in a compositional way, by using an assume-guarantee technique [21].

We have implemented the approach in our SYNTHESIS tool, which provides support for both detecting/avoiding incompatible interactions and for enhancing the communication protocol by synthesizing a suitable coordinator. This coordinator represents the glue code. It is worth mentioning that SYNTHESIS can be used either to derive the actual code implementing a coordinator or to only derive its behavioral model. When the final goal of the SYNTHESIS’s user is to derive the actual code, the underlining ap-

\footnote{This is needed to achieve composability in both specifying the enhancements and implementing them.}
proach depends by the particular development platform that is chosen to implement that code. Otherwise, the approach does not depend by any particular platform. That is, the automatic synthesis of the model of a coordinator is independent from the automatic synthesis of its actual code. Thus, SYNTHESIS has to only partially refer to one or more particular coordinator development platforms. Our SYNTHESIS tool can be applied to a variety of commonly used component-based middleware (e.g., Sun's Java Beans, J2EE, CORBA, Microsoft COM/DCOM, Microsoft .NET). So far, we have validated and applied SYNTHESIS for assembling only Microsoft COM/DCOM components. The coordinator code synthesized by the current version of SYNTHESIS refers to Microsoft Visual Studio with Active Template Library (ATL) as reference development platform.

The architectural approach presented in this thesis and implemented in the SYNTHESIS tool - which is also presented here - has to be considered as a support for developing component-based systems out of a set of already implemented heterogeneous components by ensuring the correct functioning of the assembled system at level of component interaction protocol. Moreover, it also provides the developers with a mean for evolving/maintaining the assembled component-based system by allowing them to add new functionality and, in the same time, continue to maintain (or to be consistent with respect to) the behavior of the original system.

The reminder of this thesis is organized as follows. In Chapter 2, by means of a classification of component mismatches, we precisely set the context of our approach. In Chapter 3 we provide both formal/theoretical and practical/technological background notions needed to understand the formalization and the implementation of our approach. In Chapter 4 we formalize our approach by focusing on adaptation at the level of syntactic rules of interaction and of protocol interaction among components. In this chapter we also give a formal proof of correctness and completeness of the approach. In Chapter 5 we present the overall structure of the current version of our tool which is called SYNTHESIS. It implements and automatizes the whole approach formalized in Chapter 4. In Chapter 6 we discuss three case studies that we used to validate three different prototypal versions of our SYNTHESIS tool. These versions go from the primordial version to the latest one. Chapter 8 concludes and discusses future work.

1.1 LIST OF PUBLICATIONS

This thesis is based on the following list of publications and technical reports:

- **Books and chapters:**
  
  Editors: Marco Bernardo and Paola Inverardi. 
  Publisher: Springer. 
  Volume: LNCS 2804. 
  Year: 2003.

- **International journals:**
  
  To appear on LObject journal.

  - *Deadlock-free software architectures for COM/DCOM Applications*, P. Inverardi and M. Tivoli. 
International workshops and conferences:


1.2 Funding acknowledgements

This research was funded in part by the National Science Foundation under Grant No. CCR-0113810. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.
In this chapter we propose a classification of component mismatches that are specifically related to interoperability issues. This classification allows us to define a component interface’s model capturing different aspects/dimensions of a component interface. Within our model, a component interface is seen as a set of rules of interaction specifying the assumptions made by a component about the interaction with its environment. Based on our classification of component mismatches, we review component adaptation techniques that exist in the literature and classify them by referring to the different dimensions of our component interface model. We report this classification of component adaptation techniques in order to precisely set the context our approach can be applied to. We precisely identify “where” (i.e., at level of component interface’s signature, protocol or semantics) and “when” (i.e., at design- or run-time) our approach can be applied. In other words, we characterize what kinds of component mismatch our approach is able to deal with and how/when our approach generates a particular adaptation strategy.

2.1 A CLASSIFICATION OF COMPONENT MISMATCHES

In the literature there are few classification approaches of component mismatches that are specific to interoperability issues. By referring to [74] and an its more abstract and complete extension [8] (including also aspects related to Quality of Service (QoS)), we arrange the concepts the latter classification is based on in order to give a new classification. The component mismatches classification schema, we propose in this chapter, is based on three horizontal dimensions related to three aspects of a component interface and one orthogonal dimension related to QoS aspects. In Figure 2.1 we show our component mismatches classification schema.

![Figure 2.1: Component mismatches classification schema](image)

The component interface model underlining our classification schema, is based on three layers that represent different views of the functional and conceptual aspects of a component’s interface and one layer that
encapsulates non-functional aspects. By referring to Figure 2.1, we denote these layers as Syntax, Protocols, Conceptual Semantics and Quality respectively. It is worthwhile noticing that the Quality layer is orthogonal to the other layers. That is: i) applying an adaptation strategy for a class of component mismatches which is contained in any horizontal layer might have a meaningful influence on the QoS properties [9, 36, 37] of the adapted component and hence ii) adaptation of QoS properties of a component may be needed at an interface’s “signature”, “interaction protocol” or “concepts meaning” level.

The Quality layer is concerned with non-functional incompatibilities due to different assumptions about the non-functional characteristics of components. By referring to [8], these non-functional characteristics are split in structural quality characteristics (i.e., QoS at Syntax level), assumptions on design aspects (i.e., QoS at Protocols level) and dynamic quality characteristics (i.e., QoS at Conceptual semantics level). Structural quality characteristics relate to component usability, maintainability or portability. In contrast, assumptions on design aspects and dynamic quality characteristics interfere with the interaction of components. They relate to a component’s security, persistency, reliability and performance.

In the following sub-sections, we look at each horizontal layer in more detail.

2.1.1 SIGNATURE-LEVEL ADAPTATION

The Syntax layer is related to interface’s signature mismatches [76]. In other words it is concerned with mismatches at the level of syntax rules of interaction. For instance, these mismatches are related to the signature of the methods exported by a component interface. They can be listed as follows:

- incompatible method name,
- non-existent method,
- incompatible argument name,
- incompatible order of arguments,
- incompatible argument type,
- different thrown exceptions.

Thus, in general, the Syntax layer contains mismatches between the required (provided) interface signature of a component and the provided (required) interface signature of a different component.

To better explain the nature of the component mismatches contained in the Syntax layer, the following example is introduced. Let us suppose that we are involved in the component-based development of a product data management (PDM) software system supporting cooperative work and based on a centralized repository that we call the “vault”. The vault stores all the documents related to the products and the working groups. The PDM system should enable its users (i) to work either in a cooperative or in a stand-alone mode, (ii) to access to some information contained in the vault either in reading or writing mode and (iii) to store new/updated information in the vault. For the sake of simplicity, we limit our explanatory example to that functionality.

By taking into account these requirements, the signatures identified for the required methods are established (see Figure 2.2). These signatures are compared with the signatures of the (COTS) component that has been selected to match the identified requirements (see Figure 2.3).

It is worthwhile noticing that the signatures of the single methods mismatch without adaptation. For example, note the different names and the different parameter types of the methods Checkout (or OpenDoc as
2.1 A Classification of component mismatches

HRESULT SetWorkingMode([in] WMID wModeID);
[out,retval] DocumentOBJ* docRef);
[out,retval] BOOL* result);

Figure 2.2: IDL notation of the Required interface

HRESULT StartCooperativeMode();
HRESULT StopCooperativeMode();
HRESULT CheckOut([in] UID key, [out,retval] Document* pDoc);
HRESULT CheckIn([in] UID key, [in] Document* pDoc, [out,retval] BOOL* ack);
HRESULT SaveDoc([in] CHAR* filePath, [in] Document* pDoc, [out,retval] BOOL* ack);

Figure 2.3: IDL notation of the Provided interface

well) and Open. Let us assume that the methods do the same. More precisely, assume that (i) CheckOut
does the same thing with respect to Open when the user works in a cooperative working mode; (ii) Open-
Doc does the same thing with respect to Open when the user works in a stand-alone working mode. Thus,
those problems can be solved by using an adaptor component that suitably maps methods and parameters of
the required interface to the corresponding methods and parameters of the provided interface. For example,
by assuming that WMID is a number denoting either a cooperative working mode (i.e., 0) or a stand-
alone working mode (i.e., any integer number different from 0), the method’s call SetWorkingMode(0) is
equivalent to StartCooperativeMode().

By only looking at method names, it might be possible to solve this kind of mismatch by applying ap-
proaches that automatically build the adaptor by taking into account a suitable component interface spec-
ification. On the other hand, in general, human-intervention-based approaches (and hence semi-automatic
ones) are required when the goal is to bridge each type of mismatch at the level of an interface’s signature.

2.1.2 Protocol-level adaptation

The Protocols layer is concerned with mismatches at the level of functional/dynamic aspects of interac-
tion [59, 75] such as:

- a mismatch between the order in which messages are expected/exchanged,
- a message comes too early or too late (timing mismatch),
- events handled incorrectly with respect to method’s pre- and post-condition.

Thus, in general, the Protocols layer contains mismatches between the required (provided) interaction/-
coordination protocol of a component and the provided (required) interaction/coordination protocol of a
different component.

By continuing our explanatory example introduced in Section 2.1.1, we now focus on the interaction/co-
ordination protocols that are defined by the required (see Figure 2.2) and the provided (see Figure 2.3)
interface, respectively. In the left-hand side of Figure 2.4 we show the interaction protocol expected by a component supporting the required interface. In the right-hand side of Figure 2.4 we show the interaction protocol expected by the component supporting the provided interface.

The protocol defined by the required interface specifies that every method can be called at any time. On the other hand, the protocol defined by the provided interface makes a distinction between the methods used to access the vault, and it depends on the selected working mode.

Note that by means of a component adaptor it is possible to bridge the interaction protocol incompatibilities discussed in this section. For example, without looking at the method parameters, one might build an adaptor that maps the following method call traces (where the notation used to denote a trace that represents a sequence $a_1, \ldots, a_n$ of $n$ actions is $< a_1, \ldots, a_n >$):

- $<\text{Open}>$
- $<\text{Save}>$
- $<\text{SetWorkingMode}(0)>$
- $<\text{SetWorkingMode}(n)>$ with $n \neq 0$
- $<\text{SetWorkingMode}(n), \text{Open}>$ with $n \neq 0$
- $<\text{SetWorkingMode}(0), \text{Open}>$
- $<\text{SetWorkingMode}(n), \text{Save}>$ with $n \neq 0$
- $<\text{SetWorkingMode}(0), \text{Save}>$
- $<\text{Open, SetWorkingMode}(0)>$
- $<\text{Open, SetWorkingMode}(n)>$ with $n \neq 0$
- $<\text{Save, SetWorkingMode}(0)>$
- $<\text{Save, SetWorkingMode}(n)>$ with $n \neq 0$
- etc.
2.1 A Classification of component mismatches

to the following ones, respectively:

- `<OpenDoc>`
- `<SaveDoc>`
- `<StartCooperativeMode>`
- `<StopCooperativeMode>`
- `<StopCooperativeMode, OpenDoc>`
- `<StartCooperativeMode, CheckOut>`
- `<StopCooperativeMode, SaveDoc>`
- `<StartCooperativeMode, CheckIn>`
- `<OpenDoc, StartCooperativeMode>`
- `<OpenDoc>`
- `<SaveDoc, StartCooperativeMode>`
- `<SaveDoc>`
- etc.

The assumption is that the default working mode is the stand-alone one. Moreover, besides performing the above mapping, the adaptor has to refuse a call to the method `CheckOut` if the method `StartCooperativeMode` has not been called. Calls to the method `CheckIn` are controlled in the same way. Analogously, the adaptor has to refuse a call to the method `OpenDoc` if the method `StopCooperativeMode` has not been called. The same is true for calls to `SaveDoc`.

By enriching the specification of a component interface in order to capture also component interaction behavior, it might be possible to automatically derive information about components that goes from the actual component behavior to its assumptions on the environment. This, in turn, might allow one to automatically build an adaptor by taking into account that enriched specification.

2.1.3 Semantic-level adaptation

The Conceptual Semantics layer is concerned with "meaning" incompatibilities [34] due to a different understanding of the domain underlying the execution environment expected/assumed by a component. The following is a list of examples of this kind of incompatibility:

- incompatible data conversion (i.e., representation, pointer reference),
- incompatible data structure (i.e., a value represents a parameter in another representation than expected - e.g., the order of records in a XML file is not as expected),
- incompatible meaning of data (i.e., a parameter is represented in a different way - e.g., coordinates are represented in polar-coordinates versus cartesian coordinates).
By continuing our explanatory example, the required interface has a notion of the vault which is more restrictive than the one that the provided interface assumes. In fact, the required interface looks at the vault as a database based on a relational model. On the other hand, the provided interface deals with the vault by using a more abstract notion of centralized/shared repository. For example, it might be a database, a directory in the file system or a structured file.

Commonly, this kind of mismatch can be partially solved by enriching the component interface specification with meta-information that allows one to better understand which is the execution domain expected by the component. Unfortunately, in general, this kind of mismatch cannot be bridged by using automatic approaches. For example, in a black-box (or COTS) component setting, the meta-information needed to completely understand what is the expected execution domain is (in most cases) totally missing.

![Figure 2.5: Design-time adaptation](image)

### 2.2 Our Context

The architectural approach to the automatic composition and adaptation of software components developed in this thesis is focused on automatically generating an adaptation strategy for integrating a set of incompatible software components. In general, by referring to "when" a suitable adaptation strategy is generated, there are two common approaches to deal with the classes of component mismatch discussed in Section 2.1. An adaptation strategy can be generated either statically (i.e., at design-time) or dynamically (i.e., at run-time).

When an adaptation strategy is generated at design-time, a common technique is to statically build an adaptor for the set of mismatching components. The adaptor is built by taking into account a suitable specification of the component interfaces. Then, the adaptor has to be interposed between the components forming the system in order to bridge their run-time incompatibilities. In Figure 2.5, we show an example
of two mismatched components $C1$ and $C2$, whose incompatibilities are bridged by means of a suitable adaptor.

The feasibility of design-time adaptation strongly depends on the expressiveness of the reference interface model. In fact, it is well known (and it is also pointed out by the classification of component mismatches discussed in Section 2.1) that interface models capturing protocols make possible detecting and recovering more mismatches than ones that only model signature.

On the other hand, when adaptation is established at run-time, typically, it is done through reconfiguration. In this context, the adaptation strategy is dynamically established/generated on contextual basis (e.g., moving a mobile device into a different context) or on the basis of policies telling how to reconfigure under certain conditions.

By referring to Section 2.1 and to the above discussion about “when” generating an adaptation strategy, we finally set the context our approach can be applied to. The approach that we formalize in Chapter 4 and that we validate in Chapters 5 and 6, should be considered a framework supporting the automatic composition and adaptation of black-box software components at design-time. Moreover, by dealing with black-box components and in order to make our approach as automatic as possible, it is for dealing with both mismatches at level of interface’s protocol and a restrict class of mismatches at level of interface’s signature. By referring to the component mismatches classification schema of Figure 2.1, in Figure 2.6 we graphically point out (by means of grey boxes) what classes of mismatches our approach is able to fix.

![Figure 2.6: Classes of component mismatches that can be fixed by our approach](image-url)
In this chapter we provide the background needed to understand the approach formalized in Chapter 4 and validated by means of Chapters 5 and 6. The presentation of these background notions goes from formal/theoretical notions to practical/technological ones.

Firstly, in Section 3.1, we define the reference architectural style adopted within our approach. This style represents the starting point of our work. Through this style, we impose constraints, on the architecture of the system that must be assembled, that allows us to automatically and incrementally derive - from a component specification - different models of the interaction behavior of the environment assumed by a component. Each of these models represents a partial view of the adaptor component. It is partial because it reflects only the expectations of a single component. By unifying all these partial views, our approach can automatically synthesizing a model of the interaction behavior of the adaptor. Secondly, in Section 3.2, we summarize the relevant definitions regarding the specification language (i.e., Calculus of Communicating Systems (CCS)) that is used - within our approach - to specify/model the externally “observable” behavior of a component. Then, in Section 3.3, we refer to two possible ways of building systems out of a set of already implemented components. One way is concerned with a coordinator-free architecture (CFA) in which no adaptor (i.e., the coordinator) is introduced in the composed system. That is, a CFA system is only formed by acquired third-party components and connectors. The CFA models the composed system as it is given in input to our method. The other one is concerned with a coordinator-based architecture (CBA) in which an adaptor (namely the coordinator) is introduced in the composed system. The coordinator mediates the interactions among the components - forming the system - in order to avoid possible mismatches. The CBA models the composed system as it is given in output by our method. Moreover, in Section 3.4, we introduce the formalism (i.e., Büchi Automata) that it is used to specify/model the set of desired behaviors that the composed system must exhibit through the introduction of the coordinator in the CBA that must be derived. Finally, in Section 3.5, we briefly describe the relevant notions of the application domain (i.e., Microsoft COM/DCOM component-based systems) that has been considered to validate our approach in a real-scale context.

### 3.1 The Reference Architectural Style

The starting point for our work is the definition of an architectural style [60] that allows us to use a formal model of the system’s architecture representing the components to be integrated, the connectors over which the components will communicate and a set of constrains dictating how components may be legally composed. Specifically, according to [7, 60], we define an architectural style as a set of constraints on a software architecture that identify a class of architectures with similar features. A software architectural style is determined by the following:

- a set of component types that perform some function at runtime;
- a topological layout of these components indicating their runtime interrelationships;
a set of syntactic/topological constraints;
a set of connectors that mediate communication, coordination or cooperation among components.

Under the architectural style classification described in [60], the reference architectural style we use in this thesis belongs to layered systems.

We assume each component has a top and bottom interface. Connectors between components are synchronous communication channels defining a top and bottom interface too. The top (bottom) interface of a component may be connected to the bottom (top) interface of one or more connectors.

Components communicate by passing two types of messages: notifications and requests. A notification is sent downward, while a request is sent upward. We also distinguish between two kinds of components: (i) functional components and (ii) coordinators. Functional components implement the system’s functionality, and they are the primary computational constituents of a system (typically implemented as COTS components). Coordinators, on the other hand, simply route messages and each input they receive is strictly followed by a corresponding output. We make this distinction in order to clearly separate components that are responsible for the functional behavior of a system and components that are introduced to aid the integration/communication behavior.

Within this architectural style, we will refer to a system as a Coordinator-Free Architecture (CFA) if it is defined without any coordinators. Conversely, a system in which coordinators appear is termed Coordinator-Based Architecture (CBA) and is defined as a set of functional components directly connected to one or more coordinators, through connectors, in a synchronous way.

![CFA system](image1)

![CBA system](image2)

**Figure 3.1:** A sample of a CFA and of a CBA

Figure 3.1 illustrates a CFA (left-hand side) and a CBA (right-hand side). $C_1, ..., C_7$ are functional components; $K$ is a coordinator. Each line between two components is a connector.

The main characteristics of our style are the following:

- synchronous messages;
- coordinators can directly communicate (i.e., a connector between two coordinators is allowed);
- coordinators are only routing devices, without any filtering policies;
- strictly sequential input-output behavior of the coordinators.

We have introduced the first constraint because we are partially focused on concurrency conflicts such as deadlock that are typical within synchronous systems. This is not a limitation because it is well known that with the introduction of a buffer we can always simulate an asynchronous system by a synchronous one. We have introduced the second and the third constraint because in order to apply our methodology without human intervention we have to make assumptions on the behavior of the coordinator and on the
structure of the environment of each coordinator. The last constraint introduces an input/output structure in
the coordinator. The aim of this constraint is to make the coordinator similar to a reactive component.

Although CBA style is a generic layered style, in this chapter, for the sake of presentation, we only focus
on single-layer systems. We provide the foundations to deal with multi layered systems in Section 4.2. For
the sake of simplicity, in the reminder of the thesis we will use the term component to identify a functional
component.

3.2 C A L C U L U S O F C O M M U N I C A T I N G S Y S T E M S (CCS)

We now summarize the relevant definitions regarding CCS, and refer to [50] for more details. The CCS
syntax our approach is based on is the following:

\[ p ::= \mu.p \mid \text{nil} \mid p + p \mid p|p \mid p\setminus A \mid x \mid p[f] \]

Terms generated by \( p \) (Terms) are called process terms (also called processes); \( x \) ranges over a set
\( \{X, Y, \ldots\} \), of process variables. A process variable is defined by a process definition
\( x \text{def}= p \), (\( p \) is called the expansion of \( x \)). As usual, there is a finite set of visible actions \( Vis = \{?a, !a, ?b, !b, \ldots\} \) over which \( \alpha \) ranges, while \( \mu, \nu \) range over \( \text{Act} = Vis \cup \{\tau\} \), where \( \tau \) denotes the so-called internal action. A visible action \( ?a \) denotes an input action ‘\( a \)’, while \( !a \) denotes an output action ‘\( a \)’. We denote by \( \bar{\alpha} \) the action complement: if \( \alpha = ?a \), then \( \bar{\alpha} = !a \), while if \( \alpha = !a \), then \( \bar{\alpha} = ?a \). By \( \text{nil} \) we denote the empty
process. The operators to build process terms are prefixing (\( \mu.p \)), summation (\( p + p \)), parallel composition (\( p|p \)), restriction (\( p\setminus A \)) and relabeling (\( p[f] \)), where \( A \subseteq Vis \) and \( f : Vis \rightarrow Vis \).

An operational semantics \( OP \) is a set of inference rules defining a relation \( D \subseteq \text{Terms} \times \text{Act} \times \text{Terms} \).

The rules are defined as follows:

\[
\begin{align*}
\text{Act} & \quad \alpha.p_{\sim} \alpha.p' & \quad \text{Synch} & \quad \frac{p_{\sim}^\alpha.p'Q_{\sim}Q'}{p|Q_{\sim}^\alpha.p'|Q'} \\
\text{Sum} & \quad \frac{p_{\sim}^\alpha.p'}{p + Q_{\sim}^\alpha.p'} & \quad \text{Rel} & \quad \frac{p_{\sim}^\alpha.p'}{p[f]} \\
\text{Comp} & \quad \frac{p_{\sim}^\alpha.p'}{p|Q_{\sim}^\alpha.p'|Q} & \quad \text{Res} & \quad \frac{p_{\sim}^\alpha.p'_{\sim}A_{\sim}p'}{p_{\sim}^\alpha.p'_{\sim}A_{\sim}p'} \\
\text{Con} & \quad \frac{p_{\sim}^\alpha.p'_{\sim}A_{\sim}p'}{p_{\sim}^\alpha.p'}
\end{align*}
\]

The rules \( \text{Sum} \) and \( \text{Comp} \) have a symmetric version which is omitted.

A Labeled Transition System (LTS) (or simply transition system) \( TS \) is a quadruple \((S, T, D, s_0)\), where \( S \)
is a set of states, \( T \) is a set of transition labels, \( s_0 \in S \) is the initial state, and \( D \subseteq S \times T \times S \). A transition
system is finite if \( D \) is finite.

A finite computation of a transition system is a sequence \( \mu_1, \mu_2, \ldots, \mu_n \) of labels such that:

\[ s_0 \overset{\mu_1}{\Rightarrow} s_1 \overset{\mu_2}{\Rightarrow} \cdots \overset{\mu_n}{\Rightarrow} s_n \]
Given a term $p$ (and a set of process variable definitions), and an operational semantics $OP$, $OP(p)$ is the transition system $(Terms, Act, D, p)$, where $D$ is the relation defined by $OP$. For example, $SOS(p)$ is the transition system defined by the SOS semantics for the term $p$. CCS can be used to define a wide class of systems, that ranges from Turing machines to finite systems [67]; therefore, in general, CCS terms cannot be represented as finite state systems. For our purposes, we will in the following assume that all the systems we will deal with are finite state and hence (under that assumption) there will be a strict correspondence between the CCS description of a component/system and its LTS description.

3.3 Configuration Formalization

We assume that a behavioral specification for each component in the form of a CCS process is provided. In Section 3.2 we mention that (under suitable assumptions) a CCS term corresponds to a LTS. Our model describes components in terms of their input and output actions using LTSs. Since we are assuming a synchronous communication, input and outputs are considered to be blocking actions, thus we will work with the (synchronous) parallel composition of LTS. Thus, in this thesis, component behavior will be described as CCS processes and system configuration will be specified using the parallel composition and restriction operators. As LTSs of all examples used in this thesis are shown, knowledge of CCS is not critical to follow the main ideas of our approach.

To define the behavior of a composition of components, we simply place in parallel the LTS descriptions of those components, hiding the actions to force synchronization. This gives a CFA for a set of components.

We can also produce a corresponding CBA for these components with equivalent behavior by automatically deriving and interposing a “no-op” coordinator between communicating components. That coordinator does nothing (at this point), but simply pass events between communicating components. As we will see later the “no-op” coordinator will play a key role both in restricting the set of system behaviors to a subset of deadlock-free and desired ones and in augmenting the system behavior in order to introduce more sophisticated interactions among components. The “no-op” coordinator is automatically derived by performing the algorithm described in Section 4.1.2.

**Definition 1 (CFA)** A Coordinator-Free Architecture (CFA) is a set of functional components directly connected, through connectors, in a synchronous way.

**Definition 2 (CBA)** A Coordinator-Based Architecture (CBA) is a set of functional components directly connected to one or more coordinators, through connectors, in a synchronous way.

Given $n$ components (i.e., $n$ component’s CCS description) $C_1, ..., C_n$, by means of CCS we will formalize a CFA system as follows:

$$CFA \equiv (C_1 | C_2 | ... | C_n) \cup_{i=1}^{n} \text{Act}_{C_i}$$

where for all $i = 1, ..., n$, $\text{Act}_{C_i}$ is the action set of the CCS process $C_i$. The corresponding CBA system that our method will generate can be modeled as:

$$CBA \equiv (C_1[f_1] | C_2[f_2] | ... | C_n[f_n] | K) \cup_{i=1}^{n} \text{Act}_{C_i}[f_i]$$

where for all $i = 1, ..., n$, $\text{Act}_{C_i}$ is the action set of the CCS process $C_i$, $f_i$ is a relabeling function such that $f_i(\alpha) = \alpha$ for all $\alpha \in \text{Act}_{C_i}$, and $K$ is the CSS process corresponding to the coordinator which is automatically synthesized by performing the algorithms described in Chapter 4.
3.4 Buchi automata

Referring to [21], a Büchi Automaton $B$ is a 5-tuple $< S, A, \Delta, q_0, F >$, where $S$ is a finite set of states, $A$ is a set of actions, $\Delta \subseteq S \times A \times S$ is a set of transitions, $q_0 \in S$ is the initial state, and $F \subseteq S$ is a set of accepting states. An execution path of $B$ on an infinite word $w = a_0a_1...$ over $A$ is an infinite sequence $\sigma = q_0q_1...$ of elements of $S$, where $(q_i, a_i, q_{i+1}) \in \Delta$, $\forall i \geq 0$. An execution path of $B$ is accepting if it contains some accepting state of $B$ an infinite number of times. $B$ accepts a word $w$ if there exists an accepting execution path of $B$ on $w$. The language accepted by $B$ is denoted as $L(B)$ and it is the set of infinite words $w$ such that $B$ accepts $w$.

3.5 Microsoft COM/DCOM

Although our framework can be applied to a variety of component-based middleware commonly used in the practice of component-based systems development (i.e., COM/DCOM/COM+, .NET, CORBA, J2EE), it has been validated in the context of COM/DCOM applications [11, 18, 19, 66]. In this section, we briefly recall the basic notions of interest (with respect to our framework) of our application domain (i.e., COM/DCOM applications).

3.5.1 Overview

A COM/DCOM application is based on a client/server architecture. A server is a component providing services to its clients.

A COM server is represented by an object library called Type Library. A type library is a container for a set of server objects defined by a server component. Each server object (in a type library) is an instance of a COM class defined by the code implementing the server component.

A client of a component is any program that can execute the component code. To allow clients to access its functions, a COM/DCOM server implements one or more interfaces by defining a COM class for each interface. COM/DCOM supports multiple interfaces of a component.

An interface is a binary structure in the address space of a component, whose layout is defined as a table of pointers to functions. In order to interact with a component, a client uses a reference (also called pointer) to a component interface. The specification of all functions that can be called through an interface is defined by the type of that interface. A type may inherit from another type by extending its list of function specifications. All interface types are organized in a single inheritance hierarchy with a special type IUnknown as root. Typically, COM/DCOM takes Microsoft’s version of the DCE/IDL (MIDL) as the preferred notation to define an interface type. The DCE/IDL and MIDL are two different languages. MIDL is a Microsoft’s extension of the standard DCE/IDL. A client can invoke a server method using information contained in the type library.

A type library specification is defined by its MIDL code. A COM client has all the information to use a COM server after the MIDL compiler has processed the type library specification and the marshaling/unmarshaling code defined in the server MIDL file. A type library is best thought of as a binary version of a MIDL file. It contains a binary description of the interfaces exposed by a component, defining the methods along with their parameters and return types.

COM/DCOM defines two approaches to use existing components as building blocks of new ones: containment/delegation and aggregation. For the purposes of our approach, since "the aggregation is not a
general composition mechanism” [38, 64, 65], we are interested only in the containment/delegation composition mechanism. That is, an outer component encapsulates one or more inner components and uses their services in order to implement its own.

In order to make a client able to uniquely locate a server, COM/DCOM associates a Globally Unique Identifier (GUID) to each component’s interface, class and type library. In C++, there are data types defined by COM header files for GUID, CLSID (COM class GUID), and IID (COM interface GUID).

A COM server has four types of threading models:

- **single**: in any process there is only one thread. There is an unique thread that manages all objects created by the thread itself;
- **apartment**: in any process there are one or more threads. A thread manages one or more objects, but a server does not execute at the same time two requests coming from two different threads;
- **free**: in any process there are one or more threads. A thread manages one or more objects, and a server can execute at the same time two requests from two different threads;
- **both**: apartment + free.

The services (i.e., interface’s methods) provided by a COM server have a return value\(^2\). The type of this value is called HRESULT. The HRESULT type assumes a unique success value (S_OK) and a given set of error values (E...). Typically if a request returns S_OK it means that the server has been able to provide to the client the requested service. If the server returns some E... value it means that it has not been able to provide to the client the requested service.

### 3.5.2 The Role of Windows Registry

All the information that the COM/DCOM runtime (i.e., ole32.dll) needs to uniquely locate and execute components is stored in the “Windows” registry. This information takes into account the location of a COM/DCOM component, its type library, its interfaces and its execution and access rights. The “Windows” registry has a tree structure. It has a root node (labeled with “My Computer”) and a set of sub-trees called “hives”. Each “hive” contains a set of keys. Each key may contain one or more sub-keys, and so on. A key may have multiple values and each value is assigned a name and a data.

A very interesting aspect for our purposes is that by simply acting on the registry COM/DCOM provides an easy way to interpose a component between other interacting components. For example if we have one client C interacting with one server S, we can interpose another server K between C and S in such a way that when C creates an instance of an object defined by S, the COM/DCOM runtime returns to C a reference to an instance of the same object as it is defined by K\(^3\) (instead of a reference to an instance of the object defined by S). This mechanism is hidden to C and S (i.e., it is transparent to C and S and it is possible to keep them completely unchanged). First, we have to create a TreatAs key for S under the hive HKEY_CLASSES_ROOT/CLSID/<...CLSID of S...>. Then, we have to set the value of that TreatAs to the CLSID of K. We can do that either by hands, during the deployment phase of K, or automatically (i.e., at run-time) by using the COM API function CoTreatAsClass. CoTreatAsClass takes two CLSIDs: clsid1 and clsid2. It creates the key HKEY_CLASSES_ROOT/CLSID/<clsid1>/TreatAs and sets its value to clsid2.

---

\(^1\)They are unique across all machines.

\(^2\)With respect to communication aspects at level of COM/DCOM infrastructure.

\(^3\)The assumption is that K has to define the same objects that are defined by S.
In this chapter we formalize our approach to the automatic composition and adaptation of software components. By referring to Chapter 2, in doing this, we focus on adaptation at the level of syntactic rules of interaction and of protocol interaction among components. Thus we formally characterize the automatic synthesis of four kinds of adaptors (i.e., coordinator components within our reference architectural style discussed in Section 3.1): (i) deadlock-free, (ii) interface’s signature mismatch-free, (iii) protocol interaction failure-free and (iv) protocol-enhanced adaptors.

By referring to Chapter 1, the first three kinds of adaptors address two problems: (1) how to make a set of components able to communicate and (2) how to restrict the set of system behaviors to a subset of “safe” or “desired” behaviors. Those adaptors are required in order to bridge interface’s signature incompatibilities or to restrict to the subset of deadlock-free behaviors or, in general, to a subset of system behaviors that validate specified coordination policies.

The fourth kind of adaptors addresses the problem of augmenting the set of system behaviors by introducing more sophisticated interactions among components. This adaptor is required in order to enhance reliability, add extra-functionality or deal with architectural updates such as adding new components, removing or replacing old ones by still maintaining specified functional properties of the composed system.

For the sake of presentation, in the reminder of the chapter, the synthesis technique for each of the above mentioned kinds of adaptors will be formalized by means of an iterative process.

In Section 4.1, we start by formalizing the synthesis of a “no-op” adaptor, which is a simple glue code\(^1\) for a set of components that have compatible interface signatures. Moreover, we extend the synthesis of “no-op” adaptors to formalize the synthesis of deadlock-free adaptors.

Then, in Section 4.3 we extend the synthesis algorithm described in Section 4.1 to avoid deadlocks also when we have to deal with a set of components that might have incompatible interface signatures.

Subsequently, in Section 4.4 we extend the algorithm described in Section 4.3 to synthesize adaptors that both avoid deadlocks and enforce specified interactions among components (i.e., specified coordination policies).

Finally, in Section 4.5 we extend the algorithm discussed in Section 4.4 to synthesize adaptors that are not only able to restrict the set of system behaviors but they can also augment it.

Note that although the reference architectural style our approach is based on belongs to layered systems (see Section 3.1), it is enough to consider only single-layer systems in order to fully formalize our approach. This is due to Section 4.2 where we show that, within our reference architectural style and under suitable assumptions, it is possible to decompose a \( n \)-layer system \( S \) in \( n \) single-layer (sub-)systems \( S_1, \ldots, S_n \) that

\(^{1}\)It is a simple delegator of component requests and notifications without any new logic.
we can treat independently to globally apply our approach to $S$.

All the above mentioned algorithms have been completely implemented in our SYNTHESIS tool (see Chapter 5) which is available at the following URL: http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.html.

4.1 AUTOMATIC SYNTHESIS OF DEADLOCK-FREE ADAPTORS

The problem we want to treat in this section can be phrased as follows: given a set of “interacting” components $C$, automatically derive an assembly $A$ of these components which is deadlock-free.

The term “interacting” denotes that we are considering components that have compatible interface signatures and hence they might already constitute an assembly that possibly contains deadlocks. The goal is to either assemble or reassemble these components in order to avoid all possible deadlocks in the interaction among them.

The basic ingredients of this problem are: i) the type of components we refer to, ii) the structure (i.e., the software architecture) of the assembly $A$ and iii) the meaning of “deadlock”. We consider truly black-box components (e.g., COTS or third-party components). The assembly $A$ depends on the constraints induced by the architectural model the system is based on. By referring to Chapter 3, this architectural model, which defines the rules used to build the composed system, is a CBA (i.e., Coordinator Based Architecture) style. Thus, in the reminder of the thesis we will use the term coordinator to identify an adaptor.

It is worthwhile noticing that, besides assuming that the system architecture must reflect the rules of a well defined architectural style (namely the CBA style), we also assume that a behavioral specification is provided in the form of CCS processes. Thus when we say: Given a set of “interacting” components $C$ in the problem definition we mean that we consider a set of component behavioral specifications $C$.

In our context, deadlock is the base failure because it is implicitly specified as a state of the composed system from which no action can be performed. In order to rigorously provide the meaning of “deadlock” we give the following definition to describe the deadlock problem in a component-based context:

**Definition 3 (Deadlock)** A set of components is deadlocked if each component in the set is waiting for an event that only a different component in the set can cause.

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

- observable deadlocks;
- hidden deadlocks.

For both kind of deadlocks, the behavior of a component is wrong with respect to the behavior of its environment even though the component behavior is “correct” in a stand-alone context. Moreover, the deadlock occurs during the interaction among a component and its environment. The difference between these two kinds of deadlocks is that while for the first the failure is an event that is “observable” by the component’s environment for the second class the failure is externally non-“observable” since it might depend on internal characteristics of the component.

Thus while observable deadlocks can be treated in the component setting by operating on the architectural context, namely on the coordinator (i.e., the adaptor), the hidden deadlocks cannot be addressed automatically. The only way to solve the problem is to modify the internal behavior of a component. This is not
possible with black-box components. An example for this deadlock type is offered by Compressing Proxy problem [29]. For this reason, we focus only on the first class of problems, attempting to create coordinators that can eliminate observable deadlocks. We give a more detailed characterization of observable deadlocks in Section 4.1.3. For the sake of simplicity, in the reminder of the thesis we will use the term deadlock to mean an observable deadlock.

Informally our approach is the following. The method starts with a set of black-box components, and builds a “no-op” coordinator following the CBA style constraints. We recall that the “no-op” coordinator does nothing (at this point), it simply passes events between communicating components. Then deadlocks analysis is performed. If the synthesized “no-op” coordinator contains deadlocks, a recovery technique is applied. Depending on the kind of deadlock, the analysis of only the “no-op” coordinator is enough to obtain its corresponding deadlock-free coordinator.

4.1.1 METHOD DESCRIPTION

In this section we introduce our method by using an explanatory example. As illustrated in Figure 4.1 we proceed in two steps. The first step starts with a CFA system and automatically produces a new configuration with the same components plus a “no-op” coordinator, which filters all the connections among the components obeying our CBA style. In Section 4.1.4 we proof that the two systems behave equivalently under a suitable notion of equivalence.

The second step performs the deadlock analysis, on the CBA system to detect any deadlocks. Subsequently, we can operate on the “no-op” coordinator in order to obtain a deadlock-free equivalent system.

Figure 4.1: 2 step method

---

2Note that although in principle we could perform the two steps together we decided to keep them separate. This has been done to support internal data structures traceability.
A WORKING EXAMPLE: THE DINING PHILOSOPHERS

The example we use in this section to formalize and describe our methodology is the dining philosophers problem [44]. We use this example because it is well known and it is a coordination problem exemplifying the kind of problems our methodology can solve. To simplify the presentation we have instantiated the problem with two forks and two philosophers which represent our black-box components.

Figure 4.2 represents both the component structure of the CFA system and the behavior of each component in terms of a CCS process, while Figure 4.3 shows its formalization under the CBA reference architectural style.

Figure 4.2: Structure and behavior of the components of the dining philosophers problem

Each CCS process models the component behavior in an intuitive way as a LTS. Each CCS term represents a state of the component and the term $S_0$ represents its initial state. Each I/O action performed from/towards the component’s environment (i.e., all the other components in parallel) represents a label of a transition into a new state (i.e., the CCS term which follows that action).

Figure 4.3: Architecture formalization under the CBA reference style

4.1.2 METHOD FORMALIZATION

Starting from the CCS behavioral description of a component, we associate to it a set of labeled transition systems that characterize different aspects of the component dynamics, from the actual component behavior
4.1 Automatic synthesis of deadlock-free adaptors

to its assumptions on the environment.

For our purposes we need to enrich the LTS structure by adding labels on states as well. The following definition introduces Actual Behavior Graph (AC-Graph) which is the first basic structure we use in the framework.

**Definition 4 (AC-Graph)** Let \((N_i, L_i, \rightarrow_i, S_0)\) be the labeled transition system of a component \(C_i\). The corresponding Actual Behavior (AC) Graph \(AC_i\) is a tuple of the form

\[
\langle N_{AC_i}, L_{AC_i}, A_{AC_i}, LA_{AC_i}, S_0 \rangle
\]

where \(N_{AC_i} = N_i\) is a set of nodes, \(L_{AC_i}\) is a set of state labels such that \(L_{AC_i} = \{S_j \text{ such that } 0 \leq j < |N_i|\}\), \(LA_{AC_i}\) is a set of arc labels such that \(LA_{AC_i} = \{?a \text{ such that } a \in L_i\} \cup \{!a \text{ such that } a \in L_i\}\), \(A_{AC_i} \subseteq N_{AC_i} \times LA_{AC_i} \times N_{AC_i}\) is a set of arcs and \(S_0\) is the initial state.

- We shall write \(g \overset{\alpha}{\rightarrow} h\), if there is an arc \((g, \alpha, h)\) \(\in A_{AC_i}\). We shall also write \(g \rightarrow h\) meaning that \(g \overset{\alpha}{\rightarrow} h\) for some \(\alpha \in LA_{AC_i}\).
- If \(t = \alpha_1 \cdots \alpha_n \in LA_{AC_i}^*\), then we write \(g \overset{t,n}{\longrightarrow} h\), if \(g \overset{\alpha_1}{\rightarrow} \cdots \overset{\alpha_n}{\rightarrow} h\). We shall also write \(g \overset{n}{\rightarrow} h\), meaning that \(g \overset{t,n}{\longrightarrow} h\) for some \(t \in LA_{AC_i}^*\) such that \(n\) is the length of \(t\).

The term *actual* emphasizes the difference between the component behavior and the intended, or assumed, behavior of the environment. In our case we shall derive an AC-Graph from CCS terms as described in Section 3.2. This suggests a natural way to label states: syntactic CCS terms correspond to states of the AC-Graph retaining the syntactic term as the label of the corresponding state. Thus in the following we interchangeably identify with \(S_i\) both the \(i\)-th state and the label associated to the \(i\)-th state. The context allows to distinguish when we are talking about the actual state or of the state label.

![SYNTHESIS](image)

Figure 4.4: AC-Graphs of the dining philosophers components

Figure 4.4 shows screen-shots of our SYNTHESIS tool (see Chapter 5) that illustrate the AC-Graphs of the components of our explanatory example. By using the SYNTHESIS tool, we have modeled the actual
behavior of the two forks (i.e., *Fork*1 and *Fork*2 functional components) and of the two philosophers (i.e., *Philosopher*1 and *Philosopher*2 functional components). The philosophers have exactly the same graphs thus only one appears in the figure (i.e., *Philosopher*1). We recall that by \( a \)! SYNTHESIS denotes an output action \( 'a' \) and by \( a \)? it denotes an input action \( 'a' \). By referring to Figure 4.4, the forks components can iteratively wait for a request (i.e., actions \(?fork1\) and \(?fork2\)), give the fork (i.e., actions \(!ok1\) and \(!ok2\)), and then wait for the fork to be released (i.e., actions \(?leave1\) and \(?leave2\)). Note that we refer to the components as *Fork*1 (*Fork*2) and to the actual resource fork the components hold as *fork*1 (*fork*2). Moreover, SYNTHESIS internally denotes *Fork*1, *Fork*2, *Philosopher*1 and *Philosopher*2 as \( C_1 \), \( C_2 \), \( C_3 \) and \( C_4 \) respectively.

The above graphs describe the interaction behavior of each component with the external environment.

Now, we wish to automatically derive from a component behavior the requirements on its environment that guarantee deadlock freedom. A system is in deadlock when it cannot perform any computation, thus in our setting, deadlock means that all components are blocked waiting for an action from the environment that is not possible.

**Definition 5 (AS-Graph)** Let \((N_{AC}, LN_{AC}, A_{AC}, LA_{AC}, S_0)\) be the AC-Graph \( AC_i \) of a component \( C_i \), then the corresponding Assumption (AS) Graph \( AS_i \) is \((N_{AS}, LN_{AS}, A_{AS}, LA_{AS}, S_0)\) where \( N_{AS} = N_{AC}, LN_{AS} = LN_{AC}, A_{AS} = LA_{AC}, LA_{AS} = LA_{AC} \) and \( A_{AS} = \{(\nu,!\alpha,\nu') such that (\nu,\alpha,\nu') \in A_{AC}\} \).

Figure 4.5: AS-Graphs of the dining philosophers components

AC and AS-Graphs were introduced in [35]. If a component’s AC-Graph represents the component’s actual behavior, the AS-Graph models the dynamic behavior of the environment that the component expects in order to not block. Figure 4.5 reports SYNTHESIS’s screen-shots that show the AS-Graphs of the Dining philosophers components. Analogously to AC-Graphs, we have one graph for each component. The only difference from AC-graphs is in the arcs labels, which are symmetric since they model the environment as each component expects it.

In the architectural style we have chosen, the component environment can only be represented by one or more coordinators. Thus we can refine the definition of AS-Graph in a new graph, the EX-Graph, that represents the behavior that the component expects from the coordinator. In Section 3.1 we said that the coordinator performs strictly sequential input-output operations only. Thus if it receives an input from a
component it will then immediately output the received input message to a destination component. Analogously, if the coordinator outputs a message, this means that immediately before it inputs that message from a source component. Intuitively, for each transition labeled ?a in the AS graph of a component Ci (i.e., ASi), in the corresponding EX graph (i.e., EXi) there are two strictly sequential transitions labeled ?a,i and !a,# respectively. Analogously, for each transition labeled !a in Si, in EXi there are two strictly sequential transitions labeled ?a,# and !a,i respectively. Within EXi, the action ?a,i (!a,i) denotes that Ci expects the coordinator receive (send) an input (output) message ?a (!a) from (towards) the same Ci; thus, in this case, we say that ?a (!a) is a “known” action for Ci (and, hence, in the EX-Graph of Ci that action is denoted as ?a,i (!a,i)). Action !a,# (?a,#) denotes that Ci expects the coordinator to send (receive) an output (input) message !a (?a) towards (from) a component that is unknown for Ci; thus, in this case, we say that !a (?a) is a “unknown” action for Ci (and, hence, in the EX-Graph of Ci that action is denoted as !a,# (?a,#)).

Each EX-Graph represents a partial view of the coordinator expected behavior. It is partial since it only reflects the expectations of a single component. The global coordinator behavior will be derived by taking into account all the EX-Graphs. This will be done in the next section, through a sort of unification algorithm. The following is the algorithm that SYNTHESIS uses to automatically derive from an AS-Graph its corresponding EX-Graph:

**Definition 6 (EX-Graph construction algorithm)** Let \((N_{AS}, LN_{AS}, A_{AS}, LA_{AS}, S0)\) be the AS-Graph \(AS\) of a component \(Ci\); we define the coordinator EXpected (EX) Graph \(EXi\) of the component \(Ci\); the graph \((N_{EX}, LN_{EX}, A_{EX}, LA_{EX}, S0)\) which is built by performing the following algorithm:

- at the beginning \(N_{EXi} = N_{ASi}\) and \(LN_{EXi} = LN_{ASi}\);  
- at the beginning \(A_{EXi}\) and \(LA_{EXi}\) are empty;  
- for all \((\mu, \alpha, \mu') \in A_{ASi}\) do:  
  - create a new node \(\mu_{new}\) with a new unique label, add the node to \(N_{EXi}\) and the unique label to \(LN_{EXi}\);  
  - if \((\mu, \alpha, \mu')\) is such that \(\alpha\) is an input action (i.e., \(\alpha = ?a, for some a\) then:  
    * add the labels ?a,i and !a,# to \(LA_{EXi}\);  
    * add \((\mu, ?a,i, \mu_{new})\) and \((\mu_{new}, !a,#, \mu')\) to \(A_{EXi}\);  
  - if \((\mu, \alpha, \mu')\) is such that \(\alpha\) is an output action (i.e., \(\alpha = !a, for some a\) then:  
    * add the labels !a,i and ?a,# to \(LA_{EXi}\);  
    * add \((\mu, ?a,#, \mu_{new})\) and \((\mu_{new}, !a,i, \mu')\) to \(A_{EXi}\).

\(EXi\) contains complete information for the actions exchanged on the connector in between \(Ci\) and the coordinator. On the other hand, \(EXi\) contains partial information for the actions exchanged on the connectors in between components different from \(Ci\) and the coordinator.

In Figure 4.6, the EX-Graphs of the dining philosophers components are shown. Let us look at the Fork1 component EX-Graph. It represents the partial view the component Fork1 has of the expected behavior of the coordinator. Fork1 expects the coordinator first input a request of fork1 from a non-identified component (it might be either Philosopher1 or Philosopher2), then the coordinator must output this request to the component Fork1, wait for a notification from Fork1, deliver this notification to the non-identified component that asked for fork1, wait for the fork to be released from the non-identified component and then delegate the request of releasing the fork to the component Fork1.
Chapter 4. Automatic synthesis of “correct” and protocol-enhanced adaptors

Figure 4.6: EX-Graphs of the dining philosophers components

“No-op” COORDINATOR SYNTHESIS

Let us now present the algorithm for the “no-op” coordinator creation. Since our algorithm makes use of a unification technique over first order terms [41], we name it unification of EX-Graphs. The “no-op” coordinator synthesis is actually based on the unification of the component EX-Graphs. The EX-Graph represents the behavior that the component expects from the “no-op” coordinator. Each component has only its partial view of the “no-op” coordinator behavior, by unifying all the component views we can synthesize the “no-op” coordinator global behavior. Finally, by simply acting on the model of the “no-op” coordinator behavior (i.e., a state machine) we avoid possible deadlocks.

DEFINITIONS OF “unifiable” AND OF “adjacent”

Referring to [41] we can say that abstractly, the unification problem is the following: “given two descriptions \( x \) and \( y \), can we find an object \( z \) that fits both descriptions?”

For our purpose the unification problem can be stated as follows: let an action term be a “known” action label plus an integer which denotes the component that knows this action (e.g., \( (t, i) \) is the term that denotes the action \( t \) that is known to \( C_i \)) and let an action variable be an “unknown” action plus an integer number which denotes the component that does not know this action (e.g., \( (v, j) \) is the variable which denotes the action \( v \) that is unknown to \( C_j \)) then given an action term \( (t, i) \) and an action variable \( (v, j) \), do \( t \) and \( v \) denote the same action? (i.e., \( t \) and \( v \) are the same label and \( i \) is different from \( j \)).

It is worth noticing that our notion of term and variable does not exactly correspond to the usual notion given in the context of unification over first order terms [41]. Since our synthesis algorithm is based on the idea of iteratively trying to match unknown actions in an EX-Graph with known actions in a different EX-Graph (i.e., find a syntactic match between their labels) - for the sake of simplicity - in our context we consider a known (unknown) action as an action term (action variable).

In the following we give the formal definitions necessary to fully-define the notion of unification that the coordinator synthesis algorithm is based on:
4.1 Automatic synthesis of deadlock-free adaptors

Definition 7 (action_term) Let \((N_{EX_i}, LN_{EX_i}, A_{EX_i}, LA_{EX_i}, S_0)\) be the EX-Graph \(EX_i\) of a component \(C_i\). An action_term of \(C_i\) is a pair \((t, i)\) where \(t, j \in LA_{EX_i}\) (i.e., \(t = \text{?}a\) or \(t = !a\) for some action \(a\)).

Definition 8 (action_variable) Let \((N_{EX_i}, LN_{EX_i}, A_{EX_i}, LA_{EX_i}, S_0)\) be the EX-Graph \(EX_i\) of a component \(C_i\). An action_variable of \(C_i\) is a pair \((v, i)\) where \(v, j \in LA_{EX_i}\) where (i.e., \(v = \text{?}a\) or \(v = !a\) for some action \(a\)).

Definition 9 (unifiable pair of action_term and action_variable) A pair of action_term and action_variable in the form of \(((t, i), (v, k))\) is unifiable if \(t = v\) and \(i \neq k\).

Definition 10 (adjacent node) Let \(K = (N_K, LN_K, A_K, LA_K, S_0)\) be the “no-op” coordinator’s AC-Graph, let \(g\) be an element of \(N_K\) and let \(\text{AdjSet}(g)\) be the set \(\{g'\} \text{ such that there exist } \alpha, \beta \text{ such that } (g, \alpha, g') \in A_K\} \). For all \(g'' \in \text{AdjSet}(g)\), \(g''\) is an adjacent node of \(g\).

For our purposes, the notion of adjacent node of a node \(g\) does not correspond to the natural notion of being a node that is reachable in a 1-step transition from \(g\). Differently from this, we consider as adjacent nodes of a node \(g\) all the ones that are reachable in a 2-steps transition from \(g\).

EX-GRAPHS UNIFICATION ALGORITHM

Intuitively, we attempt to match known actions in a EX-Graph \(EX_i\) (i.e., action_terms of \(C_i\)) with unknown actions in another EX-Graph \(EX_j\) (i.e., action_variables of \(C_j\)). In the following we will interchangeably use the terms state and node. The following is the EX-Graphs unification algorithm we use to automatically synthesize the actual behavior graph of the “no-op” coordinator for a single-layer system:

Definition 11 (EX-Graph unification algorithm)

- Let \(AC_1, ..., AC_n\) be the AC-Graphs of the components \(C_1, ..., C_n\) forming the CFA-version of the composed system;
- Let \(AS_1, ..., AS_n\) be the AS-Graphs corresponding to \(AC_1, ..., AC_n\) respectively;
- Let \(EX_1, ..., EX_n\) be the EX-Graphs corresponding to \(AS_1, ..., AS_n\) respectively;
- Let \(S_1, ..., S_n\) be the current states of \(EX_1, ..., EX_n\) respectively.

At the beginning the current states are the initial states of \(EX_1, ..., EX_n\).

1. Create the AC-Graph of the “no-op” coordinator, with one node (initial state) and no arcs.
2. Set as current states of the components EX-Graphs the respective initial states.
3. Label the initial state of the “no-op” coordinator AC-Graph with an ordered tuple composed of the initial states of all components EX-Graphs. For the sake of presentation we assume to order them so that the \(j - th\) element of the state label corresponds to the current state of the component EX-Graph \(EX_j\) where \(j \in [1, ..., n]\). This state is the coordinator current state \(g\).
4. Perform the following unification procedure \(\text{Unify}(g)\):
(a) Mark \( g \) as visited.

(b) Let \(< S_1, \ldots, S_n >\) be the state label of \( g \).

(c) Generate the set \( \text{TER} \) of action terms and the set \( \text{VAR} \) of action variables so that \((t, i) \in \text{TER}\) if in \( \text{EX}_i \) we have \( S_i \xrightarrow{t} V_i \). Similarly \((v, j) \in \text{VAR}\) if in \( \text{EX}_j \) we have \( S_j \xrightarrow{v} V_j \).

(d) For all unifiable pairs \((t, i), (v, j)\) do:
   i. if the two new nodes \( g_i, g_j \) with state label \(< S_1, \ldots, S_i, \ldots, S_j, \ldots, S_n >\) and
      \(< S_1, \ldots, S'_i, \ldots, S'_j, \ldots, S_n >\) respectively, where \( S_i \xrightarrow{t} S'_i \) in \( \text{AS}_i \) and
      \( S_j \xrightarrow{v} S'_j \) in \( \text{AS}_j \), do not already exist (in the “no-op” coordinator AC-Graph) then
         A. create \( g_i \) and \( g_j \);
         B. create the arc \((g, t, g_i)\) in the “no-op” coordinator AC-Graph;
         C. mark \( g_i \) as visited;
         D. create the arc \((g_i, v, g_j)\) in the “no-op” coordinator AC-Graph.

(e) Perform recursively \( \text{Unify}(g'') \) for all not marked (as visited) adjacent nodes \( g'' \) of current node \( g \).

Figure 4.7 shows the first step of this algorithm illustrating portions of EX-Graphs for the components of the Dining Philosophers system and the “no-op” coordinator portion automatically synthesized by the EX-Graphs unification algorithm after the first unification step. The figure also shows the set of action terms and the set of action variables built during the first unification step. In Figure 4.7 near each “no-op” coordinator node, we report also the “no-op” coordinator node labels written as tuples of EX-Graph states. In Figure 4.8 we show a screen-shot of our SYNTHESIS tool which illustrates the “no-op” coordinator AC-Graph automatically generated through the EX-Graphs unification.

As above mentioned, SYNTHESIS associates to each “no-op” coordinator AC-Graph’s node an ordered tuple of EX-Graphs nodes. It is worthwhile noticing that this is done only internally. In fact, externally, SYNTHESIS denotes the \( i \)-th generated node of the “no-op” coordinator AC-Graph with the label \( S_i \) (see Figure 4.8). Moreover, as showed in Figure 4.8, SYNTHESIS denotes as filled states all the nodes that are elements of a “deadlocking path”. We refer to Section 4.1.3 for a formal definition of a “deadlocking path”. Intuitively, let a “deadlocking state” be either a state of the “no-op” coordinator AC-Graph from which no action is performed (i.e., it is a sink node) or a state whose outgoing transitions all lead to “deadlocking states”, a “deadlocking path” is a sequence of “deadlocking states”. We recall that the resulting CBA system is the following CCS term:

\[
(C_1[f_1] \mid C_2[f_2] \mid \ldots \mid C_n[f_n] \mid K) \setminus \bigcup_{i=1}^{n} \text{Act}_i[f_i].
\]

Refer to Section 3.3 for a definition of relabeling functions \( f_i \).

In Section 4.1.4 we prove that the behavior of the CBA-system based on the “no-op” coordinator is equivalent to the behavior of the CFA-system, under a suitable notion of equivalence. This proves correctness and completeness of our “no-op” coordinator synthesis technique.

4.1.3 DEADLOCK DETECTION AND RECOVERY

In Section 4.1 we divided the kind of deadlocks that can occur in our context in two kinds: observable deadlocks and hidden deadlocks. We recall that we can focus only on deadlocks of the first class due to the nature of these two kinds of deadlocks.

In this section we present our technique to detect possible deadlocks in the CBA-system and to fix them.
Deadlocks can be solved by directly operating on the structure of the “no-op” coordinator’s AC-Graph:

**Definition 12 (Deadlocking state)** Let $K = \langle N_K, LN_K, A_K, LA_K, S_0 \rangle$ be a coordinator’s AC-Graph then a deadlocking state of $K$ is a state $v \in N_K$ such that either $v$ is a sink node of $K$ (i.e., $v$ has no outgoing transition) or for all $v'$ such that $v \rightarrow v'$ then $v'$ is a deadlocking state.

**Definition 13 (Deadlocking path)** Let $K = \langle N_K, LN_K, A_K, LA_K, S_0 \rangle$ be a coordinator’s AC-Graph then a deadlocking path of $K$ is a sequence $v_1, \ldots, v_h$ of states such that $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_h$ and $\forall i = 1, \ldots, h : v_i$ is a deadlocking state of $K$.

By referring to the preceding definition, if deadlocks exist we can find them by performing an analysis of the “no-op” coordinator’s AC-Graph in order to discover its possible deadlocking paths. By continuing our example these paths are the paths made only of filled states as shown in Figure 4.8. To remove the deadlocks, we have only to cut the deadlocking paths of the coordinator’s AC-Graph. Cutting a deadlocking path corresponds to rejecting (i.e., returning an erroneous notification) the component’s request that has led the coordinator to execute that path.

In Figure 4.9 we show the “no-op” coordinator’s AC-Graph of the dining philosophers example after the deadlocks removal.

By taking into account the deadlock-free coordinator’s AC-Graph and by exploiting the information stored in each node and arc we can automatically derive the code which implements the deadlock-free coordinator component (i.e., the actual assembly code which is deadlock-free). In Section 5.2.7 we show the technique
Figure 4.8: Automatically synthesized “no-op” coordinator

used to derive the actual assembly code corresponding to the model of the coordinator in detail. We applied this technique in the context of COM/DCOM applications.

In Section 4.1.4 we prove that the CBA-system based on the deadlock-free coordinator (i.e., the “no-op” coordinator’s AC-Graph without possible deadlocking paths) preserves all the correct behaviors (i.e., the infinite paths) of the CFA-system and vice versa. This proves the correctness and completeness of our technique for detecting and fixing possible deadlocks.
4.1.4 CORRECTNESS AND COMPLETENESS

In this section we prove correctness and completeness of our synthesis technique for both “no-op” and deadlock-free coordinators.

CORRECTNESS AND COMPLETENESS OF THE “no-op” COORDINATOR SYNTHESIS

We prove that the “no-op” coordinator synthesis algorithm is correct and complete with respect to the set of behaviors of the CFA-system. In so doing, let $V$ be the CBA-system obtained by inserting the automatically generated “no-op” coordinator in the CFA-system $T$. Then $T$ and $V$ can be shown to be equivalent under a suitable notion of behavioral equivalence.

First, we prove that a) given the components $C_1, \ldots, C_n$, the CBA-system $V$ formed by the $n$ components plus the “no-op” coordinator simulates the CFA-system $T$ formed by the $n$ components, and b) $V$ does not perform any new logic (with respect to $T$) except for an extra level of indirection. This proves correctness. Second, we prove that $T$ simulates $V$ except for the extra level of indirection that the “no-op” coordinator has introduced in $V$. This proves completeness.
Correctness

To state the correctness of the “no-op” coordinator synthesis we define a two step proof:

- we prove that the CBA-system can simulate the CFA-system, under a suitable notion of simulation;
- we prove that the “no-op” coordinator in the CBA-system does not introduce any new logic into the system, thus the CBA-system can perform only the transitions of the CFA-system with an extra level of indirectness since all the interactions must go through the “no-op” coordinator.

To perform the first step, we define a suitable relation of simulation; the starting point of the following definition is stuttering equivalence [52]. The notion of simulation we use (i.e., CB-Simulation) observes only the states of the CFA-system and of the CBA-system. This is due to how we have formalized the CFA-system and the CBA-system in Section 3.3. In fact, we obtain that both the CFA-system and the CBA-system are made only of \( \tau \) transitions:

**Definition 14 (CB-Simulation)** Let \( S \) and \( T \) be two systems and \( s \) and \( t \) two generic states of the systems;

- a relation \( \leq_{CB} \) is called CB-simulation if \( s \leq_{CB} t \) then:
  - the state label of \( s \) is equal to the state label of \( t \);
  - if \( s \rightarrow s' \) then there exists \( n > 0 \), \( t_0, ..., t_n \) such that \( t = t_0 \) and for all \( i < n \):
    \[
t_i \rightarrow t_{i+1}, s' \leq_{CB} t_n.
    \]
- A state \( t \) CB-simulates a state \( s (s \leq_{CB} t) \) if it exists a relation of CB-simulation between \( s \) and \( t \);
- A path \( \sigma \) CB-simulates a path \( \rho (\rho \leq_{CB} \sigma) \), if \( \rho \) can be partitioned as \( \rho_1 \rho_2... \) and \( \sigma \) can be partitioned as \( \sigma_1 \sigma_2... \) in such a way that, for all \( j \), the sequences \( \rho_j \) and \( \sigma_j \) are not empty and \( \rho_j \leq_{CB} \sigma_j \).

It is worthwhile noticing that we label a state \( s \) in a CFA-system as a tuple \( < S_1, ..., S_n > \) where \( S_i \) is the label of the state reached by the AC-Graph of the component \( C_i \) when the CFA-system reaches the state \( s \) and \( n \) is the number of the components forming the CFA-system. Analogously, we label a state \( t \) in a CBA-system. More precisely, since the state label of \( s \) in the CBA-system pass through the “no-op” coordinator, we denote a state in the CBA-system with a state of the “no-op” coordinator AC-Graph. Thus a state \( t \) in the CBA-system is a tuple \( < T_1, ..., T_n > \) where \( T_i \) is the label of the state reached by the EX-Graph of the component \( C_i \) when the CBA-system reaches the state \( t \) and \( n \) is the number of the functional components (i.e., the “no-op” coordinator is not included) forming the CBA-system.

We can now assess correctness of our “no-op” coordinator construction:

**Proposition 1 (Correctness)** Let \( T \) be a CFA-system, let \( V \) be the corresponding CBA-system and let \( t_0, v_0 \) be the initial states of \( T \) and \( V \) respectively; for all states \( t \) reachable from \( t_0 \) in \( T \), there exists a reachable state \( v \) from \( v_0 \) in \( V \) such that the label of \( v \) is equal to the label of \( t \).

**Proof:** We give proof by induction on the length of a path in \( T \). By construction, the base step (i.e., for a path length equal to 0) is trivially proved because the state label of \( t_0 \) is equal to the state label of \( v_0 \). The inductive step (i.e., for a path length equal to \( m > 0 \)) consists in supposing that a generic state \( t \) labeled with \( < T_1, ..., T_n > \) in \( T \) exists. \( t \) is such that \( t_0 \xrightarrow{m} t \) (for some \( m \)) and the state label of \( t \) is equal to the state label \( < V_1, ..., V_n > \) of a state \( v \) in \( V \) (i.e., \( \forall i = 1, ..., n : T_i = V_i \)). Moreover let us suppose that \( v_0 \xrightarrow{p} v \) (for some \( p \)). Then we have to prove that given a state \( t' \) such that \( t \rightarrow t' \), it is possible to
find a state \( v' \) such that \( v \xrightarrow{q}^* v' \) (for some \( q \)) and the state label of \( v' \) is equal to the state label of \( t' \). Let us suppose that \( t' \) is labeled by \( < T_1, ..., T_i', ..., T_k', ..., T_n > \) because \( t \xrightarrow{!l} t' \) by synchronizing \( C_i \) and \( C_k \) on actions \( !!l \) and \( !!l \), respectively.

\[
\text{It means that } \exists i = 1, ..., n \land \exists k = 1, ..., n \land i \neq k : (T_i, !!l, T_i') \in A_{AC}, \text{ and } (T_k, !!l, T_k') \in A_{AC}.
\]

By definition of AS-Graph and EX-Graph we have that \( (T_i, !!l, T_i'), (T_k, !!l, T_k') \in A_{EX} \) and \( (T_i, !!l, T_i'), (T_k, !!l, T_k') \in A_{EX} \). By definition of EX-Graphs unification algorithm we have that \( (T_i, !!l, T_i'), (T_k, !!l, T_k') \in A_{EX} \) and \( (T_i, !!l, T_i'), (T_k, !!l, T_k') \in A_{EX} \).

Thus \( v = t = < T_1, ..., T_i, ..., T_k, ..., T_n > \xrightarrow{q(=2)} < T_1, ..., T_i', ..., T_k', ..., T_n > = v' \) is in the transition relation defined for \( V \). By hypothesis, the state label of \( v' \) is equal to the state label of \( t' \) and both \( v' \) and \( t' \) are reachable from \( v_0 \) and \( l_0 \) respectively. We have verified the inductive step by construction, and thus the proof is complete.

**Corollary 1** Let \( T \) be a CFA-system, let \( V \) be the corresponding CBA-system and let \( l_0, v_0 \) be the initial states of \( T \) and \( V \) respectively; then \( T \) can be CB-simulated by \( V \).

**Proof:** The proof is trivially derived from the definition of CB-Simulation and by Proposition 1. Actually, by Proposition 1, for each state \( t \) in \( T \) reachable from \( v_0 \) it exists a state \( v \) in \( V \) reachable from \( v_0 \) in such a way that the state label of \( t \) is equal to the state label of \( v \). If we apply Proposition 1 by starting from \( l_0 \) we obtain again the CB-Simulation definition for each state of \( T \) with a state of \( V \). Thus we obtain that for each state \( t \) in \( T \) such that the state label of \( t \) is equal to the state label of some \( v \) in \( V \), if \( t \xrightarrow{l'} t' \) then there exist \( n > 0 \), \( v_0, ..., v_n \) such that \( v = v_0 \) and for all \( i < n : v_i \xrightarrow{!!l} v_{i+1} \), \( l' \leq CB v_n \). That is \( V \) CB-simulates \( T \).

The previous proposition and corollary are concerned with the first step of the proof related to the correctness of the approach. To finally state the correctness of our approach we have to prove that the “no-op” coordinator does not introduce in the system any new logic; this is trivial, because for each synchronization between two components (\( C_i \) and \( C_j \)) in the CFA-system there are two strictly sequential transitions in the CBA-system: a synchronization between the first component and the “no-op” coordinator and a synchronization between the “no-op” coordinator and the second component.

Thus we can conclude that the CFA-system is CB-simulated by the corresponding CBA-system and the “no-op” coordinator does not introduce in the system any new logic because it performs only the transitions of the CFA-system with an extra level of indirection. This directly implies that our “no-op” coordinator synthesis algorithm is correct with respect to the set of behaviors of the CFA-system because it proves that the corresponding constructed CBA-system preserves all the CFA-system behaviors.

**Completeness**

To state the completeness of our approach used for synthesizing “no-op” coordinators we prove that the CFA-system CB-simulates the corresponding CBA-system which is “restricted” in such a way that the extra level of indirection is eliminated. Let \( V \) be the CBA-system obtained by inserting the automatically generated “no-op” coordinator in the CFA-system \( T \), we consider a “restricted” version of \( V \) that we denote as \( \Pi_V \). \( \Pi_V \) is obtained by performing a procedure \( Restric(V) \) that is defined as follows:

- let \( V \) be the CBA-version of the composed system;
- let \( \Pi_V \) be the restricted (with respect to the extra level of indirection) version of \( V \);
At the beginning \( \Pi_V \) is equal to \( V \);

1. Set as current state \( v \) the initial state of \( \Pi_V \);

2. Perform the following procedure \( \text{Collapse}(v) \):
   
   \begin{enumerate}
   \item mark \( v \) as visited;
   \item for all \( v' \) such that \( v \xrightarrow{2} v' \) do:
   \begin{enumerate}
   \item collapse all the sequences of \( \tau \) transition such that \( v \xrightarrow{\tau} v'' \xrightarrow{\tau} v' \) in one \( \tau \) transition such that \( v \xrightarrow{\tau} v' \);
   \item Perform recursively \( \text{Collapse}(v') \) for all not marked (as visited) nodes \( v' \) such that \( v \xrightarrow{2} v' \).
   \end{enumerate}
   \end{enumerate}

We can now assess completeness results of our construction:

**Proposition 2 (Completeness)** Let \( T \) be a CFA-system, let \( V \) be the corresponding CBA-system, let \( \Pi_V \) be the corresponding CBA-system that is restricted in such a way that \( \Pi_V = \text{Restrict}(V) \) (i.e., in order to eliminate the extra level of indirection that the “no-op” coordinator introduces) and let \( t_0, v_0 \) be the initial states of \( T \) and \( \Pi_V \) respectively; for all states \( t \) reachable from \( t_0 \) in \( T \), it exists a reachable state \( v \) from \( v_0 \) in \( \Pi_V \) such that the label of \( v \) is equal to the label of \( t \).

**Proof:** We give proof by induction on the length of a path in \( \Pi_V \). By construction, the base step (i.e., for a path length equal to 0) is trivially proved because the state label of \( v_0 \) is equal to the state label of \( t_0 \). The inductive step (i.e., for a path length equal to \( n > 0 \)) consists in supposing that a generic state \( v \) labeled with \( < V_1, \ldots, V_i, \ldots, V_k, \ldots, V_n > \) in \( \Pi_V \) exists. \( v \) is such that \( v_0 \xrightarrow{m} v \) (for some \( m \)) and the state label of \( v \) is equal to the state label of a state \( t \) in \( T \) (i.e., \( \forall i = 1, \ldots, n : V_i = T_i \)). Moreover let us suppose that \( t_0 \xrightarrow{q} t \) (for some q). Then we have to prove that given a state \( v' \) such that \( v \xrightarrow{q} v' \), it is possible to find a state \( t' \) such that \( t \xrightarrow{q} t' \) (for some q) and the state label of \( t' \) is equal to the state label of \( v' \). Let us suppose that \( v' \) is labeled by \( < V_1', \ldots, V_i', \ldots, V_k', \ldots, V_n > \) because of \( v \xrightarrow{v'} \) by synchronizing \( C_i[f_i] \) with the “no-op” coordinator and, subsequently, the “no-op” coordinator with \( C_k[f_k] \) on coordinator actions \( \forall \lambda, \Lambda, k \), respectively.

By definition of EX-Graphs unification algorithm, it means that \( \exists i = 1, \ldots, n \wedge \exists k = 1, \ldots, n \wedge i \neq k : (V_i, \#L_i, V_i'), (V_i', \#L_i, V_i''), (V_k, \#L_k, V_k'), (V_k', \#L_k, V_k'') \in A_{EX} \), and \( (V_k, \#L_k, V_k'), (V_k', \#L_k, V_k'') \in A_{EX} \). By definition of EX-Graph, AS-Graph and AC-Graph we have that \( (V_i, \#I, V_i') \in A_{AC} \) and \( (V_k, \#I, V_k') \in A_{AC} \). By definition of CFA we have that \( (V_1, \ldots, V_i, \ldots, V_k, \ldots, V_n >, \tau, < V_1', \ldots, V_i', \ldots, V_k', \ldots, V_n >) \) is in the transition relation defined for \( T \).

Thus \( v = t \xrightarrow{p} = \langle V_1, \ldots, V_i, \ldots, V_k, \ldots, V_n >, \tau, < V_1', \ldots, V_i', \ldots, V_k', \ldots, V_n > = v' \) is in the transition relation defined for \( T \). By hypothesis, the state label of \( t' \) is equal to the state label of \( v' \) and both \( t' \) and \( v' \) are reachable from \( t_0 \) and \( v_0 \) respectively. We have verified the inductive step by construction thus the proof is given.

**Corollary 2** Let \( V \) be a CBA-system, let \( \Pi_V \) be \( \text{Restrict}(V) \), let \( T \) be the CFA-system corresponding to \( V \) and let \( v_0, t_0 \) be the initial states of \( \Pi_V \) and \( T \) respectively; then \( \Pi_V \) can be CB-simulated by \( T \).

**Proof:** The proof is trivially derived from the definition of CB-Simulation and by Proposition 2. Actually, by Proposition 2, for each state \( v \) in \( \Pi_V \) reachable from \( v_0 \) it exists a state \( t \) in \( T \) reachable from \( t_0 \) in such a way that the state label of \( v \) is equal to the state label of \( t \). If we apply Proposition 2 by starting from \( v_0 \).
we obtain again the CB-Simulation definition for each state of $\Pi_V$ with a state of $T$. Thus we obtain that for each state $v$ of $\Pi_V$ such that the state label of $v$ is equal to the state label of some $t$ in $T$, if $v \Delta t$ then there exist $n > 0$, $t_0, ..., t_n$ such that $t = t_0$ and for all $i < n : t_i \rightarrow t_{i+1}$, $v' \leq_{CB} t_n$. That is $T$ CB-simulates $\Pi_V$.

It is worthwhile noticing that by Corollaries 1 and 2 and by considering that the only new logic the coordinator adds to the composed system is the extra level of indirection, we have that the CFA-system $T$ and the restricted CBA-system $\Pi_V$ are CB-equivalent. That is $T$ and $\Pi_V$ CB-simulate each other.

**CORRECTNESS AND COMPLETENESS OF THE DEADLOCK-FREE COORDINATOR SYNTHESIS**

In this section we prove that the CBA-system based on the deadlock-free coordinator (i.e., the “no-op” coordinator’s AC-Graph without possible deadlock paths) preserves all the correct behaviors (i.e., the infinite paths) of the CFA-system. This proves the correctness of our technique for detecting and fixing possible deadlocks. Moreover, we prove that the deadlock-free CFA-system (i.e., the CFA-system without the finite paths) CB-simulates the corresponding deadlock-free CBA-system. This proves the completeness of our technique for detecting and fixing possible deadlocks.

**CORRECTNESS**

In a CBA-system the coordination of the interaction among the components has been pushed out of the functional components and completely delegated to the coordinator behavior. Functional components can only send or receive messages towards and from the coordinator, which internally manages their routing. If a deadlock is possible, then it is observable through a precise “no-op” coordinator’s behavior. This behavior is detectable by observing the “no-op” coordinator’s AC-Graph. If it contains deadlock paths, then clearly once reached the deadlock state with which a deadlock path terminates (i.e., a sink node) the “no-op” coordinator will not be able to go further on and the whole system will block. In order to fix this problem it is enough to prune all the deadlocking paths (i.e., finite branches) of the “no-op” coordinator’s AC-Graph. The resulting coordinator’s AC-Graph allows all the CBA-system correct behaviors to be performed. That is, the elimination of deadlocks does correctly preserve all the “non-deadlocking” CBA-system (and CFA-system) behaviors.

**Proposition 3 (Correctness)** Let $T$ be a CFA-system, let $V$ be the corresponding CBA-system, let $\Delta_V$ be the corresponding CBA-system based on the deadlock-free coordinator (i.e., the “no-op” coordinator’s AC-Graph without the deadlock paths) and let $\Phi_T$ be $T$ without the finite paths; then $\Phi_T$ can be CB-Simulated by $\Delta_V$.

**Proof:** By contradiction, let us suppose that $\Phi_T$ is not CB-Simulated by $\Delta_V$. This implies that $t$ labeled with $< T_1, ..., T_i, ..., T_n >$ and $t'$ in $\Phi_T$ exist in such a way that $t \rightarrow t'$ and $v$ in $\Delta_V$ exists in such a way that $t \leq_{CB} v$. Then it does not exist a $v'$ in $\Delta_V$, such that $v \rightarrow^* v'$ (for some $p$) in the transition relation of $\Delta_V$ and $t'$ is CB-simulated by $v'$. Let us suppose that $t'$ is labeled by $< T'_1, ..., T'_i, ..., T'_j, ..., T'_n >$ because of $t \rightarrow t'$ by synchronizing $C_i$ and $C_j$ on actions $\alpha$ and $\alpha$, respectively.

That is $(T_i, ?\alpha, T'_i) \in A_{AC},$ and $(T_j, \alpha, T'_j) \in A_{AC}$. By definition of EX-Graph we have that $(T_i, ?\alpha, T''_i), (T'_i, \alpha, T'''_i) \in A_{EX}$, and $(T_j, ?\alpha, T''_j), (T'_j, \alpha, T'''_j) \in A_{EX}$. By definition of EX-Graphs unification algorithm we have that $(< T_1, ..., T_i, ..., T_j, ..., T_n >, ?\alpha, ?\alpha, < T'_1, ..., T'_i, ..., T'_j, ..., T'_n >)$ are in the set of arcs of the coordinator’s AC-Graph. Since, by hypothesis, $< T_1, ..., T'_1, ..., T'_n >$ is not a sink node, in the transition relation defined for $\Delta_V$ we have that $t = v = < T_1, ..., T'_i, ..., T'_j, ..., T'_n > \rightarrow < T_1, ..., T'_1, ..., T'_j, ..., T'_n > \rightarrow < T_1, ..., T'_1, ..., T'_j, ..., T'_n >$. This is a contradiction thus the proof is given.
From the previous proposition we can state that if the system contains deadlocks, we are able to avoid them in order to obtain a deadlock-free version of the system:

**Corollary 3** Let $T$ be a CFA-system, and let $V$ be the corresponding CBA-system. If a deadlock state labeled with $< S_1, ..., S_i, ..., S_j, ..., S_n >$ exists in $T$, then the “no-op” coordinator of $V$ has a state with the same label.

**Proof:** By Proposition 3, if in $T$ a deadlock state $t =< S_1, ..., S_i, ..., S_j, ..., S_n >$ exists then in $V$ a corresponding state $v$ exists in such a way that the label of $t$ is equal to the label of $v$. To prove this proposition we have to show that $v$ is a sink node of the “no-op” coordinator’s AC-Graph. If $t$ in $T$ represents a deadlock state it means that $t$ represents a scenario in which each component in $T$ is stopped waiting for an action from a different component. That is $\forall i = 1, ..., n \in AC_i$ from $S_i \in NA_{AC}$, it is possible only perform actions $\alpha$ such that it does not exist $j = 1, ..., n \land j \neq i : \alpha \in LA_{AS_i} \land (S_j, \alpha, S'_j) \in A_{AS_i}$ for some $S'_j \in NA_{AS_i}$. Thus by definition of EX-Graph and by definition of EX-Graphs unification algorithm from the state $v$ of the “no-op” coordinator’s AC-Graph (with the same label of $t$) the unification algorithm builds a set $T E R$ of action terms and a set $V A R$ of action variables that do not produce unifiable pairs of action term and action variable. That is $v$ is a sink node.

**Completeness**

To state the completeness of our technique used for synthesizing deadlock-free coordinators, we prove that the deadlock-free CFA-system (i.e., the CFA-system without the finite paths) CB-simulates the corresponding deadlock-free CBA-system which is obtained by introducing in the system the deadlock-free coordinator. Moreover, analogously to what we have done for proving the completeness of the “no-op” coordinator synthesis, in order to not consider the extra-level of indirection introduced by the deadlock-free coordinator we consider a restricted version of the deadlock-free CBA system. This restricted version is obtained by eliminating from the CBA system the extra level of indirection that the deadlock-free coordinator introduces:

**Proposition 4 (Completeness)** Let $T$ be a CFA-system, let $V$ be the corresponding CBA-system, let $\Phi_T$ be $T$ without the finite paths, let $\Delta_V$ be the deadlock-free CBA-system which is obtained by pruning all the possible deadlock paths of the coordinator in $V$, let $\Pi_{\Delta_V}$ be Restrict($\Delta_V$) and let $t_0, v_0$ be the initial states of $\Phi_T$ and $\Pi_{\Delta_V}$, respectively; for all states $t$ reachable from $t_0$ in $\Phi_T$, there exists a reachable state $v$ from $v_0$ in $\Pi_{\Delta_V}$ such that the label of $v$ is equal to the label of $t$.

**Proof:** We give proof by induction on the length of a path in $\Pi_{\Delta_V}$. By construction, the base step (i.e., for a path length equal to $0$) is trivially proved because the state label of $v_0$ is equal to the state label of $t_0$. The inductive step (i.e., for a path length equal to $m > 0$) consists in supposing that a generic state $v$ labeled with $< V_1, ..., V_i, ..., V_k, ..., V_n >$ in $\Pi_{\Delta_V}$ exists. $v$ is such that $v_0 \xrightarrow{p} v$ (for some $m$) and the state label of $v$ is equal to the state label $< T_1, ..., T_i, ..., T_k, ..., T_n >$ of a state $t$ in $\Phi_T$ (i.e., $\forall i = 1, ..., n : V_i = T_i$).

Moreover let us suppose that $t_0 \xrightarrow{q} v$ (for some $p$). Then we have to prove that given a state $v'$ such that $v \xrightarrow{q} v'$, it is possible to find a state $t'$ such that $t \xrightarrow{p} t'$ (for some $p$) and the state label of $t'$ is equal to the state label of $v'$. Let us suppose that $v'$ is labeled by $< V'_1, ..., V'_i, ..., V'_k, ..., V'_n >$ because of $v \xrightarrow{q} v'$ by synchronizing $C[i]$ with the deadlock-free coordinator and, subsequently, the deadlock-free coordinator with $C[k]$ on coordinator’s actions $?I_i$ and $?!k$, respectively.

By definition of EX-Graphs unification algorithm, it means that $\exists i = 1, ..., n \land \exists k = 1, ..., n \land i \neq k : (V_i, \forall I_i, V'_i), (V_i, \forall k, V'_i) \in A_{EX}$ and $(V_k, \forall l, V'_k), (V'_k, \forall k, V'_k) \in A_{EX}$. By definition of EX-Graph, AS-Graph and AC-Graph we have that $(V_i, \forall I_i, V'_i) \in A_{AC_i}$ and $(V_k, \forall l, V'_k) \in A_{AC_k}$.
4.2 Dealing with normalization

In Section 3.1 we said that a component in our architectural style has a notion of both top and bottom interface. A component can request a service provided by another component and can receive a response (i.e., component client side). On the other hand, a component can receive a request for a service it provides and can return a response (i.e., component server side). In a single-layer system (e.g., a client-server application in COM/DCOM) a component that declares only its top interface is seen as a client component. Analogously, a component that declares only its bottom interface is seen as a server component. In the case of a multi layered system, it is possible to have components that are both servers and clients. Those components declare both a top and a bottom interface.

In Section 3.3, we specify the behavior of a component in terms of the CCS process representing the sequences of messages exchanged with its environment. Referring to the notions of top interface and bottom interface, we can separate the behavior of a component within the hierarchy of a multi layered system in two behaviors: i) top component behavior and ii) bottom component behavior. The former is the component behavior representing only the sequences of top interface’s messages exchanged with its environment. Analogously, the latter is the component behavior representing only the sequences of bottom interface’s messages exchanged with the environment. In Figure 4.10 we show how to decompose a n-layer system into n single layered (sub-)systems in the case of n = 2.

Under the constraints of our architectural style and by assuming we are dealing with multi-layered systems that are built by imposing a-cyclic architectural configurations, we are able to decompose component behavior in order to make the layers of a multi-layered systems completely independent to each other. In

\[ \text{definition of CFA we have that } (V_1, ..., V_k, ..., V_n >, \tau, V_1, ..., V_k', ..., V_n >) \text{ is in the transition relation defined for } \Phi_T. \]

Thus \( v = t < V_1, ..., V_k, ..., V_n > \rightarrow < V_1, ..., V_k', ..., V_n > \) is in the transition relation defined for \( \Phi_T \). By hypothesis, the state label of \( v' \) is equal to the state label of \( v \) and both \( v' \) and \( v \) are reachable from \( t_0 \) and \( v_0 \) respectively. We have verified the inductive step by construction, and thus the proof is complete.

**Corollary 4** Let \( V \) be a CBA-system, let \( \Delta_V \) be the corresponding deadlock-free CBA-system, let \( \Pi_{\Delta_V} \) be \( \text{Restrict}(\Delta_V) \), let \( T \) be the CFA-system corresponding to \( \Phi_T \). Let \( \Phi_T \) be the CBA-system \( \Pi_{\Delta_V} \) corresponding to \( V \). If \( v_0, t_0 \) be the initial states of \( \Pi_{\Delta_V} \) and \( \Phi_T \) respectively, then \( \Pi_{\Delta_V} \) can be CB-simulated by \( \Phi_T \).

**Proof:** The proof is trivially derived from the definition of CB-Simulation and by Proposition 4. Actually, by Proposition 4, for each state \( v \) in \( \Pi_{\Delta_V} \) reachable from \( v_0 \) it exists a state \( t \) in \( \Phi_T \) reachable from \( t_0 \) in such a way that the state label of \( v \) is equal to the state label of \( t \). If we apply Proposition 4 by starting from \( v_0 \) we obtain again the CB-Simulation definition for each state of \( \Pi_{\Delta_V} \) with a state of \( \Phi_T \). Thus we obtain that for each state \( v \) of \( \Pi_{\Delta_V} \) such that the state label of \( v \) is equal to the state label of some \( t \) in \( \Phi_T \), if \( v \rightarrow v' \) then there exist \( n > 0 \), \( t_0, ..., t_n \) such that \( t = t_0 \) and for all \( i < n \) : \( t_i \rightarrow t_{i+1}, v' \leq v \). That is \( \Phi_T \) CB-simulates \( \Pi_{\Delta_V} \).

It is worthwhile noticing that by Corollaries 3 and 4 and by considering that the only new logic the coordinator adds to the composed system is the extra level of indirection, we have that the deadlock-free CFA-system \( \Phi_T \) and the restricted deadlock-free CBA-system \( \Pi_{\Delta_V} \) are CB-equivalent. That is \( \Phi_T \) and \( \Pi_{\Delta_V} \) CB-simulate each other.
this way, we reduce a multi-layered system to a set of single layered subsystems for each one of them we build a coordinator. In fact by exploiting our architectural style and the above mentioned assumptions, let \( C_i \) be a component in a intermediate layer of a multi-layered system, we can always derive a partition of the actions performed by \( C_i \) in two disjoint sets of actions. These two disjoint action sets are related to the actions of the top interface and of the bottom interface of \( C_i \), respectively. Otherwise, referring to how we model a system (see Definitions 1 and 2), \( C_i \) might not be an intermediate component for all components above and below it. That is, the two set \( \{C_j\} \) and \( \{C_k\} \) of components above and below \( C_i \) (respectively) might have components that directly synchronize with components of the other set. This, in turn, implies that the system formed by \( C_i \) the components in \( \{C_j\} \) and in \( \{C_k\} \) would not be a multi-layered system (i.e., for instance a three-layered system with \( C_i \) in the intermediate layer) but it would be a single-layered system where \( C_i \) and the components in \( \{C_j\} \) and in \( \{C_k\} \) might be directly connected (through connectors) to each other.

Informally, let \( LA_{AC_i} \) be the set of action labels of the AC-Graph of a component \( C_i \). Let 
\[ TopInterface(C_i) \]
be the set of all actions in \( LA_{AC_i} \) that are also actions of the top interface of \( C_i \). From \( AC_i \), we derive the BAC-Graph (Bottom interface ACtual behavior Graph) of \( C_i \) by collapsing (in a unique state) the “from” and the “to” state of all transitions labeled with actions in \( TopInterface(C_i) \). Finally we eliminate these transitions. In this way the BAC-Graph of \( C_i \) identifies the bottom component behavior of \( C_i \). Analogously we derives the TAC-Graph (Top interface ACtual behavior Graph) of a component \( C_i \) from its AC-Graph \( AC_i \). This is done by collapsing (in a unique state) the “from” and the “to” state of all transitions labeled with actions of the bottom interface of \( C_i \) (i.e., all actions in \( BottomInterface(C_i) \)) and by eliminating these transitions. In this way the TAC-Graph of \( C_i \) identifies the top component behavior of \( C_i \). It is worthwhile noticing that BAC-Graph and TAC-Graph are always AC-Graphs. Then considering the definitions given in Section 4.1.2 we can obtain: BAS-Graph (Bottom interface ASSumption Graph), BEX-Graph (Bottom interface EXpected Graph), TAS-Graph (Top interface ASSumption Graph) and TEX-Graph (Top interface EXpected Graph). In the reminder of this section, we give the formal definitions of the algorithms used for automatically constructing BAC-Graph and TAC-Graph. To give these formal definitions we have to present first the following:

**Definition 15 (BottomInterface(\(C_i\)))** Let \( \langle N_{AC_i}, LN_{AC_i}, A_{AC_i}, LA_{AC_i}, S(0) \rangle \) be the AC-Graph \( AC_i \) of a component \( C_i \) and for all \( k \neq i \) let \( \langle N_{AC_k}, LN_{AC_k}, A_{AC_k}, LA_{AC_k}, S(0) \rangle \) be the AC-Graph \( AC_k \) of a component \( C_k \) below \( C_i \), then the bottom interface of \( C_i \) (BottomInterface(\(C_i\))) is the set of actions \( \{\alpha \in LA_{AC_i} \) such that \( \alpha \in LA_{AC_k} \) for some \( k \).
4.2 Dealing with normalization

Definition 16 (TopInterface(Ci)) Let \( \langle N_{AC_i}, LN_{AC_i}, A_{AC_i}, LA_{AC_i}, S0 \rangle \) be the AC-Graph \( AC_i \) of a component \( C_i \), and for all \( k \neq i \) let \( \langle N_{AC_k}, LN_{AC_k}, A_{AC_k}, LA_{AC_k}, S0 \rangle \) be the AC-Graph \( AC_k \) of a component \( C_k \) above \( C_i \), then the top interface of \( C_i \) (TopInterface(\( C_i \))) is the set of actions \( \{ \alpha \in LA_{AC_i} \text{ such that } \overline{\alpha} \in LA_{AC_k} \text{ for some } k \} \).

Definition 17 (TAC-Graph construction algorithm) Let \( \langle N_{AC_i}, LN_{AC_i}, A_{AC_i}, LA_{AC_i}, S0 \rangle \) be the AC-Graph \( AC_i \) of a component \( C_i \), then we derive the corresponding Top interface ACtual behavior (TAC) Graph \( TAC_i \) as follows:

1. at the beginning \( TAC_i \) is equal to \( \langle N_{TAC_i}, LN_{TAC_i}, A_{TAC_i}, LA_{TAC_i}, S0 \rangle \) where \( N_{TAC_i} = N_{AC_i}, LN_{TAC_i} = LN_{AC_i}, A_{TAC_i} = A_{AC_i}, \) and \( LA_{TAC_i} = LA_{AC_i} \);
2. for each loop \( (v, \alpha, v) \in A_{TAC_i} \), where \( \alpha \in BottomInterface(C_i) \) do:
   • remove \( (v, \alpha, v) \) from \( A_{TAC_i} \);
3. for each arc \( (v, \beta, \mu) \in A_{TAC_i} \), where \( \beta \in BottomInterface(C_i) \) do:
   • remove \( (v, \beta, \mu) \) from \( A_{TAC_i} \);
   • if \( \mu \) is equal to \( S0 \) then: set \( v \) as the starting state \( S0 \) and the label of \( v \) as the label \( S0 \);
   • for each other arc \( (v, \gamma, \mu) \in A_{TAC_i} \), do: replace \( (v, \gamma, \mu) \) with \( (v, \beta, \mu) \);
   • for each arc \( (\mu, \delta, \nu) \in A_{TAC_i} \), do: replace \( (\mu, \delta, \nu) \) with \( (\nu, \beta, \mu) \);
   • for each arc \( (\mu, \epsilon, \nu) \in A_{TAC_i} \), with \( \nu \neq \mu, \nu \) do: replace \( (\mu, \epsilon, \nu) \) with \( (\nu, \epsilon, \mu) \);
   • for each arc \( (v, \zeta, \mu) \in A_{TAC_i} \), with \( \nu \neq \mu, \nu \) do: replace \( (v, \zeta, \mu) \) with \( (v, \zeta, \nu) \);
   • for each loop \( (\mu, \eta, \mu) \in A_{TAC_i} \), do: replace \( (\mu, \eta, \mu) \) with \( (v, \eta, \nu) \);
   • remove \( \mu \) from \( N_{TAC_i} \);
4. until \( TAC_i \) does not contain arcs labeled with the same action and outgoing the same state do:
   • for each pair of loops \( (v, \iota, v) \) and \( (v, \iota, \nu) \in A_{TAC_i} \), do:
     • remove \( (v, \iota, \nu) \) from \( A_{TAC_i} \);
   • for each pair of arcs \( (v, \kappa, \mu) \) and \( (\nu, \kappa, \mu) \in A_{TAC_i} \), do:
     • remove \( (v, \kappa, \mu) \) from \( A_{TAC_i} \);
   • for each pair of arcs \( ((v, \lambda, \mu) \text{ and } (v, \lambda, \nu)) \text{ or } ((\nu, \lambda, v) \text{ and } (v, \lambda, \nu)) \in A_{TAC_i} \), do:
     • remove \( (v, \lambda, \nu) \) from \( A_{TAC_i} \);
     • if \( v \) is equal to \( S0 \) then: set \( v \) as the starting state \( S0 \) and the label of \( v \) as the label \( S0 \);
     • for each ingoing arc \( \iota \) in \( v \), outgoing arc \( \iota \) out of \( v \) and loop \( l \) on \( v \) do: move the extremity on \( v \) of \( \iota \), out and \( l \) on \( v \);
     • remove \( v \) from \( N_{TAC_i} \).

Definition 18 (BAC-Graph construction algorithm) Let \( \langle N_{AC_i}, LN_{AC_i}, A_{AC_i}, LA_{AC_i}, S0 \rangle \) be the AC-Graph \( AC_i \) of a component \( C_i \), then we derive the corresponding Bottom interface ACtual behavior (BAC) Graph \( BAC_i \) as follows:

1. at the beginning \( BAC_i \) is equal to \( \langle N_{BAC_i}, LN_{BAC_i}, A_{BAC_i}, LA_{BAC_i}, S0 \rangle \) where \( N_{BAC_i} = N_{AC_i}, LN_{BAC_i} = LN_{AC_i}, A_{BAC_i} = A_{AC_i}, \) and \( LA_{BAC_i} = LA_{AC_i} \);
2. for each loop \( (v, \alpha, v) \in A_{BAC_i} \), where \( \alpha \in TopInterface(C_i) \) do:
   • remove \( (v, \alpha, v) \) from \( A_{BAC_i} \);
3. for each arc \((\nu, \beta, \mu) \in A_{BAC}\), where \(\beta \in \text{TopInterface}(C_i)\) do:
   - remove \((\nu, \beta, \mu)\) from \(A_{BAC}\);
   - if \(\mu\) is equal to \(S_0\) then: set \(\nu\) as the starting state \(S_0\) and the label of \(\nu\) as the label \(S_0\);
   - for each other arc \((\nu, \gamma, \mu) \in A_{BAC}\) do: replace \((\nu, \gamma, \mu)\) with \((\nu, \beta, \nu)\);
   - for each arc \((\mu, \delta, \nu) \in A_{BAC}\) do: replace \((\mu, \delta, \nu)\) with \((\nu, \delta, \nu)\);
   - for each arc \((\mu, \epsilon, \nu) \in A_{BAC}\), with \(\nu \neq \mu, \nu\) do: replace \((\mu, \epsilon, \nu)\) with \((\nu, \epsilon, \nu)\);
   - for each arc \((v, \zeta, \mu) \in A_{BAC}\), with \(v \neq \mu, V\) do: replace \((v, \zeta, \mu)\) with \((v, \zeta, v)\);
   - for each loop \((\mu, \eta, \mu) \in A_{BAC}\), do: replace \((\mu, \eta, \mu)\) with \((\nu, \eta, \nu)\);
   - remove \(\mu\) from \(N_{BAC}\).

4. until \(BAC_i\) does not contain arcs labeled with the same action and outgoing the same state do:
   - for each pair of loops \((\nu, i, \nu)\) and \((\nu, i, \nu) \in A_{BAC}\), do:
     - remove \((\nu, i, \nu)\) from \(A_{BAC}\);
   - for each pair of arcs \((\nu, \kappa, \mu)\) and \((\nu, \kappa, \mu) \in A_{BAC}\) do:
     - remove \((\nu, \kappa, \mu)\) from \(A_{BAC}\);
   - for each pair of arcs \((\nu, \lambda, \mu)\) and \((\nu, \lambda, \nu) \in A_{BAC}\) do:
     - remove \((\nu, \lambda, \nu)\) from \(A_{BAC}\);
   - if \(\nu\) is equal to \(S_0\) then: set \(\nu\) as the starting state \(S_0\) and the label of \(\nu\) as the label \(S_0\);
   - for each ingoing arc \(in\) in \(v\), outgoing arc \(out\) from \(v\) and loop \(l\) on \(v\) do: move the extremity on \(v\) of \(in\), \(out\) and \(l\) on \(v\);
   - remove \(v\) from \(N_{BAC}\).

Informally, the algorithms of Definition 17 and 18 “collaps” (steps 1.2 and 3) linear and/or cyclic paths made only of actions of the component’s bottom and top interface, respectively. Moreover, they also avoid (step 4) possible “redundant” non-deterministic behaviors$^5$.

The \(n\)-layer systems decomposition presented in this section allow us to show that it is enough to consider a single-layer system in order to fully define and formalize our approach.

### 4.3 Dealing with Signature-level Mismatches

In this section we formalize an extension of the algorithm discussed in Section 4.1. We make this extension in order to deal with components that might have incompatible interfaces at level of methods signature \cite{76}. Thus, by referring to Section 2.1.1, we do that for a restricted class of incompatibilities at the level of syntax rules of interaction. For instance, the problem we want to address can be phrased as follows: given a set of components \(C\) that might have incompatible interface’s signatures at level of method names, automatically derive an assembly \(A\) of these components which is deadlock-free.

By referring to Section 4.2, we recall that without loss of generality it is enough to formalize our approach only for single-layer systems. In fact by exploiting the architectural decomposition discussed in Section 4.2, we can also apply the formalized approach to multi-layered systems. Within the problem definition, when we say that the considered components “might have incompatible interface’s signatures at level of method names” we mean that the set of required (provided) interfaces of a component in the bottom (top) side of the (single-layer) system’s architecture is different, with respect to method names, from the set of provided (required) interfaces of a component in the top (bottom) side.

For the sake of simplicity, in the reminder of this thesis we will simply refer to incompatible interfaces to identify incompatible interface signatures at the level of method names.

$^5$These behaviors might be a side effect due to the collapsing.
4.3 Dealing with signature-level mismatches

4.3.1 Method description

Informally our approach is the following. The method starts with a set of black-box components, and builds a coordinator that bridges possible incompatibilities at level of interface method names. Then, analogously to what we done in Section 4.1, deadlocks analysis is performed. If the synthesized coordinator contains deadlocks, a recovery technique is applied in order to obtain its corresponding deadlock-free coordinator.

4.3.2 Method formalization

In this section, we formalize the extension of the algorithm discussed in Section 4.1 that we made in order to deal with components that might have incompatible interfaces.

The starting point of this extension is a revisited definition of EX-Graph that takes into account the possibility to have components with incompatible interfaces. That in our context is, we deal with components that might perform a request or a notification that cannot be received by the other components that have to form the composed system. We can refine the Definition 6 to obtain a new definition of EX-Graph. We recall that the coordinator is only responsible for routing messages and it exhibits a strictly sequential input-output behavior. Moreover, since we want to deal with components that may have incompatible interfaces, the coordinator has to be able to map actions in the alphabet of a component AC-Graph to actions in the alphabet of a different component AC-Graph. Therefore, if the coordinator receives an input message \( m \) from a component \( C_i \) (i.e., \( C_i \) performs \( !m \) which corresponds to \(?m, i\) within \( EX_i \)), in order to not block \( C_i \), it will then output \( m \) to a destination component \( C_j \) which performs \(?m\). Since \( C_j \) is unknown to \( C_i \), the coordinator will output \( m \) to a destination component which is unknown to \( C_i \) (i.e., this corresponds to \(?m, \# \) within \( EX_i \)). Analogously to what we done for Definition 6, within \( EX_i \), we denote a message \( m \) which is sent towards an unknown component as \( !m, \# \). Otherwise, in the case of no component (different from \( C_i \)) performing the input of \( m \) exists, the coordinator will output a message \( m' \), which is "mapped" to \( m \), to the destination component \( C_k \) which performs \(?m'\). Since both \( m' \) and \( C_k \) are unknown to \( C_i \), the coordinator will output a message, which is unknown to \( C_i \), to a destination component which is unknown to \( C_i \) as well. Moreover this unknown message has been mapped to \( m \) in order to achieve interface’s signature compatibility. Within \( EX_i \), we denote an unknown message which is mapped to a known one \( m \) and which is sent towards an unknown component as \( !\#, \#(m, i) \). Thus, we extend Definition 6 to deal with components that might have incompatible interfaces as follows:

Definition 19 (EX-Graph construction algorithm for signature level mismatching components)

Let \((N_{AC_i}, L_{AC_i}, A_{AC_i}, LA_{AC_i}, S_0)\) be the AS-Graph \( AS_i \) of a component \( C_i \); we define the coordinator EXpected (EX) Graph \( EX_i \) of the component \( C_i \) the graph \((N_{EX_i}, L_{EX_i}, A_{EX_i}, LA_{EX_i}, S_0)\) which is built by performing the following algorithm:

- at the beginning \( N_{EX_i} = N_{AS_i} \) and \( L_{EX_i} = L_{AS_i} \);
- at the beginning \( A_{EX_i} \) and \( LA_{EX_i} \) are empty;
- for all \((\mu, \alpha, \mu') \in A_{AS_i} \), do:
  - create a new node \( \mu_{\text{new}} \) with a new unique label, add the node to \( N_{EX_i} \) and the unique label to \( L_{EX_i} \);
  - if \((\mu, \alpha, \mu') \) is such that \( \alpha \) is an input action (i.e., \( \alpha = !a \), for some \( a \) ) then:
    * if \( j \neq i \) in such a way that \(?a_\# \) there exist then set the label \( \text{lbl} \) to \(?a_\#, i\) else set the label \( \text{lbl} \) to \(?a_\#(a, i)\);
    * add the labels \( \text{lbl} \) and \( \text{ lbl } \# \) to \( LA_{EX_i} \);
    * add \((\mu, \text{ lbl }, \mu_{\text{new}}) \) and \((\mu_{\text{new}}, \text{ lbl }, \mu') \) to \( A_{EX_i} \);
- if \((\mu, \alpha, \mu')\) is such that \(\alpha\) is an output action (i.e. \(\alpha = ?a, \text{for some } a\)) then:
  * if \(j \neq i\) in such a way that \(?a \in LA_{AS_j}\) there exist then set the label \(\mathbf{lb}l\) to \(?a, \#(a, i)\);
  * add the labels \(?a, \#\) and \(\mathbf{lb}l\) to \(LA_{EX_i}\);
  * add \((\mu, ?a, \mu_{new})\) and \((\mu_{new}, \mathbf{lb}l, \mu')\) to \(AE_{EX_i}\).

By referring to our definition of CBA (see Definition 2), within \(EX_i\), action \(?a, \#(a, i)\) denotes a coordinator’s input (output) action \(a\) that synchronizes with an output (input) action \(a\) of \(C_i[f_i]\). We denote these actions as “known actions”. Action \(?a, \#(a, \#)\) denotes a coordinator’s input (output) action \(a\) that synchronizes with an output (input) action \(a\) of \(C_j[f_j]\) for some \(j \neq i\); thus the component \(C_j\) is unknown to \(C_i\) (we denote unknown information by using the symbol ‘\#’). We denote these actions as “unknown actions”. Action \(?\#(a, i), \#(a, i)\) denotes a coordinator’s input (output) action \(b\) (for some \(b \in LA_{AC_j}\)) that synchronizes with an output (input) action \(b\) of \(C_k[f_k]\) for some \(k \neq i\) which is mapped to the action \(a\) performed by \(C_i\); thus both the action \(b\) and the component \(C_k\) are unknown to \(C_i\). We denote these actions as “mismatching unknown actions”. While “known” and “unknown actions” are also used in Definition 6, “mismatching unknown actions” represent the novel aspect of the EX-Graph definition we give above.

We derive the coordinator AC-Graph through an EX-Graph unification algorithm which is a simple extension of the one formalized by means of Definition 11. We recall that the algorithm of Definition 11 is informally the following: for each step the unification procedure attempts to match “known actions” (i.e., terms) in a EX-Graph with “unknown actions” (i.e., variables) in a different EX-Graph; each match represents a new transition (in the coordinator graph) from the current node to the next new (i.e., not yet considered) adjacent node; then the algorithm proceeds in the unification procedure from each adjacent node. For our purposes we need to modify this unification algorithm to deal with components that might have incompatible interfaces. In other words, we need to extend the algorithm of Definition 11 in order to deal with “mismatching unknown actions”. Informally, for each step the extended unification procedure attempts to match “known actions” (i.e., terms) in a EX-Graph with “unknown” and “mismatching unknown actions” (i.e., variables) in another EX-Graph. When in a generic step the unification procedure attempts to match terms in \(EX_j\) with variables in \(EX_i\) (for each \(j \neq i\)), first it tries to match a term with a variable which denotes an “unknown” action (analogously to what we done for Definition 11); second, if and only if the unification procedure did not yet found successful matches for that term, it attempts to match that term with a variable which denotes a “mismatching unknown action” (this is the extension of the old unification algorithm we make to deal with incompatible interfaces). It is worthwhile noting that in a generic step it may be possible to find one “mismatching unknown action” that matches with two or more different terms and vice versa. In these cases our tool SYNTHESIS queries the user in order to know what is the right match since it cannot be clearly established in an automatic way. Moreover, note also that in order to deal with incompatible interfaces the user might want to map a “known action” with an “unknown” one although they are not unifiable. Thus, for each not-unifiable pair of “known” and “unknown” actions SYNTHESIS queries the user in order to know if he wants to map the “known” action to the “unknown” one:

**Definition 20 (EX-Graph unification algorithm for signature level mismatching components)**

- Let \(AC_1, ..., AC_n\) be the AC-Graphs of the components \(C_1, ..., C_n\) forming the CFA-version of the composed system;
- Let \(AS_1, ..., AS_n\) be the AS-Graphs corresponding to \(AC_1, ..., AC_n\) respectively;
- Let \(EX_1, ..., EX_n\) be the EX-Graphs corresponding to \(AS_1, ..., AS_n\) respectively;
- Let \(S_1, ..., S_n\) be the current states of \(EX_1, ..., EX_n\) respectively.

At the beginning the current states are the initial states of \(EX_1, ..., EX_n\).
4.3 Dealing with signature-level mismatches

1. Create the AC-Graph of the “no-op” coordinator, with one node (initial state) and no arcs.

2. Set as current states of the components EX-Graphs the respective initial states.

3. Label the initial state of the “no-op” coordinator AC-Graph with an ordered tuple composed of the initial states of all components EX-Graphs. For the sake of presentation we assume to order them so that the \( j \)-th element of the state label corresponds to the current state of the component EX-Graph \( EX_j \) where \( j \in [1, \ldots, n] \). This state is the coordinator current state \( g \).

4. Perform the following unification procedure \textbf{Unify}(g):

   (a) Mark \( g \) as visited.

   (b) Let \( < S_1, \ldots, S_n > \) be the state label of \( g \).

   (c) Generate the set \( TER \) of action terms and the set \( VAR \) of action variables so that \((t, i) \in TER\), if in \( EX_i \) we have \( S_i \xrightarrow{t} V_i \). Similarly \((v, j) \in VAR\), if in \( EX_j \) we have \( S_j \xrightarrow{v} V_j \).

   (d) For all pairs \((t, i), (v, j)\) do:

       i. if \((t, i), (v, j)\) is unifiable or (although it is not unifiable) the user answers to the query that he wants to map \( t \) to \( v \) then,

       if the two new nodes \( g_i, g_j \) with state label \( < S_1, \ldots, V_i, \ldots, V_j, \ldots, S_n > \) and

       \( < S_1, \ldots, S_i', \ldots, S_j', \ldots, S_n > \) respectively, where \( S_i \xrightarrow{t} S_i' \) in \( AS_i \) and

       \( S_j \xrightarrow{v} S_j' \) in \( AS_j \), do not already exist (in the “no-op” coordinator AC-Graph) then

       A. create \( g_i \) and \( g_j \);

       B. create the arc \((g, t \xrightarrow{2}, g_i)\) in the “no-op” coordinator AC-Graph;

       C. mark \( g_i \) as visited;

       D. create the arc \((g_i, v \xrightarrow{3}, g_j)\) in the “no-op” coordinator AC-Graph.

   (e) If \( VAR \) is empty then generate the set \( VAR' \) of action variables so that \((v, j) \in VAR'\) if in \( EX_j \) we have \( S_j \xrightarrow{v} V_j \).

   (f) For all pairs \((t, i), (v, j)\) do:

       i. if the user answers to the query that he wants to map \( t \) to \( v \) then,

       if the two new nodes \( g_i, g_j \) with state label \( < S_1, \ldots, V_i, \ldots, V_j, \ldots, S_n > \) and

       \( < S_1, \ldots, S_i', \ldots, S_j', \ldots, S_n > \) respectively, where \( S_i \xrightarrow{t} S_i' \) in \( AS_i \) and

       \( S_j \xrightarrow{v} S_j' \) in \( AS_j \), do not already exist (in the “no-op” coordinator AC-Graph) then

       A. create \( g_i \) and \( g_j \);

       B. create the arc \((g, t \xrightarrow{2}, g_i)\) in the “no-op” coordinator AC-Graph;

       C. mark \( g_i \) as visited;

       D. create the arc \((g_i, v \xrightarrow{3}, g_j)\) in the “no-op” coordinator AC-Graph.

   (g) Perform recursively \textbf{Unify}(g") for all not marked (as visited) adjacent nodes \( g" \) of current node \( g \).

Unlike the algorithm formalized by means of Definition 11, the one formalized by the previous definition is not completely automatic since it requires user interaction. This is the price in order to deal with incompatible interfaces at signature level. Once the previous algorithm is performed, we have semi-automatically obtained the AC-Graph of the “no-op” coordinator which bridges component interface mismatches at methods signature level. The deadlock detection and recovery step is the same as it was discussed in Section 4.1.3.

For the sake of simplicity, in the reminder of this thesis we will simply take into account only components with compatible interface signatures by considering that in the opposite case we can exploit the algorithm formalized in this section instead of the one formalized in Section 4.1.2.
4.4 AUTOMATIC SYNTHESIS OF FAILURE-FREE ADAPTORS

Our approach also considers the analysis of failures beyond deadlock. We specify these failures by specifying precise ways to coordinate the components forming the composed system. We do it by specifying all the composed system’s desired behaviors. Each desired behavior represents a coordination policy given in terms of a Büchi Automaton (see Section 3.4).

In this section we formalize an extension of the algorithm discussed in Section 4.3. This extension is concerned with the enforcing of the coordination policies against the deadlock-free coordinator’s AC-Graph. Moreover, by continuing our application example, we also apply our technique to the synthesized deadlock-free coordinator of Figure 4.9.

4.4.1 METHOD DESCRIPTION

The problem we want to treat in this section can be phrased as follows: Given a CFA system which is formed by a set of components \( C \); given a set \( P \) of coordination policies that describe precise ways to coordinate the interaction behavior of the components in \( C \), automatically derive a deadlock-free assembly \( A \) of these components which performs only policies in \( P \), if possible.

As illustrated in Figure 5.2 we proceed in three steps. The first step starts with a CFA system and automatically produces a new configuration with the same components plus a “no-op” coordinator which filters all the connections among the components obeying to our CBA style (see Section 4.1). If it is required, that “no-op” coordinator might also bridge possible interface signature mismatches (see Section 4.3).

The second step concerns the deadlock-freeness analysis, which is performed on the CBA system and that allows for deadlock detection. Subsequently, we can operate on the “no-op” coordinator in order to obtain a deadlock-free equivalent system (see Section 4.1.3).

The third step, which is the novel contribution of this section with respect to the previous ones, concerns the enforcing of the coordination policies against the model of the deadlock-free coordinator. This step produces the policy-satisfying coordinator representing the assembly code for the components forming the composed system.

4.4.2 METHOD FORMALIZATION

The coordination policies that must be enforced are related to behaviors of the CFA-system. The CFA-system’s behaviors that do not behave as specified by the coordination policies represent behavioral failures of the CFA-system. As proved in Section 4.1.4, since the synthesized deadlock-free CBA-system is behaviorally equivalent to the deadlock-free CFA-system except for an extra level of indirection (which is introduced by the coordinator), these behavioral failures are also related to precise behaviors of the CBA system. Thus in specifying the coordination policies we are related to the CFA-system, on the other hand in constraining the deadlock-free coordinator to perform only the specified policies (i.e., failures recovery) we are related to the CBA-system. Analogously to deadlock, we cannot recovery behavioral failures of the CBA-system that are not identifiable with precise behaviors of the synthesized deadlock-free coordinator. A coordinator behavior is simply a sequence of states (i.e., a path) within the coordinator’s AC-Graph. Thus the coordination policies we deal with are behaviors that correspond to precise sequences of coordinator’s AC-Graph states. Since each coordinator’s AC-Graph state is a tuple of components EX-Graphs states (and hence, by ignoring the extra-level of indirection, of components AC-Graphs states), this makes sense with our need to specify a coordination policy by looking to the CFA-system.
We recall that each coordination policy is specified in terms of a Büchi Automaton. Referring to Section 3.4, a Büchi Automaton is defined by giving a set $A$ of actions. Let $C_1, ..., C_n$ be the components forming the CFA-system; let $AC_1, ..., AC_n$ be the corresponding AC-Graphs; let $f_1, ..., f_n$ be relabeling functions such that $\forall \alpha \in \bigcup_{i=1}^{n} LA_{AC_i}, \forall j = 1, ..., n : (f_j(\alpha) = \alpha_j)$ and let $Negation$ be a relabeling function such that $\forall \alpha \in \bigcup_{i=1}^{n} LA_{AC_i} : Negation(\alpha) = ((? = ? - \alpha = ?\beta) \text{ or } (! = \alpha = !\beta))$ then we define the set $A$ of actions as follows:

$$A = (\bigcup_{i=1}^{n} LA_{C_1}[f_i] \cup \bigcup_{i=1}^{n} LA_{C_1}[f_i][Negation] \cup \{true\_\}) \text{ where } LA_{C_1}[f_i] \text{ is the set } \{f_i(\alpha) : \alpha \in LA_{C_i}\} \text{ and } LA_{C_1}[f_i][Negation] \text{ is the set } \{Negation(f_i(\alpha)) : \alpha \in LA_{C_i}\}.$$

By continuing the example discussed in Section 4.1.1, the following is the resulting actions set $A$:

$$A = \{fork_1, _1, leave_1, _1, fork_2, _2, leave_2, _2, fork_3, _3, ok1_3, ok2_3, leave1_3, leave2_3, leave3_3, leave4_3, ok1_4, ok2_4, leave1_4, leave2_4, leave3_4, leave4_4, ? - fork1_1, ! - fork2_2, ! - fork3_3, ? - ok1_1, ! - ok2_2, ? - leave1_1, ? - fork1_2, ! - fork2_2, ? - leave2_2, ! - fork3_3, ? - ok1_3, ! - ok2_4, ! - leave1_3, ! - leave2_3, ! - fork1_4, ? - ok1_4, ! - leave1_4, ! - fork2_4, ? - ok2_4, ! - leave2_4, true\_\}.$$

The syntax and semantics of the action labels in $A$ is quite similar to the syntax and semantics of the action labels in an AC-Graph except two kinds of action: i) a universal action (i.e., $true\_$) which represents any possible action in $\bigcup_{i=1}^{n} LA_{C_1}[f_i]$, and ii) a negative action which represents any possible action in $\bigcup_{i=1}^{n} LA_{C_1}[f_i]$ different from the same negative action; for example the negative action $! - fork2_4$ matches all the actions in $\bigcup_{i=1}^{n} LA_{C_1}[f_i]$ different from $!fork2_4$. Moreover, differently from actions in an AC-Graph, each action in $A$ has associated an identifier specifying what component performs that action. For example $!fork2_4$ matches the output action $fork2$ performed by the component $C4$ that is the “Philosopher2” functional component.
In Figure 4.12, we report SYNTHESES’s screen-shots that show possible coordination policies (expressed in terms of Büchi Automata) for our explanatory example.

Each node is a state of the CFA-system. The node with the incoming arrow is the initial state. The filled nodes are the states accepting the system behavior that the Büchi Automaton specifies. That is, in these states that behavior has been just accomplished. We refer to Section 3.4 for a formal definition of "accepting execution" of a Büchi Automaton.

"AlternatingProtocol" and "NoFork2First" coordination policies (showed on the top and on the bottom of Figure 4.12 respectively) specify behaviors of the CFA-system that guarantee the progress of all components.

The "AlternatingProtocol" coordination policy specifies that Philosopher1 (i.e., $C_3$) and Philosopher2 (i.e., $C_4$) can request the two forks by necessarily using an alternating coordination protocol. In other words, it means that if Philosopher1 has requested for both the two forks then Philosopher1 cannot requests for the two forks again if Philosopher2 has not requested for both the two forks and vice versa. Moreover "AlternatingProtocol" specifies that the alternating protocol used for requesting the two forks is activated only once the first fork has been requested; otherwise (i.e., the first fork has not been requested) it might be not activated.
The “NoFork2First” coordination policy (shown on the bottom of Figure 4.12) specifies that a philosopher cannot request the second fork if it has not still requested the first fork.

“AlternatingProtocol” and “NoFork2First” together constrain the philosophers to use an alternating coordination protocol for requesting the two forks; moreover they constrain each philosopher to first request the first fork (i.e., fork1 method call) and then the second one (i.e., fork2 method call). The coordinator that must be synthesized will avoid starvation by implementing both “AlternatingProtocol” and “NoFork2First” as component interaction protocols.

Now, by continuing our application example we both formalize the coordination enforcing technique and show an its application forcing “AlternatingProtocol” and “NoFork2First” against the model of the coordinator showed in Figure 4.9.

**ALGORITHM FOR ENFORCING A COORDINATION POLICY**

Let $p_j$ be the Büchi Automaton of a specified coordination policy. We recall that when we specify $p_j$ we are related to the CFA system but in constraining the coordinator behavior to perform only $p_j$ we are related to the CBA system. Thus, given $p_j$, we derive $p_j^{CBA}$ that is a CBA version of $p_j$. $p_j^{CBA}$ is trivially derived from $p_j$ by applying the CCS “complement” operator on all action labels of $p_j$:

**Definition 21 (CBA version of a Büchi Automaton)** Let $< S, A, \Delta, q_0, F >$ be a Büchi Automaton $p_j$, then the CBA version of $p_j$ is the Büchi Automaton $p_j^{CBA} = < S', A', \Delta', q_0, F' >$ where $S = S'$, $A' = \{ \overline{\alpha} \}$ such that $\alpha \in A$ and $\alpha \neq \text{?true}_-$ $\cup \{ \text{?true}_- \}$, $\Delta' = \{ \langle q_0, \overline{\alpha}, q_k \rangle \text{ such that } \langle q_k, \alpha, q_k \rangle \in \Delta \text{ and } \alpha \neq \text{?true}_- \} \cup \{ \langle q_s, \text{?true}_-, q_t \rangle \text{ such that } \langle q_s, \text{?true}_-, q_t \rangle \in \Delta \}$ and $F' = F$.

By referring to the automata-based model checking approach [21], to perform the enforcing of $p_j^{CBA}$ against the deadlock-free coordinator’s AC-Graph $K$ we need to automatically derive the Büchi Automaton corresponding to $K$. We can derive the Büchi Automaton $B_K$ corresponding to $K$ by exploiting the following definition:

**Definition 22 (Büchi Automaton corresponding to an AC-Graph)** Let $(N_{AC}, L_{AC}, A_{AC}, L_{A_{AC}}, S_0)$ be an AC-Graph $AC$, then the Büchi Automaton corresponding to $AC$ is the Büchi Automaton $B_{AC} = (N_{AC}, L_{AC}, A_{AC}, L_{A_{AC}}, S_0, N_{AC})$.

In our application example, let $K = (N_K, L_{N_K}, A_K, L_{A_K}, S_0)$ be the deadlock-free coordinator’s AC-Graph then $B_K = (N_K, L_{A_K}, A_K, S_0, N_K)$. In other words, $B_K$ has the same structure of the coordinator’s AC-Graph showed in Figure 4.9 with all nodes marked as accepting nodes. The initial state of $B_K$ is the initial state of $K$. Then, our method performs the following algorithm in order to enforce $p_j$:

1. build the Büchi Automaton that accepts $L(B_K) \cap L(p_j^{CBA})$; this Büchi Automaton is defined as $B_{intersection}^K = (L_{B_K}, N_K \times S', \Delta'', S_0, q_0, N_K \times F')$ where $\langle v_i, q_j \rangle >, \alpha, < v_m, q_n > \in \Delta''$ and only if $(v_i, \alpha, v_m) \in A_K$ and $(q_j, \alpha, q_n) \in \Delta''$;

2. if $B_{intersection}^K$ is not empty then return $B_{intersection}^K$ as the Büchi Automaton corresponding to the $p_j$-satisfying execution paths of $K$.

The previous algorithm allows us to synthesize a deadlock-free coordinator’s AC-Graph that performs only the behaviors specified by means of $p_j^{CBA}$. Note that if $B_{intersection}^K$ is empty then for all possible executions of the CBA system the coordination policy $p_j^{CBA}$ is violated and hence it cannot be enforced. That
is, it is impossible to assemble the components by means of a coordinator which performs only the interactions defined by \( p_j \). Otherwise, our method derives from \( B^K_{\text{intersection}} \) the corresponding coordinator’s AC-Graph. This AC-Graph is constructed by considering the execution paths of \( B^K_{\text{intersection}} \) that contain only accepting paths. We define an accepting path of a Büchi Automaton as follow:

**Definition 23 (Accepting path of a Büchi Automaton)** Let \( B = < S, A, \Delta, s_0, F > \) be a Büchi Automaton then an accepting path of \( B \) is a sequence of states \( \gamma = v_0, v_1, v_2, .., v_n \) such that:

- \( \forall \ i = 0, .., n : v_i \in S \) and \( v_0 = s_0 \) is the initial state of \( B \);
- for \( 0 \leq j \leq n - 1 \), \( (v_j, \alpha_j, v_{j+1}) \in \Delta \);
- \( \exists h = 1, .., n : v_0 \xrightarrow{p}^* v_h \) (for some \( p \)) and \( v_h \in F \).

By referring to the precedent definition, the accepting paths of \( B^K_{\text{intersection}} \) are the only linear or cyclic paths starting from the initial state and containing at least an accepting state. Subsequently, our method reiterates the entire process described in this section by considering the new synthesized failure-free coordinator (obtained from the accepting paths of \( B^K_{\text{intersection}} \)) and the next specified coordination policy \( p_{j+1} \). The method successful terminates when each specified coordination policy has been enforced; otherwise the final failure-free coordinator’s AC-Graph will be empty because at least one policy that cannot be enforced exists.

In Figure 4.13, we show the failure-free coordinator’s AC-Graph for our example.

It is worthwhile noticing that by depending on the coordination policy, in the failure-free coordinator’s AC-Graph we can have either cyclic or linear (i.e., finite) accepting paths\(^6\). In the failure-free coordinator’s AC-Graph of Figure 4.13 we do not have linear accepting paths. Differently from the example showed in this chapter, when we obtain a failure-free coordinator’s AC-Graph with linear accepting paths we consider that each of their accepting states identifies the achievement of a desired behavior. Once the failure-free coordinator execution achieves a desired behavior, it restarts from its initial state.

By taking into account the failure-free coordinator’s AC-Graph and by exploiting the information stored in each node and arc we can automatically derive the code that implements the deadlock-free coordinator component that performs only each specified coordination policy (i.e., the correct assembly code). In Section 5.2.7 we show the technique used to derive the actual assembly code corresponding to the model of the coordinator in detail. We applied this technique in the context of COM/DCOM applications. In the following section we state the correctness and completeness of the enforcing algorithm with respect to the set of policy-satisfying and deadlock-free behaviors of the CFA-system.

### 4.4.3 Correctness and Completeness

In this section we prove the correctness and completeness of the algorithm used for enforcing coordination policies. To state correctness, we prove that the CBA-system based on the failure-free (i.e., policy-satisfying and deadlock-free) coordinator preserves all the policy-satisfying behaviors of the corresponding deadlock-free CFA-system. To state the completeness of our technique we prove that the failure-free CFA-system (i.e., the execution paths of the CFA-system that are both deadlock-free and coordination policy-satisfying) CB-simulates the corresponding failure-free CBA-system which is obtained by introducing into the system the failure-free coordinator.

\(^6\)It might be in the case of the Büchi Automaton of the coordination policy has finite accepting paths.
Figure 4.13: Deadlock-free coordinator’s AC-Graph performing only “AlternatingProtocol” and “No-Fork2First”

**Proposition 5 (Correctness)** Let $T$ be a CFA-system, let $V$ be the corresponding CBA-system, let $\Delta^p_j$ be the corresponding CBA-system based on the deadlock-free coordinator that enforces the $p_j$ coordination policy on the interaction behavior of the functional components in the system and let $\Phi_T = T$ without both the finite paths and the execution paths $\rho$ that are also accepting execution paths of the Büchi Automaton $\neg p_j$ (‘$\neg$’ is the logical not), then $\Phi_T$ can be CB-Simulated from $\Delta^p_j$.

**Proof:** By contradiction, let us suppose that $\Phi_T$ is not CB-Simulated from $\Delta^p_j$. This implies that it exists $t$ labeled with $< T_1, \ldots, T_i, \ldots, T_j, \ldots, T_n >$ and $t'$ in $\Phi_T$ such that $t \rightarrow t'$ and it exists $v$ in $\Delta^p_j$, such that $t \leq_{CB} v$. Then it does not exist a $v'$ in $\Delta^p_j$, such that $v \rightarrow^* v'$ (for some $p$) is in the transition relation of $\Delta^p_j$ and $t'$ is CB-simulated by $v'$. Moreover let us suppose that $t'$ is labeled by $< T_1, \ldots, T'_i, \ldots, T'_j, \ldots, T_n >$ because of $t \rightarrow t'$ by synchronizing $C_i$ and $C_j$ on actions $a$ and $\neg a$, respectively.

By hypothesis we have $(T_i, ?a, T'_i) \in A_{AC}$, and $(T_j, ?a, T'_j) \in A_{AC}$. By definition of EX-Graph we have that $(T_i, ?a, T'_i, T''_i), (T'_i, ?a, T_i) \in A_{EX}$, and $(T_j, ?a, T''_j, T'_i), (T''_j, ?a, T'_j) \in A_{EX}$. By definition of EX-Graphs unification algorithm we have that $(< T_1, \ldots, T_i, \ldots, T_j, \ldots, T_n >, \neg a, < T_1, \ldots, T''_i, \ldots, T''_j, \ldots, T_n >), (< T_1, \ldots, T''_i, \ldots, T''_j, \ldots, T_n >, \neg a, < T_1, \ldots, T_i, \ldots, T_j, \ldots, T_n >)$ are in the set of arcs of the failure-free (i.e., $p_j$-satisfying and deadlock-free) coordinator’s AC-Graph. Since, by hypothesis, $< T_1, \ldots, T''_i, \ldots, T''_j, \ldots, T_n >$ is not a sink node and, for all paths $\rho$ starting from it, $\rho$ is not an accepting execution path of $\neg p_j$, in the transition relation defined for $\Delta^p_j$ we have that $t = v = < T_1, \ldots, T_i, \ldots, T_j, \ldots, T_n > \rightarrow < T_1, \ldots, T_i, \ldots, T_j, \ldots, T_n > \rightarrow < T_1, \ldots, T_i, \ldots, T_j, \ldots, T_n > \rightarrow < T_1, \ldots, T_i, \ldots, T_j, \ldots, T_n > = v' = t'$, which means that it exists a $v'$ that CB-simulates $t'$. This is a contradiction, and thus the proof is complete.

Now we state the completeness of the enforcing algorithm. Analogously to what we have done in Section 4.1.4, we consider the failure-free CBA system which is restricted with respect to the extra level of
that for each state \( v \) in \( \Pi_{\Delta^p_T} \), reachable from \( t_0 \) in \( \Phi^p_T \), it exists a reachable state \( v \) from \( v_0 \) in \( \Pi_{\Delta^p_T} \) such that the label of \( v \) is equal to the label of \( t \).

**Proposition 6 (Completeness)** Let \( T \) be a CFA-system, let \( V \) be the corresponding CBA-system, let \( \Phi^p_T \) be \( T \) without both the finite paths and the paths that are also accepting execution paths of \( \gamma \), let \( \Delta^p_T \) be the failure-free CBA-system based on the deadlock-free coordinator that enforces the \( p_j \) coordination policy on the interaction behavior of the functional components in the system, let \( \Pi_{\Delta^p_T} \) be \( \text{Restrict}(\Pi_{\Delta^p_T}) \) and let \( t_0, v_0 \) be the initial states of \( \Phi^p_T \) and \( \Pi_{\Delta^p_T} \), respectively; for all states \( t \) reachable from \( t_0 \) in \( \Phi^p_T \), it exists a reachable state \( v \) from \( v_0 \) in \( \Pi_{\Delta^p_T} \) such that the label of \( v \) is equal to the label of \( t \).

**Proof:** We give proof by induction on the length of a path in \( \Pi_{\Delta^p_T} \). By construction, the base step (i.e., for a path length equal to 0) is trivially proved because the state label of \( v_0 \) is equal to the state label of \( t_0 \). The inductive step (i.e., for a path length equal to \( n \)) consists in supposing that a generic state \( v \) labeled with \( < V_1, \ldots, V_k, \ldots, V_n > \) in \( \Pi_{\Delta^p_T} \) exists. \( v \) is such that \( v \rightarrow^* v \) (for some \( m \)) and the state label of \( v \) is equal to the state label \( < T_1, \ldots, T_k, \ldots, T_n > \) of a state \( t \) in \( \Phi^p_T \) (i.e., \( \forall i = 1, \ldots, n : V_i = T_i \)).

Moreover let us suppose that \( t_0 \rightarrow t \) (for some \( t \)). Then we have to prove that given a state \( v' \) such that \( v \rightarrow v' \), it is possible to find a state \( t' \) such that \( t \rightarrow^* t' \) (for some \( q \)) and the state label of \( t' \) is equal to the state label of \( v' \). Let us suppose that \( v' \) is labeled by \( < V'_1, \ldots, V'_i, \ldots, V'_n > \) because of \( v \rightarrow v' \) by synchronizing \( C_i[f_i] \) with the failure-free coordinator and, subsequently, the failure-free coordinator with \( C_k[f_k] \) on coordinator’s actions \( ?!_i \) and \( ?!_k \), respectively.

By definition of EX-Graphs unification algorithm, it means that \( \exists i = 1, \ldots, n \land \exists k = 1, \ldots, n \land i \neq k : (V_i, ?!_I_i, V'_i), (V'_i, ?!_I_i, V''_i) \in A_{EX} \) and \( (V_k, ?!_I_i, V''_k), (V''_k, ?!_I_k, V''''_k) \in A_{EX} \). By definition of EX-Graph, AS-Graph and AC-Graph we have that \( (V_i, ?!_I_i, V'_i) \in A_{AC} \) and \( (V_k, ?!_I_i, V''_k) \in A_{AC} \). By definition of CFA we have that \( < V_1, \ldots, V_i, \ldots, V_k, \ldots, V_n > \), \( \tau < V_1, \ldots, V'_i, \ldots, V''_k, \ldots, V_n > \) is in the transition relation defined for \( \Phi^p_T \).

Thus \( v = t = c < V_1, \ldots, V_i, \ldots, V_k, \ldots, V_n > \rightarrow^* c < V_1, \ldots, V'_i, \ldots, V''_k, \ldots, V_n > = v' \) is in the transition relation defined for \( \Phi^p_T \). By hypothesis, the state label of \( t' \) is equal to the state label of \( v' \) and both \( t' \) and \( v' \) are reachable from \( t_0 \) and \( v_0 \), respectively. We have verified the inductive step by construction, and thus the proof is complete.

**Corollary 5** Let \( V \) be a CBA-system, let \( \Delta^p_V \) be the corresponding failure-free CBA-system, let \( \Pi_{\Delta^p_V} = \text{Restrict}(\Delta^p_V) \), let \( T \) be the CFA-system corresponding to \( V \), let \( \Phi^p_T \) be \( T \) without both the finite paths and the paths that are also execution paths of \( \gamma \), and let \( v_0, t_0 \) be the initial states of \( \Pi_{\Delta^p_V} \) and \( \Phi^p_T \), respectively; then \( \Pi_{\Delta^p_V} \) can be CB-simulated from \( \Phi^p_T \).

**Proof:** The proof is trivially derived from the definition of CB-Simulation and by Proposition 6. Actually, by Proposition 6, for each state \( v \) in \( \Pi_{\Delta^p_V} \) reachable from \( v_0 \) it exists a state \( t \) in \( \Phi^p_T \) reachable from \( t_0 \) in such a way that the state label of \( v \) is equal to the state label of \( t \). If we apply Proposition 6 by starting from \( v_0 \) we obtain again the CB-Simulation definition for each state of \( \Pi_{\Delta^p_V} \) with a state of \( \Phi^p_T \). Thus we obtain that for each state \( v \) of \( \Pi_{\Delta^p_V} \) such that the state label of \( v \) is equal to the state label of some \( t \) in \( \Phi^p_T \), if \( v \rightarrow v' \) then there exist \( \exists n > 0 \), \( t_0, \ldots, t_n \) such that \( t = t_0 \) and for all \( i < n : t_i \rightarrow t_{i+1}, v' \leq_{\text{CB}} t_n \). That is \( \Phi^p_T \) CB-simulates \( \Pi_{\Delta^p_V} \).

It is worthwhile noticing that by Proposition 5, Corollary 5 and by considering that the only new logic the coordinator adds to the composed system is the extra level of indirection, we have that the failure-free CFA-system \( \Phi^p_T \) and the “restricted” failure-free CBA-system \( \Pi_{\Delta^p_V} \) are CB-equivalent. That is \( \Phi^p_T \) and \( \Pi_{\Delta^p_V} \) CB-simulate each other.
4.4.4 Failure-free Coordinator Synthesis Algorithm

In this section, we summarize the steps of the entire algorithm used to automatically build the failure-free coordinator’s AC-Graph and to automatically derive, from it, the correct (with respect to failure freeness) assembly code for the functional components forming the given CFA-system:

- let \( T \) be a CFA-system formed by \( C_1, \ldots, C_n \) functional components;
- let \( P \) be the set of coordination policies we want to enforce on the interaction behavior of \( C_1, \ldots, C_n \);
- let \( K \) be the failure-free coordinator we want to build for \( C_1, \ldots, C_n \);
- let \( A \) be the assembly code we want to derive for \( C_1, \ldots, C_n \);

At the beginning both \( A \) and \( K \) are empty:

1. FOR EACH functional component \( C_i \) in \( T \) build the EX-Graph \( EX_i \) of \( C_i \);
2. IF it is impossible to unify the EX-Graphs \( EX_i \) THEN exit (FAILURE) ELSE unify the EX-Graphs \( EX_i \) and put in \( K \) the EX-Graphs unification’s result;
3. IF \( K \) contains deadlocking paths THEN cut these paths from the paths of \( K \);
4. build the Büchi Automaton \( B_K \) corresponding to \( K \);
5. FOR EACH policy \( p_j \) \( \in P \) DO:
   (a) build the CBA version \( p_j^{CBA} \) of \( p_j \);
   (b) build \( B_{K,p_j^{CBA}}^{\text{intersection}} \) from \( B_K \) and \( p_j^{CBA} \);
   (c) assign \( B_{K,p_j^{CBA}}^{\text{intersection}} \) to \( B_K \);
   (d) IF \( B_K \) is empty THEN exit (FAILURE);
6. from \( B_K \) prune all its non-accepting paths;
7. assign to \( K \) the AC-Graph corresponding to \( B_K \);
8. derive from \( K \) the assembly code \( A \);
9. exit (SUCCESS).

We recall that we refer entirely to Section 5.2.7 for a detailed description of the step 8.

4.5 Automatic synthesis of protocol-enhanced adaptors

Besides simply restricting the set of system behaviors to a subset of “safe” or “desired” ones, our approach also considers the possibility of enhancing the current communication protocol of the components forming the composed system. This requires augmenting the set of system behaviors to introduce more sophisticated interactions among components. These enhancements (i.e., protocol transformations) might be needed to achieve dependability, to add extra-functionality or to properly deal with a system’s architecture updates (i.e., aggregating, inserting, replacing and removing components). We address these problems enhancing
our architectural approach which allows for detection and recovery of incompatible interactions by synthesizing a suitable coordinator (see Sections 4.1, 4.2, 4.3 and 4.4). This coordinator represents an initial glue code. So far, as reported in the previous sections, the approach only focused on the restriction of the system’s behavior to a subset of safe (i.e., desired) behaviors. In this section, we propose an extension that makes the coordinator synthesis approach also able to automatically transform the coordinator’s protocol by enhancing the initial glue code. We recall that we implemented the whole approach in our SYNTHESIS tool which is presented in Section 5 in details.

Starting from the specification of the system to be assembled and from the specification of the desired behaviors, SYNTHESIS automatically derives the initial glue code for the set of components. This initial glue code is implemented as a coordinator mediating the interaction among components by enforcing each desired behavior as reported in the previous sections.

Subsequently, taking into account the specification of the needed protocol enhancements and performing the extension we formalize in this section, SYNTHESIS automatically derives, in a compositional way, the enhanced glue code for the set of components. This last step represents the contribution of this section with respect to the previous ones. The enhanced glue code implements a software coordinator which avoids not only incompatible interactions but also provides a protocol-enhanced version of the composed system. More precisely, this enhanced coordinator is the composition of a set of new coordinators and components assembled with the initial coordinator in order to enhance its protocol.

Each new component represents a wrapper component. A wrapper intercepts the interactions corresponding to the initial coordinator’s protocol in order to apply the specified enhancements without modifying the initial coordinator and the components in the system.

The new coordinators are needed to assemble the wrappers with the initial coordinator and the rest of the components forming the system in such a way that the whole assembly is deadlock-free.

It is worthwhile noticing that, in this way, we are readily compose-able; we can treat the enhanced coordinator as a new composite initial coordinator and enforce new desired behaviors as well as apply new enhancements. This allows us to perform a protocol transformation as composition of other protocol transformations by improving on the reusability of the synthesized glue code.

When we apply the specified protocol enhancements to produce the enhanced coordinator, we might re-introduce incompatible interactions avoided by the initial coordinator. That is, the enhancements do not hold the desired behaviors specified to produce the initial coordinator. In this section, we also show how to check if the protocol enhancement holds the desired behaviors enforced through the initial coordinator. This is done, in a compositional way, by using an assume-guarantee technique [21].

4.5.1 Method description

In this section, we informally describe the extension of the coordinator synthesis approach discussed in Section 4.4. We will formalize the extended approach in Section 4.5.2. The extension starts with a deadlock-free CBA which satisfies specified desired behaviors (i.e., coordination policies) and produces the corresponding protocol-enhanced CBA. Thus this extension can be practically seen as a directly subsequent step of the algorithm described in Section 4.4.

The problem we want to solve can be informally phrased as follows: let $P$ be a set of coordination policies, given a deadlock-free and $P$-satisfying CBA system $S$ for a set of black-box components interacting

\footnote{This is needed to achieve compose-ability in both specifying the enhancements and implementing them.}

\footnote{As showed in Section 4.4, the deadlock-free and desired behaviors-satisfying CBA is automatically obtained out of a set of black-box components.}
through a coordinator \( K \), and a set of coordinator protocol enhancements \( E \), automatically derive the corresponding enhanced, deadlock-free and \( P \)-satisfying CBA system \( S' \), if possible.

We are assuming a specification of: i) \( S \) in terms of a description of components and coordinator as AC-Graphs, ii) \( P \) in terms of a set of Büchi Automata, and of iii) \( E \) in form of bMSCs and HMSCs specification [3]. In Chapter 5, we explain why we choose bMSCs and HMSCs as the input language for the enhancements specification. In the following, we discuss our method proceeding in two steps as illustrated in Figure 4.14.

In the first step, by starting from the specification of \( P \) and \( S \), we apply each protocol enhancement in \( E \), if possible. This is done by inserting a wrapper component \( W \) between \( K \) (see Figure 4.14) and the portion of \( S \) concerned with the specified protocol enhancements (i.e., the set of \( C_2 \) and \( C_3 \) components of Figure 4.14). It is worth mentioning that we do not need to consider the entire model of \( K \) but we just consider the “sub-coordinator” which represents the portion of \( K \) that communicates with \( C_2 \) and \( C_3 \) (i.e., the “sub-coordinator” \( K_{2,3} \) of Figure 4.14). \( K_{2,3} \) represents the “unchangeable” environment that \( K \) “offers” to \( W \). The wrapper \( W \) is a component whose interaction behavior is specified in each enhancement of \( E \). Depending on the logic it implements, we can either built it by scratch or acquire it as a pre-existent COTS component (e.g., a data translation component). \( W \) intercepts the messages exchanged between \( K_{2,3}, C_2 \) and \( C_3 \) and applies the enhancements in \( E \) on the interactions performed on the communication channels 2 and 3 (i.e., connectors 2 and 3 of Figure 4.14). We first decouple \( K \) (i.e., \( K_{2,3} \), \( C_2 \) and \( C_3 \) to ensure that they no longer synchronize directly. Then we automatically derive a behavioral model of \( W \) (i.e., an AC-Graph) from the bMSCs and HMSCs specification of \( E \). We do this by exploiting our implementation of the translation algorithm described in [71]. Finally, if the insertion of \( W \) in \( S \) allows the resulting composed system (i.e., \( S' \) after the execution of the second step) to still satisfy each desired behavior in \( P \), \( W \) is interposed between \( K_{2,3}, C_2 \) and \( C_3 \). To insert \( W \), we automatically synthesize two new coordinators \( K' \) and \( K'' \). In general, \( K' \) always refers to the coordinator between \( W \) and the components affected by the enhancement. \( K'' \) always refers to the coordinator between \( K \) and \( W \). By referring to Section 4.2, to do this, we automatically derive two behavioral models of \( W \): i) \( W_{\text{TOP}} \) which is the behavior of \( W \) only related to its top interface (i.e., the TAC-Graph of \( W \)) and ii) \( W_{\text{BOTTOM}} \) which is the behavior of \( W \) only related to its bottom interface (i.e., the BAC-Graph of \( W \)).

\[ \text{Since we want to be readily compose-able, our goal is to apply the enhancements without modifying the coordinator and the components.} \]
In the second step, we derive the implementation of the synthesized glue code used to insert \( W \) in \( S \). This glue code is the actual code implementing \( K'' \) and \( K' \). By referring to Figure 4.14, the parallel composition \( K_{\text{new}} \) of \( K \), \( K' \), \( K'' \) and \( W \) represents the enhanced coordinator.

By iterating the whole approach, \( K_{\text{new}} \) may be treated as \( K \) with respect to the enforcing of new desired behaviors and the application of new enhancements. This allows us to achieve compose-ability of different coordinator protocol enhancements (i.e., modular protocol’s transformations). In other words, our approach is compositional in the automatic synthesis of the enhanced glue code.

### 4.5.2 Method Formalization

In this section, by using an explanatory example different from the one discussed in the previous sections, we formalize the two steps of our method.

![AC - Graph specification of \( S \) and Büchi Automata specification of \( P \)](image)

In Figure 4.15, we consider screen-shots of the SYNTHESIS tool related to both the specification of \( S \) and of \( P \). \( \text{Client1, Client2 and Server} \) are the components in \( S \). \( \text{Property-satisfying Coordinator} \) is the coordinator in \( S \) which satisfies the desired behavior denoted with \( \text{AlternatingProtocol} \). In this example, \( \text{AlternatingProtocol} \) is the only element in the specification \( P \). The CBA configuration of \( S \) is shown in Figure 4.16 where \( C_1, C_2, C_3 \) and \( K \) are \( \text{Client1, Client2, Server} \) and \( \text{Property-satisfying Coordinator} \) of Figure 4.15 respectively.

We recall that each AC-Graph describes the behavior of a component or of a coordinator instance in terms of the messages (seen as I/O actions) exchanged with its environment. Each node is a state of the instance. An arc from a node \( n_1 \) to a node \( n_2 \) denotes a transition from \( n_1 \) to \( n_2 \). The transition labels prefixed by “!” denote output actions (i.e., sent requests and notifications), while the transition labels prefixed by “?” denote input actions (i.e., received requests and notifications). By referring to Section 3.3, in producing the CBA  

---

10The environment of a component/coordinator is the parallel composition of all others components in the system.
4.5 Automatic synthesis of protocol-enhanced adaptors

configuration, we apply relabeling functions on the transition labels of each component AC-Graph. In each transition label, the symbol "\(\alpha\)" followed by a number denotes the component which performs the action (i.e., \(\alpha_j\) denotes an I/O action \(\alpha\) performed by \(C_i\)). The filled nodes on the coordinator’s AC-Graph denote states in which one execution of the behavior specified by the Büchi Automaton AlternatingProtocol has been accomplished.

We recall also that each Büchi Automaton (see AlternatingProtocol in Figure 4.15) describes a desired behavior for \(S\). Each node is a state of \(S\). The node with the incoming arrow is the initial state. The filled nodes are the states accepting the desired behavior. The syntax and semantics of the transition labels is the same of the component and coordinator AC-Graphs except two kinds of action: i) a universal action (e.g., \(?true_1\) in Figure 4.15) which represents any possible action\(^{11}\), and ii) a negative action (e.g., \(!-req_2\) in Figure 4.15) which represents any possible action different from the same negative action\(^{12}\).

\(Client_1\) performs a request (i.e., action \(!req_1\)) and waits for a erroneous or successful notification: actions \(?err_1\) and \(?ok_1\) respectively. \(Client_2\) simply performs the request and it never handles erroneous notifications. \(Server\) receives a request and then it may answer either with a successful or an erroneous notification\(^{13}\).

AlternatingProtocol specifies the behavior of \(S\) that guarantees the progress of all components. It specifies that \(Client_1\) and \(Client_2\) must perform requests by using an alternating coordination protocol. More precisely, if \(Client_1\) performs an action \(req\) (the transition \(!req_1\) from the state \(S_{47}\) to the state \(S_{50}\) in Figure 4.15) then it cannot perform \(req\) again (the loop transition \(!-req_1\) on the state \(S_{50}\) in Figure 4.15) if \(Client_2\) has not performed \(req\) (the transition \(!req_2\) from the state \(S_{50}\) to the accepting state \(S_{125}\) in Figure 4.15) and vice versa.

In Figure 4.17(a), we consider the specification of \(E\) as given in input to the SYNTHESIS tool. In this example, the RETRY enhancement is the only element in \(E\).

\(Client_1\) is an interactive client and once an erroneous notification occurs, it shows a dialog window displaying information about the error. The user might not appreciate this error message and he might lose the degree of trust in the system. By recalling that the dependability of a system reflects the users degree of trust in the system, this example shows a commonly practiced dependability-enhancing technique. The wrapper \(WR\) attempts to hide the error to the user by re-sending the request a finite number of times. This is the RETRY enhancement specified in Figure 4.17.(a). The wrapper \(WR\) re-sends at most two times.

\(^{11}\)The prefixed symbols "!" or "?", in the label of a universal action, are ignored by SYNTHESIS.

\(^{12}\)The prefixed symbols "!" or "?", in the label of a negative action, are still interpreted by SYNTHESIS.

\(^{13}\)The error could be either due to an upper-bound on the number of request that \(Server\) can accept simultaneously or due to a general transient-fault on the communication channel.
Moreover, the RETRY enhancement specifies an update of $S$ obtained by inserting $Client3$ which is a new client. As already mentioned, in specifying enhancements, we use HMSCs and bMSCs. By referring to [3], each bMSC represents a possible execution scenario of the system. Each execution scenario is described in terms of a set of interacting components, sequences of method call and possible corresponding return values. To each vertical lines is associated a set of component instances\(^{14}\) (e.g., \{Client1, Client3\} in Figure 4.17.(a)). Each horizontal arrow represents a method call or a return value. Each usual HMSC describes possible continuations from a scenario to another one. It is a graph with two special nodes: the starting and the ending node. Each other node is related to a specified scenario. An arrow represents a transition from a scenario to another one. In other words, each HMSC composes the possible execution scenarios of the system.

\(^{14}\)This is helpful when we need to group components having the same interaction behavior.
4.5 Automatic synthesis of protocol-enhanced adaptors

**FIRST STEP: WRAPPER INSERTION PROCEDURE**

By referring to Section 4.5.1, each enhancement MSC specification (see Figure 4.17(a)) is generally described in terms of the sub-coordinator $K_j$ (i.e., $K_{j,1}$ in Figure 4.17(a)), the wrapper (WR), the components in $S(\text{Client1})$ and the new components (Client3). The AC-Graph of the sub-coordinator is automatically derived from the AC-Graph of the coordinator in $S(K)$ by performing the following algorithm:

**Definition 24 ($K_{j,...,j+h}$ construction algorithm)** Let $K$ be the AC-Graph of a coordinator, we derive the AC-Graph $K_{j,...,j+h}$, $h \geq 0$, of the interaction behavior of $K$ with components $C_{j,1}, ..., C_{j+h}$ as follows:

1. set $K_{j,...,j+h}$ equal to $K$;
2. for each loop $(\nu, \nu)$ of $K_{j,...,j+h}$ labeled with an action $\alpha = a_k$ where $k \neq j, ..., j + h$ do: remove $(\nu, \nu)$ from the set of arcs of $K_{j,...,j+h}$;
3. for each arc $(\nu, \mu)$ of $K_{j,...,j+h}$ labeled with an action $\alpha = a_k$ where $k \neq j, ..., j + h$ do:
   - remove $(\nu, \mu)$ from the set of arcs of $K_{j,...,j+h}$;
   - if $\mu$ is the starting state then set $\nu$ as the starting state;
   - for each other arc $(\nu, \mu)$ of $K_{j,...,j+h}$ do: replace $(\nu, \mu)$ with $(\nu, \nu)$;
   - for each arc $(\mu, \nu)$ of $K_{j,...,j+h}$ do: replace $(\mu, \nu)$ with $(\nu, \nu)$;
   - for each arc $(\mu, \nu)$ of $K_{j,...,j+h}$ with $\nu \neq \mu, \nu$ do: replace $(\mu, \nu)$ with $(\nu, \nu)$;
   - for each arc $(\nu, \mu)$ of $K_{j,...,j+h}$ with $\nu \neq \mu, \nu$ do: replace $(\nu, \mu)$ with $(\nu, \nu)$;
   - for each loop $(\mu, \mu)$ of $K_{j,...,j+h}$ do: replace $(\mu, \mu)$ with $(\nu, \nu)$;
   - remove $\mu$ from the set of nodes of $K_{j,...,j+h}$;
4. until $K_{j,...,j+h}$ is a non-deterministic AC-Graph (i.e., it contains arcs labeled with the same action and outgoing the same node) do:
   - for each pair of loops $(\nu, \nu)$ and $(\nu, \nu)$ of $K_{j,...,j+h}$ labeled with the same action do: remove $(\nu, \nu)$ from the set of arcs of $K_{j,...,j+h}$;
   - for each pair of arcs $(\nu, \mu)$ and $(\nu, \mu)$ of $K_{j,...,j+h}$ labeled with the same action do: remove $(\nu, \mu)$ from the set of arcs of $K_{j,...,j+h}$;
   - for each pair of arcs $(\nu, \mu)$ and $(\nu, \nu)$ or $(\nu, \nu)$ and $(\nu, \nu)$ of $K_{j,...,j+h}$ labeled with the same action do:
     - remove $(\nu, \nu)$ from the set of arcs of $K_{j,...,j+h}$;
     - if $\nu$ is the starting state then set $\nu$ as the starting state;
     - for each ingoing arc in $\nu$, outgoing arc out from $\nu$ and loop $l$ on $\nu$ do: move the extremity on $\nu$ of in, out and $l$ on $\nu$;
     - remove $\nu$ from the set of nodes of $K_{j,...,j+h}$.
Informally, the algorithm of Definition 24 "collapses" (steps 1, 2 and 3) linear and/or cyclic paths made only of actions performed by the components \( C_k \) with \( k \neq j, \ldots, j + h \). Moreover, it also avoids (step 4) possible "redundant" non-deterministic behaviors\(^{15}\). The algorithm of Definition 24 is a generalization of the algorithm used to synthesize the TAC-Graph and the BAC-Graph of a component/coordinator (see Definitions 17 and 18 respectively). In fact, it synthesizes an AC-Graph of the coordinator which concerns only the interactions among it and a specified set of component. Unnecessarily this set of components has to be the set of all components on the top or on the bottom of the coordinator; it may simply be the set of some components interacting with the coordinator (i.e., it does not matter if they are on the top or on the bottom of the coordinator).

By referring to Figure 4.18, the AC-Graph of \( K_1 \) is the AC-Graph denoted as \( \text{Restricted Coord} \).

![Figure 4.18: AC-Graphs of wrapper, sub-coordinator and \( K'' \)](image)

In general, once we derived \( K_{j, \ldots, j+h} \), we decouple \( K \) from the components \( C_j, \ldots, C_{j+h} \) (i.e., only \( \text{Client1} \) of our explanatory example) that are related to the specified enhancement. To do this, we use the following decoupling function:

**Definition 25 (Decoupling function)** Let \( \text{Act}_K \) be the set of action labels of the coordinator \( K \), let be \( ID = \{ j, \ldots, j + h \} \) a subset of all component identifiers\(^{16}\) of \( K \) and let be \( \delta \neq j, \ldots, j + h \) a new component identifier, we define the "decoupling" function \( f^{j, \ldots, j+h}_\delta \) as follows:

- \( \forall a_\tilde{i} \in \text{Act}_K \), if \( i \in ID \) then: \( f^{j, \ldots, j+h}_\delta(a_\tilde{i}) = a_\delta; \)

The unique component identifier \( \delta \) is automatically generated by SYNTHESIS. In this way we ensure that \( K \) and \( \text{Client1} \) no longer synchronize directly. In Section 5.2.7, we detail the correspondence between the decoupling function and components/coordinator deployment.

---

\(^{15}\)These behaviors might be a side effect due to the collapsing.

\(^{16}\)By referring to Section 4.5.2, the component identifiers are postfixed to the labels in \( \text{Act}_K \).
Now, by continuing the method described in Section 4.5.1, we derive the AC-Graphs for the wrapper (see WR in Figure 4.18) and the new components (the AC-Graph of Client3 is equal to the AC-Graph of Client1 in Figure 4.15 except for the component identifier). We recall that SYNTHESIS does that by taking into account the enhancements specification and by performing its implementation of the translation algorithm described in [71]. It is worthwhile noticing that SYNTHESIS automatically generates the component identifiers for the actions performed by WR and Client3. By referring to Section 3.1, WR is connected to its environment through two connectors: i) one on its top interface (i.e., the connector 432 in Figure 4.17.(b)) and ii) one on its bottom interface (i.e., the connector 401 in Figure 4.17.(b)). Finally, as we will see in detail in Section 4.5.3, if the insertion of WR allows the resulting composed system to still satisfy AlternatingProtocol, WR is interposed between $K_1[f_{417}]$, Client1 and Client3. We recall that $K_1[f_{417}]$ is $K_1$ (see Restricted Coord in Figure 4.18) renamed after the decoupling. To insert WR, SYNTHESIS automatically synthesizes two new coordinators $K'$ and $K''$. Coordinator in Figure 4.18 is the AC-Graph of $K''$. For the sake of presentation, in Figure 4.18, we omit the AC-Graph of $K'$. $K''$ is derived by taking into account both the AC-Graphs of $K_1[f_{417}]$ and WR_TOP (see Figure 4.18) and by performing the synthesis approach formalized in 4.1. While $K'$ is derived analogously from the AC-Graphs of $Client1$, $Client3$ and WR_Bottom (see Figure 4.18). The AC-Graphs of WR_TOP and WR_Bottom are WR_{432} and WR_{401} respectively. By referring to Definitions 17 and 18, WR_{432} and WR_{401} are the TAC-Graph and the BAC-Graph of WR respectively. The resulting enhanced, deadlock-free and Alternating Protocol-satisfying system is:

$$S'' \equiv ((Client_1 | Client_2 | Client_3 | Server | K_{new}) \setminus (Act_{Client1} \cup Act_{Client2} \cup Act_{Server} \cup Act_{Client3})) \text{ where } K_{new} \equiv (K[f_{417}] | WR | K' | K'') \setminus (Act_{WR} \cup Act_{K_1[f_{417}]})$$

and it is graphically showed in Figure 4.17.(b).

**SECOND STEP: SYNTHESIS OF THE GLUE CODE IMPLEMENTATION**

The parallel composition $K_{new}$ represents the model of the enhanced coordinator. By referring to Section 4.5.1, we recall that $K$ is the initial glue code for $S$ and WR is a third-party/ad-hoc component whose interaction behavior is specified by the enhancements specification $E$. That is, the actual code for WR and $K$ is already available. Thus, in order to derive the code implementing $K_{new}$, SYNTHESIS automatically derives the actual code implementing $K'$ and $K''$. This is done by exploiting the information stored in the nodes and arcs of the AC-Graphs of $K'$ and $K''$. More precisely, the code implementing $K'$ and $K''$ reflects the structure of their AC-Graphs which describe state machines. Refer to Section 5.2.7 for a detailed description of the technique used to derive the actual code implementing a coordinator component. We validated and applied our SYNTHESIS tool for assembling Microsoft COM/DCOM components. The reference development platform of the current version of SYNTHESIS is Microsoft Visual Studio 7.0 with Active Template Library.

### 4.5.3 CHECKING ENHANCEMENT CONSISTENCY

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

In general, we have to check:

$$((K[f_{i}^{j-\ldots+j+h}]) | WR | K' | K'') \setminus (Act_{WR} \cup Act_{K_1[f_{i}^{j-\ldots+j+h}]}) \models P_i.$$
By exploiting the constraints of our architectural style, it is enough to check:

\[(K[f^1_{j,...,j+m}] \cap \text{Act}_{K_{j,...,j+m}}) \models P_i;\]

where \(\{j, ..., j+m\}\) is the set of component identifiers which are both component identifiers in \(\{j, ..., j+h\}\) and in the set of component identifiers for the action labels in \(P_i\). In order to avoid the state explosion phenomenon we should decompose the verification without composing in parallel the processes \(K[f^1_{j,...,j+m}]\) and \(K''\). We do that by exploiting the assume-guarantee paradigm for compositional reasoning [21].

By recasting the typical proof strategy of the assume-guarantee paradigm in our context, we know that if \(\langle A \rangle K[f^1_{j,...,j+m}] (P_i)\) and \(\langle \text{true} \rangle K'' \langle A \rangle\) hold then we can conclude that \(\langle \text{true} \rangle (K[f^1_{j,...,j+m}] \cap \text{Act}_{K_{j,...,j+m}}) (P_i)\) is true. This proof strategy can also be expressed as the following inference rule:

\[
\frac{\langle \text{true} \rangle K'' \langle A \rangle \quad \langle A \rangle K[f^1_{j,...,j+m}] (P_i)}{\langle \text{true} \rangle (K[f^1_{j,...,j+m}] \cap \text{Act}_{K_{j,...,j+m}}) (P_i)}
\]

where \(A\) is an LTL formula (and hence it is modeled as a Büchi Automaton). We recall that, in \(S, K\) already satisfies \(P_i\). Once we applied \(E_i\) to obtain the enhanced system \(S'\), \(A\) represents the assumptions (in \(S'\)) on the environment of \(K\) that must be held by the component \(C_i\) in order to make \(K\) able to still satisfy \(P_i\). Without loss of generality, let \(\{j, ..., j+m\}\) be \(\text{Components}_{P_i} \cap \{j, ..., j+h\}\) where \(\text{Components}_{P_i}\) is the set of component identifiers for the action labels in \(P_i\); then \(A\) is the Büchi Automaton corresponding to \(K_{j,...,j+m}f^1_{j,...,j+m}[\text{env}]\) where \(f_{\text{env}}(\alpha) = !\alpha\) and \(f_{\text{env}}(\alpha) = ?\alpha\). For the example illustrated in Section 4.5.2, \(K''\) and \(A\) are the Büchi Automata corresponding to \(K''_{417}\) (i.e., \(K2\) showed in Figure 4.19) and to \(K_1[f^1_{417}][\text{env}]\) (Assumption showed in Figure 4.19) respectively.

![Figure 4.19: Büchi Automata of \(K''_{417}, K_1[f^1_{417}][\text{env}]\) and \(K_1[f^1_{417}][\text{env}]\)](image)

In general, a formula \(\langle \text{true} \rangle M (P)\) means \(M \models P\). While a formula \(\langle A \rangle M (P)\) means if \(A\) holds then \(M \models P\). In our context, \(P\) is modeled as the corresponding Büchi Automaton \(B_P\). \(M\) is modeled as the corresponding LTS. By referring to [21], to a LTS \(M\) always corresponds a Büchi Automaton \(B_M\). With \(L(B)\) we denote the language accepted by \(B\). Exploiting the usual automata-based model checking approach [21], to check if \(M \models P\) we first automatically build the product language \(L_{M \cap P} \equiv L(B_M) \cap L(B_P)\) and then we check if \(L_{M \cap P}\) is empty.

**Theorem 1 (Enhancement consistency check)** Let \(P_i\) be the Büchi Automata specification of a desired behavior for a system \(S\) formed by \(C_1, ..., C_n\) components; let \(K\) be the deadlock-free and \(P_i\)-satisfying coordinator for the components in \(S\); let \(E_i\) be the MSCs specification of a \(K\)-protocol enhancement; let...
4.5 Automatic synthesis of protocol-enhanced adaptors

Let $K''$ be the adaptor between $K$ and the wrapper implementing the enhancement $E_i$; let $\delta$ the identifier of $K''$; let $\{j, \ldots, j + m\}$ the set of component identifiers which are both in the set of component identifiers for the action labels in $P_i$ and in the set of identifiers of components affected by the enhancement $E_i$; and let $f_{\text{env}}$ be a relabeling function in such a way that $f_{\text{env}}(\alpha) = \omega$ and $f_{\text{env}}(\omega) = \omega$ for all $\alpha \in \text{Act}_K$.

If $L_{K''} \cap K_{j, \ldots, j + m}[f_{\text{env}}] = \emptyset$ then

\[
\left( (K'[f_{\text{env}}] \mid K''_{\delta}) \setminus \text{Act}_{K_{j, \ldots, j + m}[f_{\text{env}}]} \right)_{P_i}
\]

is $P_i$ and hence $E_i$ is consistent with respect to $P_i$.

**Proof:** Let $A$ be the Büchi Automaton corresponding to $K_{j, \ldots, j + m}[f_{\text{env}}]$. If $L_{K''} \cap K_{j, \ldots, j + m}[f_{\text{env}}] = \emptyset$ then $K''_{\delta} \models A$. That is, $(\text{true})K''_{\delta}(A)$ holds.

Moreover, by construction of $A$, $(A)K[f_{\text{env}}]_{j, \ldots, j + m}(P_i)$ holds too. By applying the inference rule of the assume-guarantee paradigm, $(\text{true})((K[f_{\text{env}}]_{j, \ldots, j + m} \mid K''_{\delta}) \setminus \text{Act}_{K_{j, \ldots, j + m}[f_{\text{env}}]})(P_i)$ is true and hence

\[
\left( (K[f_{\text{env}}]_{j, \ldots, j + m} \mid K''_{\delta}) \setminus \text{Act}_{K_{j, \ldots, j + m}[f_{\text{env}}]} \right)_{P_i}
\]

holds. In other words, it is enough to check if $K''$ (which is identified by $\delta$) provides $K$ with the environment it expects (to still satisfy $P_i$) on the channel connecting $K''$ to $K$. In the example illustrated in Section 4.5.2, RETRY is consistent with respect to AlternatingProtocol. In fact, by referring to Figure 4.19, $\text{NOT}(A)$ is the Büchi Automaton for $K_{117}[f_{\text{env}}]$ (i.e., for $A$ of Theorem 1) and $K_2$ is the Büchi Automaton for $K''_{117}$ (i.e., for $K''_{\delta}$ of Theorem 1). By automatically building the product language between the languages accepted by $K_2$ and $\text{NOT}(A)$, SYNTHESIS concludes that $L_{K''_{\delta} \cap K_{117}[f_{\text{env}}]} = \emptyset$ and hence that

\[
\left( (K_{117}[f_{\text{env}}] \mid K''_{\delta}) \setminus \text{Act}_{K_{117}[f_{\text{env}}]} \right)_{\text{AlternatingProtocol}}
\]

That is RETRY is consistent with respect to AlternatingProtocol.
In this chapter we provide a description of our tool (called SYNTHESIS), which implements and automates the approach formalized in Chapter 4. By referring to Chapter 4, SYNTHESIS both detects/avoids incompatible interactions and enhances the communication protocol of the components forming the system to be assembled by synthesizing a suitable adaptor/coordinator for the components.

SYNTHESIS can be used either to derive the actual code implementing a coordinator or to only derive its behavioral model (i.e., its state machine or, in other words, its AC-Graph). When the goal of the SYNTHESIS user is to derive the coordinator actual code, the underlining approach depends on the particular development platform that is chosen to implement that code. Otherwise, the approach does not depend by any particular platform. That is, while keeping the automatic synthesis of the model of a coordinator independent from the automatic synthesis of its actual code, SYNTHESIS has to partially refer to one or more particular coordinator development platforms. So far we have validated and applied SYNTHESIS for assembling only Microsoft COM/DCOM components. Thus, the code synthesized by the current version of SYNTHESIS refers to Microsoft Visual Studio with Active Template Library (ATL) as reference development platform.

A prototype of our SYNTHESIS tool is available at the following URL: http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.html.

5.1 SYNTHESIS’S METHOD DESCRIPTION

Since presenting an overview of SYNTHESIS cannot avoid referring to the method formalized in Chapter 4, in this section, for the sake of presentation, we briefly recall this method which is the one implemented by SYNTHESIS. Thus one may skip this section and directly go to Section 5.2 if he/she keeps well in mind Chapter 4.

In using SYNTHESIS, we can distinguish two main phases. The first phase concerns the deadlock-free restriction of the system’s behavior to a specified set of desired interactions. The second phase is related to the augmentation of the system’s behavior to introduce more sophisticated interactions among components.

5.1.1 FIRST PHASE: INITIAL COORDINATOR SYNTHESIS

The first phase concerns with the automatic synthesis of the so-called “initial” coordinator. This coordinator represents an initial glue code. It behaves as a “no-op” coordinator restricted to the only interactions which are deadlock-free and which reflect the ones specified by means of coordination policies (i.e., it performs
only the desired interactions). Given a CFA system $T$ for a set of black-box interacting components and a set of coordination policies $P$, SYNTHESIS automatically derives the corresponding CBA system $V$ which is deadlock-free and which performs only every policy in $P$, if possible. The CBA system $V$ is obtained by interposing the synthesized “initial” coordinator between the components forming $T$.

SYNTHESIS assumes that a specification of the system to be assembled is provided in terms of basic Message Sequence Chart (bMSC) and High-level Message Sequence Chart (HMSC) specification [3]. Moreover, it can take as input a specification of the coordination policies to be enforced in terms of Büchi Automata [21]. With these specifications, SYNTHESIS is able to automatically derive the assembly code for the components forming the specified system. This code implements the “initial” coordinator component. During the first phase, SYNTHESIS proceeds in three steps as showed in Figure 5.1.

(1) The first step builds a behavioral model (i.e., a LTS) of the “no-op” coordinator. Starting from the bMSC and HMSC specification, for each component forming the system to be assembled SYNTHESIS derives a set of LTSs. Each set of component LTSs characterizes different aspects of the component dynamics, from the actual component behavior to its assumptions on the environment. More precisely, for each component instance in the bMSC and HMSC specification, SYNTHESIS firstly extracts a LTS describing the component actual behavior in terms of input/output actions exchanged with its environment. We denote it as AC-Graph. Secondly, for each component’s AC-Graph SYNTHESIS derives a LTS representing the environment expected from the component in order to not block. We denote it as AS-Graph. Finally, for each component’s AS-Graph, by following the CBA style constraints, SYNTHESIS derives a LTS which represents the behavior that the component expects from the “no-op” coordinator in order to not block. We denote it as EX-Graph. Each EX-Graph represents a partial view of the “no-op” coordinator global behavior. By means of an EX-Graph unification algorithm, SYNTHESIS derives the LTS that models the “no-op” coordinator global behavior.

(2) The second step performs the deadlock detection and recovery process against the LTS of the “no-op” coordinator. It simply prunes the paths of this LTS ending with a sink node. This provides us with the LTS of the deadlock-free coordinator.

(3) Finally, the third step synthesizes the LTS of the “initial” coordinator by enforcing the specified coordination policies against the LTS of the deadlock-free coordinator.

From the LTS of the “initial” coordinator SYNTHESIS derives the code implementing it which is by construction correct with respect both to deadlock-freeness and to the coordination policies. Note that although in principle we could carry on the three steps together we decided to keep them separate. This has been done to support internal data structures traceability.

### 5.1.2 Second phase: from the initial coordinator to the protocol-enhanced coordinator

The second phase of the method implemented in SYNTHESIS starts with a deadlock-free CBA system $V$, which performs only the specified desired behaviors (i.e., the coordination policies), and produces the cor-
responding protocol-enhanced CBA system $V'$. Let $P$ be the set of desired behaviors specified within the first phase of our method, given the deadlock-free and $P$-satisfying CBA system $V$ (automatically obtained after the execution of the first phase), and a set of coordinator protocol enhancements $E$. SYNTHESIS automatically derives the corresponding enhanced, deadlock-free and $P$-satisfying CBA system $V'$, if possible. The second phase assumes a specification of $E$ in form of bMSCs and HMSCs. In the following, we recall the method performed by SYNTHESIS, during this phase, proceeding in two steps as illustrated in Figure 5.2.

Figure 5.2: Second phase method

(1) In the first step, by taking into account $P$ and $V$, SYNTHESIS applies each protocol enhancement in $E$, if possible. This is done by inserting a wrapper component $W$ between the initial coordinator $K$ (see Figure 5.2) and the portion of $V$ affected by the specified protocol enhancements (i.e., the set of $C2$ and $C3$ components of Figure 5.2). Note that we do not need to consider the entire model of $K$ but we just consider a “sub-coordinator” which represents the portion of $K$ that communicates with $C2$ and $C3$ (i.e., the “sub-coordinator” $K_{2,3}$ of Figure 5.2). In general, SYNTHESIS denotes this “sub-coordinator” as $SubK$. $K_{2,3}$ (i.e., $SubK$) represents the “unchangeable” environment that $K$ “offers” to $W$. The wrapper $W$ is a component whose interaction behavior is specified in each enhancement of $E$. Depending on the logic it implements, we can either build it by scratch or acquire it as a pre-existent (COTS) component (e.g., a data translation component). $W$ intercepts the messages exchanged between $K_{2,3}$, $C2$ and $C3$ and implements the enhancements in $E$ by acting on the interaction behaviors performed on the communication channels 2 and 3 (i.e., connectors 2 and 3 of Figure 5.2). We first decouple $K$ (i.e., the “sub-coordinator” $K_{2,3}$), $C2$ and $C3$ to ensure that they no longer synchronize directly. Then, we automatically derive two behavioral models of $W$ (i.e., two AC-Graphs) from the bMSC and HMSC specification of $E$. These two AC-Graphs are: i) $W.1$ which is the behavior of $W$ only related to the interactions with the components affected by the enhancement and ii) $W.2$ which is the behavior of $W$ only related to the interactions with $K$. We do this analogously to what we done, in Section 5.1.1, to derive the AC-Graph for each component in the CFA system. Finally, if the insertion of $W$ in $V$ will allow the resulting composed system (i.e., $V'$ after the execution of the second step) to still satisfy each desired behavior in $P$, $W$ is interposed between $K_{2,3}$, $C2$ and $C3$. To insert $W$, we automatically synthesize two new coordinators $K.1$ and $K.2$. In general, $K.1$ always refers to the coordinator between $W.1$ and the components affected by the enhancement; $K.2$ always refers to the coordinator between $K$ (i.e., $SubK$) and $W.2$.

(2) In the second step, we derive the implementation of the synthesized glue code used to insert $W$ in $V$. This glue code is the actual code implementing $K.2$ and $K.1$. By referring to Figure 5.2, the parallel composition $K_{\text{new}}$ of $K$, $K.1$, $K.2$ and $W$ represents the enhanced coordinator. By iterating the whole approach, $K_{\text{new}}$ may be treated as $K$ with respect to the enforcing of new desired behaviors and the appli-
cation of new enhancements. This allows us to achieve compose-ability of different coordinator protocol enhancements (i.e., modular protocol’s transformations). In this way, our approach is compositional in the automatic synthesis of the enhanced glue code.

5.2 THE SYNTHESIS TOOL

In this section, we describe the architecture of our SYNTHESIS tool by looking at its main constituent software modules. For some modules we also point out both implementation decisions and justification of these choices. It is worth mentioning that the current version of SYNTHESIS should be considered a prototype. We are developing it by following an evolutionary approach to systems development. Thus the current prototypical version of SYNTHESIS is still subjected to further extensions. SYNTHESIS is implemented by using Java and some of its input and output data are coded using XML. This is done to achieve both platform-independence and data format portability.

In Figure 5.3 we show the overall structure of SYNTHESIS focussing on the main software modules and their relationships.

SYNTHESIS can be seen as a container of the above mentioned modules (shown in Figure 5.3) which defines relationships among them and also provides editing and viewing functionality. Analogously to what we did in Section 5.1, in describing the SYNTHESIS architecture, we distinguish two main phases.
5.2 The SYNTHESIS tool

5.2.1 FIRST PHASE

In Figure 5.4 we show the input and output data performed by SYNTHESIS within the first phase. It is worthwhile noticing that, by means of capital letters, we obtain a direct mapping between Figure 5.3 and Figure 5.4. This mapping allows us to correlate each module with the I/O data it manages, performs or builds.

In the remainder of the chapter, we briefly describe each SYNTHESIS module. In doing this we refer to the method recalled in Section 5.1.

5.2.2 MODULE A: COMPONENT INTERFACE PARSER

This module contains a superclass (i.e., represented by the "IDLParser" entity in Figure 5.3) that manages a specific data structure which is for storing an abstract representation of an IDL file possibly given as input. That superclass has to be specialized in order to implement a parser of IDL files based on a particular IDL notation (e.g., Microsoft IDL for COM/DCOM, DCE/IDL for CORBA, etc.). In the current version of SYNTHESIS, we specialized that class to implement a parser of Microsoft IDL (MIDL) files. We choose to support MIDL because so far we have validated SYNTHESIS only in the context of COM/DCOM application.

5.2.3 MODULE B: bMSC AND HMSC SPECIFICATION OF THE CFA

This module is used to specify the CFA system to be assembled in terms of a bMSC and HMSC specification. In doing that, this module exploits an ad-hoc library that we developed to allow creation, validation...
and manipulation of bMSCs and HMSCs coded in XML. To check if these XML files are valid, an ad-hoc XML schema is used. This module requires a suitable implementation of the “IDLParser” entity (see module A showed in Figure 5.3) only when the user’s goal is to derive the actual code of the coordinator assembling the specified system. In this case, the parser provides the user with a list of all requests exported by a component. In this way, he/she can directly select those requests while specifying a bMSC. Otherwise each bMSC can be specified in terms of labels modeling imaginary method names.

In order to better propose our approach to practical software development settings, we chosen to use bMSC and HMSC specification as input language for our framework. By exploiting the translation algorithm described in [71], we can derive LTS descriptions for each component (i.e., its AC-Graph) forming a system from their bMSC and HMSC specification. Thus, CCS and LTS can be regarded as an internal specification language. Moreover, bMSC and HMSC match our purposes since, by means of HMSC, composing single execution scenarios of a system (i.e., bMSCs) is possible. By referring to [33], although bMSCs and HMSCs might be completely substituted by UML2.0 interactions and overview diagrams [2] which are recently introduced specification languages more commonly used in practical software development, we decide to keep bMSCs and HMSCs since they have been introduced in our framework when UML2.0 specification has not been released yet. However, by exploiting the XML Metadata Interchange (XMI) [2] of UML2.0, our framework can easily comply with UML2.0.

In Figure 5.5, we report screen-shots of SYNTHESIS showing the bMSC specification of the CFA system S for the explanatory example discussed in Section 4.5.2.

Analogously to what we done in Section 4.5.2, each bMSC represents a possible execution scenario of S (i.e., CLIENT1_ERR, CLIENT1_OK and CLIENT2_OK). Each execution scenario is described in terms of a set of interacting components, sequences of requests and possible corresponding notifications. To each vertical line we associate an instance of a component. Each horizontal arrow represents a request or a notification. CLIENT1_ERR describes the scenario in which Client1 performs the request req and receives err as notification. CLIENT1_OK is analogous to CLIENT1_ERR but the notification which is ok. CLIENT2_OK is analogous to CLIENT1_OK but Client2 which performs the request req. As we will explain in the description of the module F, for our purpose, SYNTHESIS does not require to specify the actual parameters list for each request. Figure 5.6 is a screen-shot of SYNTHESIS showing the HMSC specification of S.

Each HMSC describes possible continuations from a scenario to another one. It is a graph with two special nodes: a starting and a possible ending node. Each other node is related either to a specified scenario or to a nested HMSC (i.e., an HMSC is a composition of either bMSCs or nested HMSCs). An arrow represents a transition from a scenario to another one. In other words, each HMSC composes the possible execution scenarios of S. By referring to Figure 5.6, from the starting state, the system S can execute one of the three specified scenarios. Then, from each scenario, T can re-execute that scenario or a different one.
5.2 The SYNTHESIS tool

5.2.4 Module C: Component Assumption Generator

By referring to Figure 5.3 this module exploits (i.e., requires) the module B. From the bMSC and HMSC specification of Figures 5.5 and 5.6, this module outputs the component AC-Graphs showed in the left-hand top-side of Figure 4.15. It implements a revisited version of the algorithm described in [71]. In accordance with the bMSC and HMSC data format, these AC-Graphs are coded in XML too.

By referring to Section 5.1.1, from the component AC-Graphs, this modules can also output the component AS- and EX-Graphs.

5.2.5 Module D: Büchi Automata Specification of the Coordination Policies

This module is used to specify the coordination policies (i.e., the desired behavior) for the CFA system S in terms of Büchi Automata. The current implementation of this module codes each Büchi Automaton as a binary object by exploiting Java serialization. A corresponding version based on XML is still work in progress. Each Büchi Automaton describes a coordination policy that must be implemented by the coordinator to mediate the interaction among the components. In other words, the coordinator has to be derived in order to force each coordination policy on the interaction behavior of the components forming the CFA system.

An example of an input performed by the module D is the “AlternatingProtocol” Büchi automaton showed in the right-hand top-side of Figure 4.15.

5.2.6 Module E: Builder of the Initial Coordinator Model

This module is responsible for deriving the model of the initial coordinator. In particular, it is specialized by the “EXUnificator”, “DlockEraser” and “PolicyEnforcer” entities showed in Figure 5.3.E. These entities respectively implement: (i) the EX-Graph unification algorithm which is for building the LTS of the “no-op” coordinator (see Section 4.1.2); (ii) the deadlock avoidance algorithm to derive the LTS of the deadlock-free coordinator (see Section 4.1.3); (iii) the coordination policy enforcing algorithm that is for obtaining the LTS of the initial coordinator (see Section 4.4).

In the bottom-side of Figure 4.15 we show the initial coordinator that implements the “AlternatingProtocol” policy for the components “Server”, “Client1” and “Client2” showed in the top-side of the figure as well.
5.2.7 **MODULE F: GENERATOR OF THE INITIAL COORDINATOR ACTUAL CODE**

This module implements a generator of the initial coordinator actual code. It is structured analogously to module A. Thus (when it is used) it refers to one or more specific development platforms. Currently it only supports the generation of the code implementing the coordinator COM/DCOM component. This code refers to *Microsoft Visual Studio* with *Active Template Library* (ATL) as reference development platform.

The AC-Graph of the initial coordinator is a state machine and module F derives the actual code of a COM/DCOM server that reflects the state machine’s structure and behavior. By visiting the coordinator AC-Graph, showed in the bottom-side of Figure 4.15, and by exploiting the information stored in its states and transitions, module F derives the code that implements the coordinator COM/DCOM component. In this way, the traceability between model’s structure and implementation is obtained. In deriving this code, module F takes into account that the coordinator component is a single-thread and composite server that encapsulates all servers into the CFA system through containment/delegation mechanism; moreover it also exports all its inner interfaces (i.e., the interfaces exported by the encapsulated servers). After SYNTHESIS used the MIDL compiler¹ to derive the *Type Library* of the encapsulated servers, the coordinator component code imports each type library and implements each inner interface by defining a new COM class. This class re-implements and exports the same methods declared by the inner interfaces. To automatically generate the GUIDs of the interface exported by the coordinator (i.e., *IKCoordinatorPSat*) and of its COM class respectively (i.e., *CKCoordinatorPSat*), module F uses another tool (called "uuidgen.exe") which is bundled with *Microsoft Visual Studio*. The following is the automatically derived header (*KCoordinatorPSat.h*) file of the coordinator component showed in the bottom-side of Figure 4.15:

```c++
#pragma once
#include "resource.h"
#include <vector>
using namespace std;

typedef vector<int> SLABELVECT;

#import "_ServerPrj.tlb" raw_interfaces_only, raw_native_types, no_namespace,
named_guids, auto_search
[
  object,
  uuid("536F2DB6-398A-4c1d-AE02-2BC29AC9F35F"),
  dual, helpstring("IKCoordinatorPSat Interface"),
  pointer_default(unique)
]__interface IKCoordinatorPSat : IDispatch
{
  [id(1),helpstring("method req")]
  HRESULT req([in] int val,
               [out,retval] BSTR *status);
}

[  
  coclass,
  threading("apartment"),
  vi_progid("KCoordinatorPrjPSat.KCoordinatorPSat"),
  progid("KCoordinatorPrjPSat.KCoordinatorPSat.1"),
  version(1,0),
]*

¹It is bundled with *Microsoft Visual Studio*.
It is worthwhile noticing that the COM class of the coordinator component defines for its objects both shared and non-shared state. The shared state is represented by a set of static private members while the non-shared state is represented by the remaining private members. “pIServer” is a COM/DCOM smart pointer [4] referring to the interface exported and implemented by the component Server of left-hand top-side of Figure 4.15 (i.e., IServer). In general, for each server object encapsulated into coordinator component we have a set of private members which are smart pointers. Each smart pointer refers to an interface exported by the encapsulated server object. By defining and using “sLabelVect”, SYNTHESIS is able to model a vector of coordinator graph’s state labels. “clientsCounter” counts the number of clients.
connected to the coordinator component. “chId” is different for each client connected to the coordinator component. It models the identifier of the channel that the client uses to be connected to the coordinator (and hence it identifies a client connected to the coordinator component too). In the constructor of the coordinator COM class, module $F$ has automatically derived the code initializing part of the private members.

“sLabelVect” is initialized by inserting the label of the starting state of the coordinator graph (i.e., the state label $S_0$). “clientsCounter” is incremented each time a client requests to create an instance of the coordinator component. The value of “chId” is the value of “clientsCounter” after its increment. In deploying the CBA version of the system, SYNTHESIS takes into account that the coordinator has to be registered in the same machine of the servers it encapsulates. This, in turn, implies all the encapsulated servers have to be registered in the same machine. Moreover, by referring to Section 3.5.2, for each server $S$ the value of the key $HKEY\_CLASSES\_ROOT/CLSID/<...unique class identifier of S...>/TreatAs is the value of the CLSID of the coordinator component (i.e., uuid("56207063-CB18-43d3-BE53-705D8E1969C4")). This is needed to interpose the coordinator component between the clients and the servers in order to ensure that they no longer communicate directly. Thus, in the constructor, before instantiating a reference to $IServer$, module $F$ adds the code in order to not instantiate a reference to the same coordinator (i.e., the code line $CoTreatAsClass(uuidof(CServer), CLSID\_NULL)$). After the $IServer$ reference instantiation, module $F$ adds the code to interpose the coordinator between the clients and the servers again (i.e., the code line $CoTreatAsClass(uuidof(CServer), uuidof(CKCoordinatorPSat))$). In general, this mechanism has to be implemented for each object server encapsulated into coordinator component. The following is the automatically derived source (KCoordinatorPSat.cpp) file of the coordinator component:

```cpp
#include "stdafx.h"
#include "KCoordinatorPSat.h"
#include "_ServerPrj.tlb" raw_interfaces_only, raw_native_types, no_namespace, named_guids, auto_search

IServerPtr CKCoordinatorPSat::pIServer = NULL;
SLABELVECT CKCoordinatorPSat::sLabelVect;
int CKCoordinatorPSat::clientsCounter = 0;

HRESULT CKCoordinatorPSat::req(int val, BSTR *status)
{
    HRESULT hr = S_OK;
    try
    {
        if(chId == 1) {
            if(ElementOf(0)) { // a state in which a request of req_1 is allowed
                // method call delegation
                hr = pIServer->req(val, status);
                // Update the vector of consistent state labels...
                sLabelVect.erase(sLabelVect.begin(), sLabelVect.end());
                sLabelVect.push_back(5);
                // ...end of updating
            } else if(ElementOf(3)) { // a state in which a request of req_1 is allowed
                // method call delegation
                hr = pIServer->req(val, status);
            }
        }
    } catch(...) {
        hr = E_FAIL;
    }
    return hr;
}
```

2At the beginning “clientsCounter” is equal to 0
5.2 The SYNTHESIS tool

// Update the vector of consistent state labels...
sLabelVect.erase(sLabelVect.begin(), sLabelVect.end());
sLabelVect.push_back(0);
// ...end of updating
}
}
else if(chId == 2) {
    if(ElementOf(5)) { // a state in which a request of req_2 is allowed
        // method call delegation
        hr = pIServer->req(val, status);

        // Update the vector of consistent state labels...
        sLabelVect.erase(sLabelVect.begin(), sLabelVect.end());
sLabelVect.push_back(0);
        // ...end of updating
    }
    else if(ElementOf(0)) { // a state in which a request of req_2
        // is allowed
        // method call delegation
        hr = pIServer->req(val, status);

        // Update the vector of consistent state labels...
        sLabelVect.erase(sLabelVect.begin(), sLabelVect.end());
sLabelVect.push_back(3);
        // ...end of updating
    }
}
}

catch (_com_error e) {
    MessageBox(NULL, e.ErrorMessage(), "SMART K: COM Error", MB_OK);
}

return hr;
}

bool CKCoordinatorPSat::ElementOf(int sl)
{
    for(unsigned int i=0; i<sLabelVect.size(); i++)
        if((int)sLabelVect[i] == sl)
            return true;

    return false;
}

In the source file, for each public method \textit{m}, module 	extbf{F} derives the code implementing the logic of the coordinator related to \textit{m}. This logic is trivially derived by visiting the coordinator graph. Informally, let “\textit{admissible state labels of m}” be the set of labels of states in which a request of \textit{m} is allowed; this information is derived by visiting the coordinator graph and looking for all the states with “\textit{m}_i” (for some \textit{i}) among their outgoing transition labels. For each client identifier (i.e., for each admissible value of \textbf{“chId”}), module \textbf{F} derives an “if-statement” in which, for all “\textit{admissible state labels of m}”, the following two steps are performed:

1. if \textit{s} is an element of “\textbf{sLabelVect}”, then the coordinator component’s execution has reached a state...
in which a request of \( m \) (performed by the client identified through “chId”) is allowed; thus module \( F \) adds the code to delegate the request of \( m \) towards the encapsulated server;

2. module \( F \) updates “sLabelVect” by erasing all the old elements and by adding new elements; each new element represents a label of a state which is reachable after the delegation of \( m \) (this information is derived by visiting the coordinator graph and looking for all the states reachable from \( s \) and corresponding to “admissible state labels of \( m \)).

It is worthwhile noticing that the coordinator component does not care about checking the value of actual parameters of a method. In fact, it is only a method call delegator whose delegation logic respects both the deadlock-freeness and the specified coordination policies. It solves the problem of making the components (forming the CFA system) able to interact, in a deadlock-free way, by following only the specified desired behaviors (i.e., the coordination policies). Dealing in a black-box components setting, the coordinator does not care about details related to the components internal behavior. That is the reason for which, in drawing the MSC specification of the CFA system, module \( B \) does not require that a user must specify the actual parameters list of a method call.

Once module \( F \) has automatically derived the header (KCoordiantorPSat.h) and the source (KCoordiantorPSat.cpp) file for the coordinator, in order to build its binary executable source (KCoordiantorPSat.exe), it is enough to launch Microsoft Visual Studio, create a new ATL project through the project’s wizard, add, as existing items, KCoordiantorPSat.h and KCoordiantorPSat.cpp to the project and execute the builder. At this point, KCoordiantorPSat.exe is the initial coordinator component\(^3\). We recall that it has to be registered in the same machine where the encapsulated servers have been registered. In doing that, the value of the “TreatAs” keys under the registry’s hives related to the CLSIDs of the encapsulated servers must be set to the CLSID of the coordinator component as reported in Section 3.5.2.

5.2.8 SECOND PHASE

In Figure 5.7 we show the input and output data performed by SYNTHESIS within the second phase. Analogously to what we done in Section 5.2.1, by means of capital letters, we obtain a direct mapping between Figure 5.3 and Figure 5.7.

In the following, we describe the module \( G \) which is involved in the second phase. In doing this, we refer to Section 5.1.2.

5.2.9 MODULE \( G \): WRAPPER ASSUMPTION GENERATOR

This module is responsible for deriving behavioral models of the wrapper component corresponding to its assumptions on the environment (i.e., its AC-, AS- and EX-Graph). We recall that the wrapper is needed to apply a specified enhancement. Since an enhancement is specified in terms of a bMSC and HMSC specification of the wrapper, to do that, this module exploits both the module \( B \) and \( C \). More precisely, it directly requires the module \( E \) which, in turn, exploits \( C \) and subsequently \( B \). Moreover it might also use the module \( A \) to parse IDL files of the wrapper component and of possible new components. We recall that depending on the enhancement, new components might be required in order to deal with architectural updates. We given an example of an input performed by this module in Figure 4.17.

In order to apply the specified enhancement, this module performs the following operations:

\(^3\)Its MIDL specification is automatically generated by Visual Studio.
5.2 The SYNTHESIS tool

Figure 5.7: Input and output data performed by SYNTHESIS within the second phase

1. from the MSC specification (see B in Figure 5.7), it derives both the AC-Graphs of the new inserted components and the AC-Graphs WR.1 and WR.2. Within the CBA style, WR.1 (WR.2) describes the actual behavior of the wrapper only related to its cooperation with the components below (above) it;

2. from the model of the initial coordinator (corresponding to E in Figure 5.7) and the MSC specification of the enhancement, it derives the AC-Graph of SubK;

3. it decomposes the CBA system obtained by the execution of the first phase in two CFA systems. One is formed by WR.1 and the components below (above) it; the other CFA is formed by WR.2 and SubK which is above (below) the wrapper.

To insert the wrapper component, this module automatically synthesizes two new coordinators K.1 and K.2. This is did, analogously to what we reported in Section 5.2.1, by performing the first phase for the two CFA systems above mentioned. It is worthwhile noticing that, at this stage, it is also possible to enforce new desired behavior on K.1 and K.2.

The parallel composition $K_{new}$ of K.1, K.2 and WR represents the model of the enhanced coordinator. In order to derive the code implementing $K_{new}$, this module (by means of the module F) automatically derives the actual code implementing K.1 and K.2 analogously to what we done for deriving the code of the initial coordinator.
In this chapter we report three case studies we used to evaluate the current prototype version of our SYNTHESIS tool (see Chapter 5). In presenting these case studies we used different versions of SYNTHESIS that go from the first version (see Section 6.1) to an intermediate version (see Section 6.2) and, finally, to the latest one (see Section 6.3). This is due to the fact that the three case studies were carried out in different periods of time. The latest version of our tool is the one presented in Chapter 5. The first version was a primordial one and it did not provide the user with a Graphic User Interface (GUI) (i.e., it was a command line version). This is the reason for which, in showing the input and output data of our tool, in Section 6.1 we cannot provide screen-shots of SYNTHESIS. With respect to the intermediate version, the module B (see Chapter 5), whose functionality is to perform the bMSC and HMSC specification of the system to be assembled and of possible protocol enhancements, had no GUI. For this reason in Section 6.2 we draw each bMSC and HMSC specification by hand.

The case studies in Sections 6.1 and 6.2 respectively address the problem of restricting the system behavior to a set of component interactions that are deadlock-free and satisfy a specified desired behavior. The case study in Section 6.3 concerns the possibility to deal with components that might have incompatible interface signatures and the problem of not only properly restricting the system behavior but also augmenting it in order to introduce more sophisticated interactions among components.

### 6.1 Automatic Coordinator Synthesis for a COTS Group-ware System

Computer Supported Cooperative Work (CSCW) [73] is a multi-disciplinary research area mainly focused on effective methods of sharing information and coordinating concurrent activities. The coordination of these activities is a very important and difficult task. Many approaches for the construction of large-scale flexible (in coordination) group-ware applications there exist in literature. Most of them are inspired by coordination languages and models [5, 53] which propose the separation of computation from coordination for multi-threaded applications. These approaches are mainly concerned with design and coordination architectural models [45, 48, 57], coordination protocols [27] and languages [22]. They provide support for developing flexible group-ware applications, specifying and controlling the interaction among collaborative activities, evaluating the behavior of a CSCW system before its implementation, modeling, analyzing and prototyping the coordination policies of a collaboration system, and designing CSCW systems. These approaches work in a white-box components setting and are flexible in terms of supported controlled coordination policies. On the other hand they cannot straightforwardly be exported in a black-box component setting. When we deal with group-ware applications built by assembling third-party components which are truly black-box, the issue is not only in specifying and analyzing a coordination policy rather in being able to enforce it out of a set of already implemented (local) behaviors. In a black-box components setting, one of the challenges is related to the ability to predict possible coordination policies of the components interaction behavior by only exploiting a limited knowledge of the single components computational behavior.
In this section we apply our architectural approach (see Chapter 4), whose implementation and automation is supported by our SYNTHESIS tool (see Chapter 5), to a case study in the context of COTS-based group-ware applications development. The assembly of these applications must hold specified behavioral properties. In the context of the case study we only consider behavioral properties that model coordination policies. Our approach puts together COTS-systems by assuming a well defined architectural style (see Section 3.1) in such a way that it is possible to detect and to fix coordination anomalies. We assume that a specification of the desired assembled system is available and that a precise definition of the coordination policies to enforce exists. The coordination policies are specified in terms of behavioral properties of the composed system. With these assumptions our tool SYNTHESIS is able to automatically derives the assembly code for a set of components so that, if possible, a coordination policy-satisfying system is obtained. The assembly code implements an explicit software coordinator which mediates all interactions among the system components as a new component to be inserted in the composed system. The coordinator can then be analyzed and modified in such a way that the specified coordination policies are enforced.

In the reminder of this section we apply our approach in order to build from a set of suitable COTS components a Collaborative Writing (CW) system [32, 46]. The CW system is composed through coordinator components automatically synthesized in order to satisfy a given specification. This specification might be not satisfied from the CFA version of our CW system. First, we provide in Section 6.1.1 some background notion on the group-ware applications domain with particular interest on the CW systems. Then, in Section 6.1.2, we describe a CW system designed by basing on the main features of nowadays existent CW systems and finally we apply our approach in order to guarantee a given system’s specification.

6.1.1 CW systems: State of the art

Collaborative writing is one discipline of the multi-disciplinary research area known as Computer Supported Cooperative Work (CSCW) [73]. Collaborative writing is defined in [46] as: “the process in which authors with different expertise and responsibilities interact during the invention and revision of a common document”. A CW system involves two or more people (geographically distributed) working together to produce a common document. CW systems proposed in literature [42, 43, 51, 58] are often categorized according to the time/location matrix [32] in two major groups. First, there are systems supporting synchronous editing such as GROVE [25], PREP [51] and Aspects [20]. This group of CW systems provides the cooperating team partners (i.e., authors and co-authors) with updates in real-time. The second group is related to asynchronous writing tools, such as PREP [51], CONCORD [58], Col•laboració [20], IRIS group editor environment [42, 43] and Quilt [26]. To better support all the CW stages, in literature have been proposed also semi-synchronous CW systems supporting the integration of asynchronous and synchronous styles of work such as PREP [20, 47] and COOP/Orm [49]. Semi-synchronous CW systems seem to be the better solution for the complete support of all the activities related to CW during the document life cycle. Actually, in [47] the analysis of synchronous and asynchronous CW systems concludes that “a collaborative software environment have to support both synchronous and asynchronous working strategies and smooth transitions between them”.

Referring to [20, 42, 43, 51], we can summarize the requirements of a generic CW system. A CW system must involve phases of writing (i.e., synchronous, asynchronous or semi-synchronous) and phases of communication. It must be able to support:

- different writing strategies;
- all authors must be able to work in their preferred working styles and working environments;
- geographically distributed (possibly mobile) group members;
- concurrent data access and data consistency (i.e., distributed users should see identical or, at least, consistent views of shared data);
6.1 Automatic coordinator synthesis for a COTS group-ware system

- different communication protocols;
- generation and distribution of the information for group awareness;
- extensibility and configurability;
- availability (i.e., users should be able to gain access to data when they need it);
- responsiveness (i.e., the data access process should not interfere with the interactive response of the system);
- physical mobility of a group member (i.e., local replication of the shared data and document’s versions handling mechanisms);
- version-controlled documents.

All the CW systems cited above support these requirements in different ways. It is worthwhile noticing that a common challenge for every CW system is the concurrency control problem [25, 42, 58]. Concurrency control is needed within CW systems to avoid or to resolve conflicts between different accesses to the same data. Depending on synchronous, asynchronous or semi-synchronous writing a CW system applies different concurrency control policies [42, 47]. The methods for concurrency control are grouped in: i) pessimistic methods and ii) optimistic methods. The former are more restrictive than the latter and guarantee that no conflict occurs. Typically the pessimistic methods are implicit in a CW system and are obtained through roles, privileges, group member restrictions and/or by storing every document in a hierarchical data structure [43, 49]. The latter are more permissive and guarantee to detect conflicts after they occurred. Conflicts detection mechanisms are required. The conflict recovery is not always automatic but in most cases it requires a human intervention [42]. A concurrency control policy has great influence on the availability, consistency and responsiveness of a CW system. To fulfill the requirements of availability, consistency and responsiveness, a better solution is to make use of both optimistic and pessimistic methods [42, 58]. In [47], they use version-controlled hierarchical documents in order to avoid conflicts during concurrent accesses. One step of evolution of a document is represented by one version of the document. When a user wants to edit a version that is edited by another user, an alternative is created instead of a version. Subsequently, all the alternatives of the last version are merged into the new version in order to produce a new version of the current document. This makes possible the simultaneous editing among multiple users.

6.1.2 A SEMI-SYNCHRONOUS CW SYSTEM

In this section we present our CW system and apply our approach in order to accommodate it to a given specification. Basing on the CW systems analysis of Section 6.1.1, we can identify the COTS components that provide the main features of a CW system. Once we have identified them, let us suppose that we acquire these third-party (COTS) components and we assemble them in order to build our CW system. We apply our approach in order to automatically derive the assembly code that satisfies a particular specification. This specification is formulated in form of coordination policies (i.e., behavioral properties) to be checked on the assembly.

Referring to the component-based design in group-ware applications domain [43], it is worthwhile noticing that to fulfill the requirements listed in Section 6.1.1, a CW system must be designed as an integrated environment. This can be done by exploiting object-oriented concepts and component technology in CW systems design. Thus by referring to the above listed CW systems requirements and by being inspired by the architectural design of three existent CW systems (i.e., IRIS group editor environment [42, 43], PREP [20, 47, 51] and COOP/Orm [49]) we can identify four third-party components which are useful to build our semi-synchronous CW system. These four COTS components are showed in Figure 6.1.

Our CW system is a two-layered CFA (i.e., three-tier application) formed by the following COTS components we have identified:
• **DBO**: This component is a database. The database stores all information useful to support a group activity; in particular, it stores all group awareness information such as user information, user group information, document information, document structure information, access information related to a document or to an element of a document’s structure (i.e., document partition), an history of all document’s updates. Moreover the DBO can be modeled like a server component that exports two services: i) **update** and ii) **query**. The former models both the insertion, the deletion and the update of an information in database; the latter models the execution of a query on the database in order to retrieve some information. This component provides persistence for group awareness information and a support for users mobility. Actually, through DBO, a group member (i.e., a user) can know in every time the work status of another user even though this user is logged out from the system while the current CW work session is not still closed. Moreover, it provides an implicit method (pessimistic method) for the concurrency control.

• **CWE**: This component is a CW engine; it provides all services useful to perform a group activity in the context of CW. It is an handler of all group awareness information stored in DBO and of the following CW activities: i) document structure defining and editing, ii) user registration, iii) user deregistration, iv) users group defining and editing, v) user login and logout, vi) the access to a document or to an its partition, vii) the display of the information related to the access to a document or to an its partition and viii) the display of the history of all document’s updates. According to the above CW activities the CWE exports the following services to its clients: i) **strEd**, ii) **reg**, iii) **unreg**, iv) **grpEd**, v) **lIn** and **lOut**, vi) **accEd**, vii) **accSh** and viii) **hist**.

• **DSS**: Referring to [20, 42, 43, 47, 49, 51] and according to a document’s structure (e.g., chapters, sections, sub-sections), a document is a set of document partitions; each document partition might contain different types of data (e.g., text, graphic, video). An example are hierarchical document structures, hypertext graph structures, hierarchical structures tree [43] or version-controlled hierarchical documents [47]. The granularity of partitions is determined by the manager of the CW system (e.g., the author of the document). This granularity can be exploited for conflict avoidance. This component is an abstraction of the physical container of the shared documents that are logically partitioned according to their structure. In an asynchronous working mode it should be replicated and we have to use optimistic concurrency control for maintaining replication consistency. Referring to [47],
in asynchronous mode we use version-controlled documents. In a synchronous working mode it is shared among the users and we have to use pessimistic concurrency control for handling the concurrent access to shared data. The DSS exports the following services to its clients: i) open in order to open or to join a work section, ii) close in order to close or to exit from a work session\(^1\), iii) read in order to display the latest version of the shared document or of its partitions, iv) write in order to commit the own updates applied to the shared document or to its partitions and v) replicate in order to allow asynchronous writing by replicating the shared document or its partitions in a local copy. Referring to the version-controlled hierarchical documents [47], the local copy is an alternative and the globally shared document is the last document’s version. In an asynchronous writing scenario, the centralized version is used to maintain a safe state of the latest accepted global update on the document and its alternatives represent the current updates to be still accepted. Thus when a user wants to work in asynchronous mode, the DSS expects that all others users work in asynchronous mode as well. In this way, the DSS can maintain a consistent (with respect to the users updates) version of the globally shared document and it evolves in a new consistent version only after the merging of all users alternatives.

- **IU1**: Referring to [42, 43] this component is an integrated environment of many tools and it is for editing, navigation, display of awareness communication among the group members and import/export of data from external applications. It is composed by a CW user interface supporting all CW operations, editors for many data types, communication tools such as e-mail and chat. Each IU1 is a client component of CW E and DSS that, in turn, are clients for DBO.

According to our approach, let us suppose that the designer of the composed CW system provides a behavioral specification of the system. We recall that this specification is given in terms of bMSCs and HMSCs. For the sake of both simplicity and presentation, in our example, we consider only two instances of IU1: IU1\(_1\) and IU1\(_2\). The continued lines on the bMSCs are method calls; the hatched lines are the corresponding responses.

![Figure 6.2: bMSCs of REG and UNREG scenarios](image)

In Figure 6.2 we show the bMSCs representing the “user registration” scenario and the “user deregistration” scenario. These two scenarios are activated from the system manager in a setup phase when he inserts the information related to the working group members and when he deletes this information respectively.

In Figure 6.3 we show the bMSCs representing the “user login” scenario in both the cases of a successful login and of an unsuccessful login and the “user logout” scenario. A user can perform the login only if he has previously been registered by the manager.

In Figure 6.4 we show the bMSCs representing the “document’s structure, users group and access privileges definition” scenario, the “access privilege viewing” scenario and the “display of the history of the updates” scenario. ED is activated by the users with higher privileges in order to create or to edit a document’s structure, to create or to edit a users group and to edit the access privileges of the others users (i.e., the users with lower privileges); ACCSH is activated by a user in order to display the awareness information related to the access privileges of the other users in the group; HIST is activated by a user in order to display an history of all updates applied to a particular document.

\(^{1}\)A work session is truly closed when all the users have closed it.
In Figure 6.5 we show the bMSCs representing the “open work session” scenario in both the cases of a successful and an unsuccessful opening and the “close work session” scenario. In our CW system, a group-ware activity is a pair <work session, list of participants>. The list of participants is a previously defined users group. We recall that the component DSS can handle one or more shared documents that are all partitioned in document partitions. Each work session is associated to a shared document and each participant is associated to a defined document’s partition. A user can open a work session only if he has the access privileges on the shared document associated to the work session. Otherwise, the opening of the work session fails. In synchronous mode it is impossible to have more than one participant on the same partition. In asynchronous mode it might be possible to have two or more working participants on the same document partition. A work session is truly closed (i.e., DSS releases the resources kept busy by the shared document) only when all participants have closed it. In order to support physical mobility of a group member, a user can perform a logout even though he is a participant in a work session that he has not still closed. In this way although a user is not connected to the CW system the others users can know his document access information (e.g., which document or which partition the disconnected user is working on).

In Figure 6.6 we show the bMSCs representing the “shared data displaying” scenario, the “shared data synchronous updating” scenario, the “shared data asynchronous updating” and the “shared data replication for asynchronous writing” scenario. READ is activated by a user that has access privileges when he requires to display a shared document or an its partition. SYNCWR is activated by a user that has access
privileges when he requires to commit his synchronous updates to a shared document or to an its partition. 

ASYNCWR is activated by a user that has access privileges when he requires to commit his asynchronous updates to a shared document or to an its partition. REPL is activated from a user that has access privileges when he requires to replicate in a local copy a shared document or an its partition. Successively (see Figure 6.9), the user performs a logout in order to work on the shared document in asynchronous mode. Finally, the user performs a login in order to return in a synchronous working mode and to commit his updates applied to the shared document or to its partitions previously replicated.

In Figure 6.7 we show the HMSC specification for the composed CW system. This specification is expressed by using the HMSC specifications of the “configuration” sub-system of the CW system (i.e., HMSC_C) and of the “writing” sub-system of the CW system (i.e., HMSC_W) respectively.

In Figure 6.8 we show the HMSC specification of the “configuration” sub-system of the CW system.

In Figure 6.9 we show the HMSC specification of the “writing” sub-system of the CW system.

In Figure 6.10, we show the CBA-scheme of the above specified CW system in which, for the sake of simplicity, we consider only two instances of IUI (i.e., IUI1 and IUI2). In the CBA-scheme, for each functional component we denote its top and bottom interface. A top or a bottom interface is modeled as a set of service labels. Within a service label, the prefix “r_” denotes a request. On the other hand, the prefix “n_” denotes a notification. Let us suppose that the designer of the CW system requires the system to satisfy a particular coordination policy. The policy is specified in form of the Büchi automaton showed in Figure 6.11.

The double-circled state is the accepting state of the automaton (i.e., p_2) and the state with an incoming arrow is the initial state (i.e., p_0). The action labels of P_1 denote input actions. P_1 is a general progress
property related to a writing scenario. It specifies that the CW system must be always able to sequentially serve the two writing users (IUI$_1$ and IUI$_2$). In other words it means that if a user requires to commit an update on the shared document then (sooner or later) he will be always able to perform this commit.

In the reminder of this section we apply our approach in order to synthesize the property-satisfying assembly code for the components of the specified CW system. This assembly code is synthesized by deriving in an automatic way the models of two property-satisfying coordinators: i) the property-satisfying coordinator that assembles the components IUI$_1$, IUI$_2$, CWE and DSS; and ii) the property-satisfying coordinator that assembles the components CWE, DSS and DBO.

From the HMSC and bMSC specification, we can automatically derive the following behavioral models for each component in our CW system.

In Figure 6.12, we show the AC-Graphs of the components constituting the first layer of the CFA configuration of our CW system. Referring to Section 4.2, we consider the TAC-Graphs of IUI$_1$ and IUI$_2$ and the BAC-Graphs of CWE and DSS. Analogously in Figure 6.13 we show the BAC-Graph of component DBO and the TAC-Graphs of components CWE and DSS. The double-circled states are the initial states. Overlined labels denote output actions. On the other hand, plain labels denote in put actions. We recall that our approach proceeds by considering the two layers as two independent sub-systems of the CW-system. The sub-system corresponding to the first layer is built by assembling the components corresponding to TAC-Graphs of IUI$_1$ and IUI$_2$, and BAC-Graphs of CWE and DSS. The sub-system corresponding to the second layer is built by assembling the components corresponding to TAC-Graphs of CWE and DSS, and BAC-Graph of DBO.

**K2: THE COORDINATOR FOR THE SECOND LAYER**

From TAC-Graphs of CWE and DSS, and from BAC-Graph of DBO we can automatically derive

---

$^2$We recall that the specified CW system is an instance of a two-layered CFA.
the corresponding AS-Graphs. An AS-Graph is derived by simply applying the complement operator on all actions of the corresponding AC-Graph. From the AS-Graphs we derive the corresponding EX-Graphs and by unifying them we derive the coordinator graph for $K_2$.

In Figure 6.14, we show the EX-Graphs for the components constituting the second layer of the CBA CW-system and the synthesized coordinator graph $K_2$. Within each EX-Graph, an action such as $\alpha_?$, denotes an unknown action $\alpha$. The application of the approach to the second layer of the CW-system results trivial. Actually, $K_2$ is deadlock-free (i.e., it has no finite branches) and since $P_1$ is a behavioral property related only to the interaction among $IU_1$, $IU_2$ and $DSS$, then $P_1$ is trivially satisfied by $K_2$. Thus, in this case, the code derivation step of our approach does not derive a code implementing the coordinator $K_2$. In fact $K_2$ is only a “no-op” coordinator and hence it is a simple delegator of the requests of $CWE$ and $DSS$ toward $DBO$. Inserting $K_2$ in the system we can only introduce overhead. In next section we describe a more interesting application of our approach.

**K1: THE COORDINATOR FOR THE FIRST LAYER**

In this section we apply our approach in order to synthesize the code implementing the coordinator $K_1$. $K_1$ is the coordinator for the first layer of the CW-system. We want to enforce the property $P_1$ against $K_1$. From TAC-Graphs of $IU_1$ and $IU_2$, and BAC-Graphs of $CWE$ and $DSS$ we derive the corresponding AS-Graphs. Then from the AS-Graphs we derive the corresponding EX-Graphs.

In Figure 6.15, we show the EX-Graph of the component $IU_1$. The EX-Graph of $IU_2$ is different only in the identifier of the component specified in known action labels. This component identifier is 2 instead of 1. In Figure 6.16, we show the BEX-Graphs (bottom-domain EX-Graphs) of $CWE$ and $DSS$. For the sake of presentation, we reduce the sub-system constituting the first layer of our CW-system. Actually for the reminder of this section we consider a sub-system made of the only components $IU_1$, $IU_2$ and $DBO$. Thus we continue with the application of our approach by only considering the EX-Graphs of $IU_1$, $IU_2$ and the BEX-Graph of $DBO$. By unifying the above considered EX-Graphs we can derive the coordinator graph showed in Figure 6.17.

---

**Figure 6.8: HMSC of the “configuration” sub-system of the CW system**

---
The $i^{th}$ generated node of the coordinator graph is annotated as $k_i$ and its label is reported in figure. In order to not further complicate Figure 6.17, we have showed in Figure 6.18 the sub-graph $K_{1,1}$ represented by the gray triangle. The start node for the sub-graph $K_{1,1}$ is labeled with $k_{24}$.

For sake of simplicity, we further reduce our application to the only coordinator represented by the sub-graph $K_{1,1}$ of $K_1$. The coordinator $K_{1,1}$ has two deadlocks represented by two finite branches. In order to synthesize the deadlock-free version of $K_{1,1}$ we simply prune the two finite branches. These deadlocks are related to the consistency maintenance in an asynchronous writing scenario. They put in evidence one of the possible problems when we assemble COTS components. We recall that in order to maintain a consistent version of the shared document, the DSS expects that all users work in asynchronous mode when another user wants to work in asynchronous mode too. On the other side the third-party components IUI$_1$ and IUI$_2$ do not respect this DSS assumption. Thus, the composed system deadlocks. The coordinator $K_{1,1}$ without finite branches (i.e., deadlock-free $K_{1,1}$) forces IUI$_1$ and IUI$_2$ in order to respect the DSS assumption (i.e., it forces all users in order to work in asynchronous mode when another user wants to work in asynchronous mode too). Once we have obtained the deadlock-free version of the coordinator, the approach goes to the property enforcing step. Trivially $B_{K_{1,1}}$ is equal to $K_{1,1}$ with all nodes marked as accepting states. Then the method builds the automaton $B_{K_{1,1};P_1}^{intersection}$ accepting the language $L(B_{K_{1,1}}) \cap L(P_1)$ in order to synthesize the $P_1$-satisfying behaviors of the deadlock-free $K_{1,1}$ (i.e., the execution paths of $B_{K_{1,1}}$ that are execution paths of $B_{P_1}$ too). Finally in order to synthesize the "fair"$^3$ $P_1$-satisfying deadlock-free connector, our method extracts the sub-automaton of $B_{K_{1,1};P_1}^{intersection}$ that contains the only accepting cycles of $B_{K_{1,1};P_1}^{intersection}$ as execution paths. In Figure 6.19 we have showed this sub-automaton. It corresponds to the behavioral model for the $P_1$-satisfying starvation and deadlock-free $K_{1,1}$.

It is worthwhile noticing that this sub-automaton is a model for the only coordinator behavior which is related to the deadlock-free property, starvation property and $P_1$. This behavioral model is enough in order to derive the (deadlock and starvation-free) $P_1$-satisfying coordinator actual code. For the methods that have not influence on the specified property the coordinator is a simple delegator, thus the implementing code is a simple request delegation. The following is the (deadlock and starvation-free) property-satisfying code implementing the methods write and replicate of the coordinator component $K_{1,1}$. We recall that we deal

---

$^3$With respect to starvation property.
with COM components and we use Visual C++ with ATL as programming environment. Let us suppose that the method `write` of the DSS object gets a parameter of type `S_DA`. `S_DA` is a document alternative `struct`. It contains information about the document update to be committed. The method `replicate` of the DSS object gets a parameter of type `S_DA`. It contains information about the document portion to be replicated. Moreover, `replicate` returns a parameter of type `LCOPY`. It contains the replicated local copy of the specified document’s alternative.

```c++
HRESULT write(S_DA da) {
    HRESULT result;
    if(sLbl == 24) {
        if((chId == 1) && (pState == 0)) {
            result = dssObj->write(da);
            pState = 1; sLbl = 24;
        } else if((chId == 2) && (pState == 1)) {
            result = dssObj->write(da);
        } else if((chId == 3) && (pState == 2)) {
            result = dssObj->write(da);
        }
    }
    // Other cases...
}
```
Figure 6.12: TAC-Graph of components $IUI_1$, $IUI_2$ and BAC-Graphs of components $CWE$, $DSS$

Figure 6.13: BAC-Graph of component $DBO$ and TAC-Graphs of components $CWE$, $DSS$

```
pState = 0; sLbl = 24;
}
else if(sLbl == k56) {
    if((chId == 1) && (pState == 0)) {
        result = dssObj->write(da);
        pState = 1; sLbl = 44;
    }
    else if((chId == 2) && (pState == 1)) {
        result = dssObj->write(da);
        pState = 0; sLbl = 42;
    }
}
else if(sLbl == 42) {
    if((chId == 1) && (pState == 0)) {
        result = dssObj->write(da);
    }
}
```
Figure 6.14: BEX-Graph of component DBO, TEX-Graphs of components CWE, DSS and 2nd-layer coordinator graph (K2)

```c
pState = 1; sLbl = 24;
}
}
else if(sLbl == 44) {
    if((chId == 2) && (pState == 1)) {
        result = dssObj->write(da);
        pState = 0; sLbl = 24;
    }
}
return result;
}

HRESULT replicate(S_DA da, LCOPY* plc) {
    HRESULT result;
    if(sLbl == 24) {
        if((chId == 1) && (pState == 0)) {
            result = dssObj->replicate(da, plc);
            pState = 0; sLbl = 42;
        }
        else if((chId == 2) && (pState == 0)) {
            result = dssObj->replicate(da, plc);
            pState = 0; sLbl = 44;
        }
```
else if((chId == 1) && (pState == 1)) {
    result = dssObj->replicate(da, plc);
    pState = 1; sLbl = 42;
}
else if((chId == 2) && (pState == 1)) {
    result = dssObj->replicate(da, plc);
    pState = 1; sLbl = 44;
}
else if(sLbl == 42) {
    if((chId == 2) && (pState == 0)) {
        result = dssObj->replicate(da, plc);
        pState = 0; sLbl = 56;
    }
    else if((chId == 2) && (pState == 1)) {
        result = dssObj->replicate(da, plc);
        pState = 1; sLbl = 56;
    }
}
else if(sLbl == k44) {
    if((chId == 1) && (pState == 0)) {
        result = dssObj->replicate(da, plc);
        pState = 0; sLbl = 56;
    }
    else if((chId == 1) && (pState == 1)) {
        result = dssObj->replicate(da, plc);
        pState = 1; sLbl = 56;
    }
}
6.1 Automatic coordinator synthesis for a COTS group-ware system

This code is automatically synthesized by visiting the sub-automaton of Figure 6.19 and by exploiting the information stored in its states and transitions labels. For the others methods (i.e., open, close and read) the implementing code consists of a delegation of the request (corresponding to the considered method) toward the inner object dssObj. The coordinator component $K_{1.1}$ is an aggregated server component that encapsulates as private member an instance of an inner DSS component (i.e., dssObj into the implementing code). It exports the services corresponding to the automatically synthesized methods open, close, read, write and replicate. Moreover, it handles two clients: i) $IUI_1$ and ii) $IUI_2$. Thus the coordinator component $K_{1.1}$ implements the COM interface IDSS of the DSS component by defining a COM class $K_{1.1}$ and by implementing a wrapping mechanism in order to wrap the requests that $IUI_1$ and $IUI_2$ perform on DSS component. The following are fragments of the IDL (Interface Definition Language) definition for $K_{1.1}$ COM library and of the $K_{1.1}$ COM class respectively.

import idss.idl; ... library K1_1_Lib {

... coclass K1_1 {
  [default] interface IDSS;
}

... class K1_1 : public IDSS {

return result;
}

Figure 6.17: The reduced 1st-layer coordinator graph ($K_1$)

// stores the current state of the connector
private static int sLbl;

// stores the current state of the
// property automaton
private static int pState;

// stores the number of clients
private static int clientsCounter = 0;

// channel’s number of a client
private int chId;

// COM smart pointer; is a reference to
// the DSS object
private static DSS* dssObj;

// the constructor
K1_1() {
    sLbl = 24;
    pState = 0;
    clientsCounter++;
}
6.2 Automatic component assembly for a Product Data Management cooperative system

Integrating a system with reusable software components or with COTS components introduces a set of problems. One of the main problems is related to the ability to properly manage the dynamic interactions of the components. In the area of CSCW (Computer Supported Cooperative Work) [32, 42, 43], the management of dynamic interactions of the components forming the application can become very complex in

```java
chId = clientsCounter;
dssObj = new DSS();
...
}
// implemented methods
...
}
```

In order to enforce a new behavioral property we have to repeat the enforcing step. In this way we synthesize a new implementing code that we have to add (or to accommodate with) to the already synthesized code in order to implement the both coordination policies (i.e., the one related to the old property and the one related to the new property).

6.2 AUTOMATIC COMPONENT ASSEMBLY FOR A PRODUCT DATA MANAGEMENT COOPERATIVE SYSTEM

Integrating a system with reusable software components or with COTS components introduces a set of problems. One of the main problems is related to the ability to properly manage the dynamic interactions of the components. In the area of CSCW (Computer Supported Cooperative Work) [32, 42, 43], the management of dynamic interactions of the components forming the application can become very complex in
order to prevent and avoid undesired interactions. A CSCW application constitutes an integrated environment formed by one or more CSCW servers and many CSCW clients [42, 43]. In general, both servers and clients are heterogeneous components built by different organizations of software development. CSCW servers are black-box components providing the main functionalities concerning a cooperative activity (e.g., repository management functionalities as data check-in and check-out, data persistence, concurrent accesses to data, group-aware information management, etc.). CSCW clients are black-box and third-party components which exploit the services provided by the CSCW servers in order to execute a group-aware task.

Typically, the servers are sold by the vendor of the CSCW framework. The clients are used by the customer organization of the CSCW framework which is the organization acquiring the CSCW framework on the market. A very important issue concerns the integration between the CSCW servers and CSCW clients. Depending on the customer organization and on the typology of its product manufacture, a CSCW client can be a text editor or a spreadsheet or a computer aided design (CAD) application. Given the huge diversity of applications that could be clients of a CSCW server, it is worthwhile noticing that a CSCW server has to be integrated with a CSCW client by following ad-hoc strategies. This means that once the customer acquires the servers forming the CSCW framework, the vendor will have to implement the code integrating the clients with the servers. Moreover the vendor will have to repeat this heavy phase of deployment of the CSCW framework for each different customer. Thus an issue of great influence on the good setting of the vendor on the market concerns the full automation of the phase of integration code development.

In this section, by exploiting our approach, we describe our experience in the automatic derivation of the assembly code for a set of components forming a product data management (PDM) cooperative system. The PDM system we refer to has been developed by Think3 company [1] in Bologna, ITALY. We use our
SYNTHESIS tool to automatically derive the code integrating components forming that PDM system. This system is called ThinkTeam. ThinkTeam has been developed by using Microsoft Component Object Model (COM) \[55\] with Active Template Library (ATL) in Microsoft Visual Studio development environment. ThinkTeam is a PDM solution that provides a solid platform for a successful product life-cycle management implementation. For engineering departments, ThinkTeam provides the data and document management capabilities required to manage all product documentation, including 3D models, 2D drawings, specifications, analysis and test results. Multiple users will always have access to updated, released and work in progress product information. Also provided is a changes management solution that enables engineers to interface and communicate with the rest of the organization. ThinkTeam is packaged into five modules to provide specific capabilities and solution features. For the purposes of this section the module we are interested on is the ThinkTeam client (TTClient) component. This is a stand-alone application that is integrated into a CAD application and Microsoft Office applications and provides features to manage documents, versions, data attributes, and relationships among documents. We are interested in applying our approach in order to automatically derive the integration code assembling the TTClient component with the distributed instances of the third-party CAD application. This code is derived to force the composed system to satisfy the coordination policies that will be described later.

6.2.1 ThinkTeam architecture

In Figure 6.20 we show a ThinkTeam network.

The following are the interacting components in a ThinkTeam network:

- the TTClient component which provides general purposes PDM functionalities such as documents, multiple users, and changes management, versions controlling, data attributes and relationships among documents management. ThinkTeam owns the documents and metadata flow and mediates between
Chapter 6. Case studies

the applications and the stores. *Metadata is data-about-data*, recording things such as custom attributes, relations among items (i.e., product structure), historical data (i.e., document revisions). The backing store is the *RDBMS* component;

- the distributed applications used by the *ThinkTeam*’s customer organization. Depending on the kind of customer organization manufacture, these applications can be either a CAD application or a text editor application (such as Microsoft Word or a spreadsheet application such as Microsoft Excel) or any other kind of application managing the product data;

- a centralized repository (i.e., the *Vault*) for the documents related to the products of the customer organization and for the relationships among hierarchical documents. A hierarchical document can be either a document with all information itself contained or a document making use of references to other hierarchical documents. *Vault* operations pertain to document data and allow reservation (i.e., check-out), publishing (i.e., check-in) and unreserved read access. The backing store is a filesystem-like entity. Actually an entity corresponds to a file into *Vault*.

- a centralized repository (i.e., the *RDBMS*) for the *metadata*.

In a *ThinkTeam* network, the *TTClient* component manages many types of data related to documents (e.g., 3D models, 2D drawings, data sheets, technical reports, etc.), to the work flow (e.g., processes, activities, automatic actions, etc.), to product parts (e.g., assemblies, parts, technical bill of materials, production bill of materials, etc.) and to the organization. (e.g., users, groups, roles, authorizations, etc.). All these data are classified in terms of *entity* types and their attributes. Thus an *entity* represents any data which has an associated set of attributes. *TTClient* is extendable. Actually a *ThinkTeam* client allows the adaption of its structure on customer needs, both in terms of entity types and entity attributes.

### 6.2.2 *ThinkTeam/Application Integration Schema*

As showed in Figure 6.20, the distributed applications used by the customer share an instance of the *TTClient* component. One of the goals of *Think3* company is to automatically derive the code integrating the *TTClient* component with a particular application which, in our case study, is a CAD system. This is an important goal for the company because an automatic and correct integration of *TTClient* with the customer application makes the *ThinkTeam* system more competitive on the market. In Figure 6.21, we show the integration schema for *TTClient* component and the CAD system used by the customer organization.

The integration schema of Figure 6.21 represents the CBA version of the system we are considering for the case study of this section. On the left side we show the component provided by *Think3*. It is the *TTClient* black-box component plus an auxiliary component (i.e., the integration component on the *TTClient* side) which has the only function to export to its clients the services provided by the *TTClient* component. In the following of this section, we consider this composite component (i.e., *TTClient* plus the auxiliary component) as a single component called *ThinkTeam* component. *ThinkTeam* component is a black-box component which provides to its clients a specified set of services. The following is the subset of all services provided by the *ThinkTeam* component relevant to our purposes:

- **afterinit**: this method has to be called when the application integrated with the *ThinkTeam* component is completely initialized;

- **checkout**: locks the specified file into *Vault* for writing operations;

- **checkin**: releases the lock activated for writing operations on the specified file into *Vault*;

- **get**: gets a local copy of a file;

- **import**: copies a local file into *Vault*;
6.2 Automatic component assembly for a Product Data Management cooperative system

ThinkTeam client
integration component (TTClient side)

 coordinator component

Application (CAD)
third-party software

Figure 6.21: Integration between ThinkTeam and the customer’s application

- **getattrib**: obtains a read only copy of the attributes of an entity into Vault;
- **setvalue**: sets/modifies the value of a certain entity attribute;
- **setvalues**: set/modify all entity attributes values;
- **remove**: removes the entity from Vault;
- **start**: starts-up the integration between ThinkTeam and the application used by the customer;
- **stop**: shuts-down the integration between ThinkTeam and the application used by the customer.

On the right side of Figure 6.21 we show the application used by the customer organization. In our case the customer’s application is a CAD system and the ThinkTeam component has to be integrated into it. In the following of this section, we refer to the customer’s application as CAD component. The CAD component is a black-box component which provides to its clients the following services.

- **ttready**: this method has to be called when the ThinkTeam component integrated into the customer’s application is completely initialized;
- **openfile**: open a file;
- **save**: save the changes made on a file;
- **closefile**: closes a file.

Between ThinkTeam and CAD components we show the coordinator component whose aim is to integrate the ThinkTeam component and the CAD component. The coordinator mediates the interaction between ThinkTeam and the distributed instances of CAD by following specified coordination policies. We use SYNTHESIS to automatically derive the code implementing the coordinator component. We start from the following input data: i) a bMSCs and HMSCs specification of the CFA version of the system to be assembled; ii) MIDL and Type Libraries files for ThinkTeam and CAD components; iii) a Büchi automata specification of the desired coordination policies. In Section 6.2.3, we show the application of SYNTHESIS to the automatic and correct integration of ThinkTeam and CAD components.
6.2.3 SYNTHESIS FOR INTEGRATING THINKTEAM AND CAD

In this section we apply SYNTHESIS in order to automatically derive the code of the coordinator component showed in Figure 6.21. This code is derived in order to limit all possible interactions among ThinkTeam and CAD to a subset of deadlock-free interactions corresponding to a set of specified coordination policies. In Figure 6.22 we show the bMSCs representing the execution scenarios of the composed system formed by ThinkTeam (i.e., TT in Figure 6.22) and CAD components.

![Figure 6.22: bMSC specification for the CFA version of ThinkTeam/CAD system](image)

The scenarios in Figure 6.22 are defined in terms of the messages exchanged between ThinkTeam and CAD. These messages correspond to the methods provided by ThinkTeam and CAD components. In Figure 6.23 we show the HMSC specification of the composed system formed by ThinkTeam and the CAD components.

![Figure 6.23: HMSC specification for the CFA version of ThinkTeam/CAD system](image)

The HMSC of Figure 6.23 is defined in terms of three nested HMSCs (i.e., H_WRITE, H_READ and H_ADD_REMOVE showed in Figure 6.24). In all HMSCs we have showed in Figures 6.23 and 6.24, each
bMSC is reachable from every other bMSC into the HMSC. For the sake of brevity, we do not show the MIDL files for ThinkTeam and the CAD components. This is not a limitation for the understanding of the case study. Actually by referring to Section 6.2.2, we can consider as MIDL files for ThinkTeam and CAD the two set of services provided by ThinkTeam and CAD respectively. The .tlb files for ThinkTeam and CAD are binary files and they are used internally to SYNTHESIS. In addition to the bMSC and HMSC specification plus the MIDL and .tlb files, SYNTHESIS needs to know how many instances of ThinkTeam and CAD components have to be considered. This information is provided by interacting with a dialog control of the SYNTHESIS’s user interface. In our case study, we consider two instances of the CAD component sharing an instance of the ThinkTeam component. From this additional information (i.e., components instances) and from the two input data considered above (i.e., i) bMSC and HMSC specification and ii) MIDL + .tlb files), SYNTHESIS is able to automatically derive an AC-Graph for each component’s instance forming the specified composed system. Figure 6.25 is a screen-shot of SYNTHESIS in which we show the automatically derived AC-Graph for the instance of ThinkTeam component.

For the sake of brevity we do not show the AC-Graph for the two instances of CAD component (i.e., C1 and C2432 on the left panel of the SYNTHESIS’s user interface in Figure 6.25). The last input data for SYNTHESIS is the B¨uchi automata specification of the coordination policies to be enforced on the composed system through the automatically synthesized coordinator component. The coordination policy we want to enforce in our case study is the following: “a document cannot be removed if someone has checked it out. Moreover, the attributes cannot be modified if someone is getting a copy of the entity as a reference model.” Figure 6.26 is a screen-shot of SYNTHESIS in which we show the provided automaton for the above coordination policy.

Informally, the automaton in Figure 6.26 describes a set of desired behaviors for the composed system formed by the ThinkTeam’s instance and the two CAD’s instances in parallel under the point of view of a hypothetical observer. Each node is a state of the observed composed system. The node with the incoming...
Figure 6.25: AC-Graph of ThinkTeam component

arrow is the initial state. The black nodes are the accepting states. Once the automaton execution reaches an accepting state, it restarts from the initial state. We recall that each transition label (except for a particular kind of transition) is postfixed by "." followed by a number. This number is an identifier for a component’s instance. Referring to Figure 6.26, 1 identifies an instance of the CAD component, 2432 identifies the other instance of the CAD component and 2 identifies the instance of the ThinkTeam component. From the AC-Graph for each component and from the automaton of the desired coordination policy, SYNTHESIS derives the model of the coordinator component that assembles ThinkTeam and the two CADs by implementing the coordination policy of Figure 6.26. For the sake of brevity, we do not show the model of the coordinator synthesized before the coordination policy enforcing step. Figure 6.27 is a screen-shot of SYNTHESIS in which we show the model of the coordinator after the coordination policy enforcing step.

The sink filled states identify the achievement of a desired behavior. Once the coordinator component’s execution achieves a desired behavior, it restarts from the initial state. From this model by exploiting the information stored in each node and arc, SYNTHESIS automatically derives the code implementing the policy-satisfying coordinator. This code implements an ATL COM component. It is constituted by a MIDL file (.idl), an ATL header file (.h) and an ATL source file (.cpp). In order to produce the code implementing the COM coordinator component SYNTHESIS uses also the MIDL and .tlb files for ThinkTeam and CAD provided in input. In the following we only report the meaningful parts of the implementing code of the ATL COM coordinator component produced by SYNTHESIS. For the sake of brevity we do not report the MIDL code (.idl) specifying the interface of the ATL COM coordinator component. That interface is defined by exploiting the interfaces of ThinkTeam and CAD components. The following is the ATL header file (.h):

```cpp
#include "TTIntegrator.tlb" #import "CAD.tlb" ... class ATL_NO_VTABLE TTConnector : ... { public:
    TTConnector() {
```
6.2 Automatic component assembly for a Product Data Management cooperative system

Figure 6.26: A coordination policy

```cpp
sLbl = S4640_S11760;
clientsCounter++;
chId = clientsCounter;
ttObj = new ThinkTeam();
cadObj = new CAD();
```

```cpp
... public:
  //implemented methods
  STDMETHOD(get)(...);
  STDMETHOD(checkout)(...);
  STDMETHOD(checkin)(...);
  STDMETHOD(afterinit)(...);
  STDMETHOD(import)(...);
  STDMETHOD(getattrib)(...);
 DMETHOD(setvalue)(...);
 DMETHOD(setvalues)(...);
  STDMETHOD(remove)(...);
  STDMETHOD(start)(...);
  STDMETHOD(stop)(...);
  STDMETHOD(ttready)(...);
  STDMETHOD(openfile)(...);
  STDMETHOD(save)(...);
  STDMETHOD(closefile)(...);
private:
  //stores the current state of the connector
  static int sLbl;
  //stores the number of connector clients
```
The class declaration for the ATL COM coordinator component exploits the class declarations of the ThinkTeam and CAD components. The ATL COM coordinator component encapsulates references to ThinkTeam and CAD objects and uses a set of private members in order to identify a caller of a service and to store the state reached during the execution. This is needed in order to reflect the behavior of the coordinator model showed in Figure 6.27. The following is the ATL source file (.cpp):

```cpp
... STDMETHODIMP TTConnector::get(...) {
    HRESULT res;
    if(sLbl == S4640_S11760)
    {
        if(chId == 1) // it corresponds to an instance of CAD
        {
            res = ttObj->get(...);
            sLbl = S5007_S12997;
            return res;
        }
        else if(chId == 2) // it corresponds to the other instance of CAD
```
6.3 Automatic synthesis of adaptors for a cooling water pipe system

```c
{  
    res = ttObj->get(...);
    sLbl = S5055_S12733;
    return res;
}

return E_HANDLE;
```

For the sake of brevity, we have only reported the code for the `get` coordinator method. It reflects the structure of the model in Figure 6.27. All other methods are synthesized analogously to the `get` method. The only difference is that while `get`, `setvalue`, `setvalues`, `remove`, `checkout` and `checkin` contain delegations of the corresponding methods on the `ThinkTeam` object (i.e., `ttObj`), the methods `openfile` and `closefile` contain delegations of the corresponding methods on the `CAD` object (i.e., `cadObj`). All remaining methods (i.e., `afterinit`, `import`, `getattrib`, `start`, `stop`, `ttready` and `save`) are synthesized as simple delegations toward the object (i.e., `ttObj` or `cadObj`) which provides them.

6.3 AUTOMATIC SYNTHESIS OF ADAPTORS FOR A COOLING WATER PIPE SYSTEM

This case study concerns the (semi-)automatic assembly of a cooling water pipe system that collects and correlates data about the amount of water that flows in different water pipes. The water pipes are placed in two different zones, denoted by `P` and `S`, and they transport water that has to be used to cool industrial machinery.

The zone `P` (or `S`) is monitored by the server `CPHandler` (or `CSHandler`). `CPHandler` (or `CSHandler`) supports cooperative work and allows the access to a collection of data related to the water pipes it monitors. `CPHandler` (or `CSHandler`) implements the interface `IPHandler` (or `ISHandler`). Since some of the water pipes do not include a Programmable Logic Controller (PLC) system, the two servers cannot always automatically obtain the data related to the water that flows in those water pipes. Therefore, `IPHandler` (or `ISHandler`) provides the method `WFRCheckOut` to get an exclusive access to the data collection related to the water which flows in the pipes. This allows a client to: (i) read the data automatically stored by the server and (ii) manually update the report related to the water which flows in the pipes that are not monitored by a PLC. Correspondingly, `IPHandler` (or `ISHandler`) provides the method `WFRCheckIn` to both publish the updates made on the data collection and release the access gained to it. `IPHandler` (or `ISHandler`) does not provide an authentication service for its clients.

It is worthwhile noticing that `IPHandler` (or `ISHandler`) exports a certain set of others methods which serves to accomplish tasks that are not relevant for the purposes of the case study. For the sake of presentation, in the following we show an application of our approach only related to the system’s interaction behavior of interest and we omit a complete description of the case study. It is required to bridge possible component incompatibilities only for a sub-set of the system behaviors’ set. For the complementary sub-set (which is “integration-incompatibility-free”) our synthesized interaction mechanisms behave like simple delegators of requests and notifications from a component to another one.

We want to apply our SYNTHESIS tool in order to assemble a client-server cooling water pipe system formed by `CPHandler`, `CSHandler` as servers and two clients: `Client1` and `Client2`. The two servers and the two clients are distributed by two different organizations. That is, `CPHandler`, `CSHandler`, `Client1` and `Client2` are black-box components developed by third-parties. `Client1` and `Client2` require an interface that exports two methods: (i) `open` to get the access to the data collection related to the resource that the servers...
handle and (ii) save to save updates on that data collection and release the access gained for it. By referring to the purposes of the case study we have to face two problems:

1. **achieving interface-compatibility**: although Client1 and Client2 require an interface different to the one provided by CPHandler and CSHandler, we have to make the clients able to communicate with the servers;
2. **enhancing dependability**: we want to add security by augmenting the interaction protocol of the composed system in order to support clients authentication.

By referring to Figure 6.28, we address the first problem by using our SYNTHESES tool to automatically synthesize a deadlock-free coordinator $K1$ which makes the two servers and the two clients able to communicate avoiding possible deadlocks (i.e., Step 1). We address the second problem by using SYNTHESES to semi-automatically synthesize two wrappers (each one of them for each server) that exploit the "Kerberos Client API" \[63\] in order to add an authentication service for avoiding the access by non-authorized clients (i.e., Step 2). By referring to [63], the "Kerberos Client API" allows to interact with the "Kerberos Server" in order to get a "Granting Ticket" which identifies an authorized client. The "Kerberos Server" handles a data base which represents the list of authorized clients.

As shown in Figure 6.28, to insert the wrappers by maintaining deadlock-freedom, SYNTHESES automatically derives four coordinators (each two of them for each wrapper) interposed on the top and on the bottom of the two wrappers respectively. The two wrappers are inserted in the system by applying two enhancements in a compositional way. It is worthwhile noticing that in Figure 6.28 we show a CFA without connectors since its components have incompatible interface’s signatures.

### 6.3.1 SYNTHESES AT WORK

We start by taking into account the bMSC and HMSC specification of the components forming the CFA, as showed in Figure 6.29, and their IDL (Interface Definition Language) files that we omit for the sake of brevity.

By referring to Figure 6.29, (a) denotes the HMSC specification of CPHandler. This HMSC combines only one bMSC called $S1$ and denoted as (c) in figure. (a) specifies that from the initial state (i.e., $start$) CPHandler performs the scenario $S1$ and then continues to perform it infinitely often.

(c) specifies that during the execution of the scenario $S1$, CPHandler receives from one instance of its environment (i.e., $ENV1$) a request of $WFRCheckOut$ followed by a request of $WFRCheckIn$. Since CSHandler has the same MSCs specification we omit it.

\[6\] With respect to its signature.
6.3 Automatic synthesis of adaptors for a cooling water pipe system

Figure 6.29: bMSCs and HMSCs of the components forming the CFA

Figure 6.30: AC graphs of the components forming the CFA

(b) denotes the HMSC specification of Client1 which combines two bMSCs called S1 and S2 and denoted as (d) and (g) in figure respectively. In these bMSCs, Client1 interacts with two different instances of its environment (i.e., ENV1 and ENV2 respectively). SYNTHESIS interprets each distinct environment’s instance in a bMSC S of a component Ci as a component’s instance Cj (j ≠ i) which is a part of the environment of Ci and interacts with Ci as specified by S. Since the HMSC of Client2 is equal to the one given for Client1 we omit it. The scenario S1 in the HMSC of Client2 corresponds to the scenario denoted as (e) in figure. While S2 in the HMSC of Client2 corresponds to (f).

From the bMSC and HMSC specification showed in Figure 6.29, SYNTHESIS automatically derives the AC graphs for each component forming the CFA as showed in Figure 6.30. From these AC graphs, SYNTHESIS automatically derives the coordinator graph showed in Figure 6.31.

The coordinator K1 maps the requests of open and save performed by both the clients to the requests of WFRCheckOut and WFRCheckIn respectively. The filled nodes denotes all the states that are elements of the so called “deadlocking paths”. Deadlocks might occur because of a race condition due to the fact that one client (e.g., Client1) locks the data collection related to the zone S and waits for accessing to the one related to P while the other client (e.g., Client2) locks the data collection related to S and waits for the one related to P. Since each client does not release the locked data collections before having used both of them, a deadlock occurs. SYNTHESIS simply prunes the deadlocking paths of the coordinator graph in order to obtain the deadlock-free equivalent one.

From the deadlock-free coordinator graph, SYNTHESIS semi-automatically\(^5\) derives the actual code implementing the corresponding adaptor/coordinator COM/DCOM component.

\(^5\) Only in the case of incompatible interfaces, the parameters list of a request \(r1\) might not correspond to the one of a request \(r2\) which is mapped to \(r1\). Thus the SYNTHESIS’s user has to provide that parameters list mapping by interacting with the tool.
Now we proceed by taking into account the bMSC and HMSC specification of the first enhancement (i.e., the first protocol enhancement-policy) to be applied on the deadlock-free coordinator protocol in order to add a clients’ authentication service with respect to the server \textit{CPHandler}. For the sake of brevity, we omit a description of the application of the second enhancement (i.e., the one with respect to the server \textit{CSHandler}) since it is applied analogously to what we do for the first one. The Figure 6.32 shows the bMSC and HMSC specification of the first enhancement. By referring to Figure 6.32, (a) denotes the HMSC specification of the wrapper \textit{W1} showed in Figure 6.28. This HMSC combines two bMSCs called \textit{S1} and \textit{S2} and denoted as (b) and (c) in figure respectively. (b) specifies the scenario for a successful authentication of a \textit{CPHandler}’s client, while (c) specifies the one for a non-authorized \textit{CPHandler}’s client. \textit{W1} “wraps” the requests of the \textit{CPHandler}’s clients and, before establishing whether to delegate them or not, exploits \textit{kclnt32.dll} to authenticate the client which has performed the request. Note that, for a non-authorized client, \textit{W1} does not delegate to \textit{CPHandler} the received request. Sub(K, 1) denotes the sub-coordinator \textit{K1} where 1 was the channel connecting \textit{CPHandler} to \textit{K1} in the initial CBA. By performing the approach described in Section 4.5, from the LTS of \textit{K1}, SYNTHESIS first derives the LTS of the decoupled sub-coordinator \textit{K11_f19}. Second, by exploiting the layered structure of the imposed architectural model, from the first enhancement MSC specification, SYNTHESIS automatically derives the LTSs of \textit{W1_BOTTOM} (i.e., \textit{W110}), \textit{W1_TOP} (i.e., \textit{W116}) and \textit{kclnt32.dll}. Third, from the LTSs of \textit{K11_f19} and \textit{W1_BOTTOM}, SYNTHESIS automatically derives the LTS of the deadlock-free coordinator \textit{K2bottom} (i.e., \textit{K1'}). From the LTSs of \textit{CPHandler}, \textit{kclnt32.dll} and \textit{W1_TOP}, SYNTHESIS automatically derives the LTS of the deadlock-free coordinator \textit{K2top} (i.e., \textit{K1'}). In Figure 6.33 we show all these LTSs as derived by SYNTHESIS.

Before inserting \textit{W1} in the system through the actual code synthesized for \textit{K2top} (i.e., \textit{K1'}) and \textit{K2bottom} (i.e., \textit{K1''}), SYNTHESIS checks if the behavior of \textit{W1} is consistent with respect to the behavior of \textit{K1} (i.e., if the first enhancement is behavioral consistent with respect to the behavior of the initial CBA). In so doing, by exploiting the Theorem 1, SYNTHESIS builds the product language of the languages accepted.
6.3 Automatic synthesis of adaptors for a cooling water pipe system

by the Büchi Automata corresponding to the LTSs of $K_1''$ (i.e., $L(K_1'')$ in Figure 6.33) and the negation of $K_1'[f_{env}']$ (i.e., $L(K_1'[f_{env}'])$ in Figure 6.33) and checks its emptiness. Since such product language is empty (by looking at the Büchi Automata of $L(K_1'')$ and $L(K_1'[f_{env}'])$ showed in Figure 6.33, is easy to see that they accept two perfectly complementary languages), the wrapper component $W_1$ is inserted in the system in such a way that deadlock-freedom is maintained. This is done, by means of the deadlock-free coordinators $K_2^{top}$ (i.e., $K_1'$) and $K_2^{bottom}$ (i.e., $K_1''$) as showed in Figure 6.28. In this way, by using SYNTHESIS we correctly and semi-automatically applied the first enhancement specified by the protocol enhancement-policy of Figure 6.32.

By referring to Figure 6.28, the parallel composition $K_{new}$ (i.e., in CCS, $(K_1' | W_1 | K_1'' | K) \setminus (Act_{10} \cup Act_{16} \cup Act_9)$ where $Act_k$ is the set of actions performed on the connector $k$) represents the model of the enhanced coordinator. The LTS of $K_{new}$ is automatically synthesized by SYNTHESIS, as usual, by taking into account the LTSs of $kclnt32.dll$, $CPHandler$, $CSHandler$, $Client1$ and $Client2$. $K_{new}$ is considered as a new initial glue code for the first enhanced CBA. It can be further enhanced by applying on its protocol the second enhancement (as illustrated in Figure 6.28) analogously to what we have done for the first one.

Figure 6.32: bMSCs and HMSC of the first protocol enhancement policy

Figure 6.33: Behavioral models related to the first enhancement
on the protocol of $K$. By considering $K_{\text{new}}$ instead of $K$, the application of the second enhancement is completely similar to the application of the first one.
CHAPTER 7

RELATED WORK

The architectural approach to the automatic composition and adaptation of software components presented in this thesis is related to a number of other approaches that have been considered by researchers. In this chapter, we briefly describe only the works closest to our approach.

7.1 SUPERVISORY CONTROL THEORY FOR SYNTHESIZING SCHEDULERS

The most strictly related approaches are in the “scheduler synthesis” research area. In the discrete event domain they appear as “supervisory control” problem [6, 15, 56, 68, 69].

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 these approaches the possible system executions are modeled as a set of event sequences, 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 approach is to restrict the system behavior so that it is contained in a desired behavior, called the specification. To do this, the system is constrained to perform events only in strict synchronization with another system, called the supervisor (or controller). This is achieved by automatically synthesizing a suitable supervisor with respect to the system specification.

Differently from 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 behaviors 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 behaviors since they might be unpredictable. Moreover, the synthesis of supervisors is only focused on restricting the set of system behaviors to a subset of desired behaviors. The approach we present in this thesis goes beyond simply restricting the set of system behaviors. In fact it is also able to augment the set of system behaviors in order to introduce more sophisticated interactions among the components forming the system allowing to add, for example, missing functionality.

7.2 INTERFACE THEORIES FOR CONVERTER SYNTHESIS

Promising formal techniques for the compositional analysis of component-based design have been developed [23, 24, 54].

111
The key of these works is the modular-based reasoning that provides a support for the modular checking of behavioral properties. They use an automata-based approach to capture both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The formalism supports automatic compatibility checks between interface models, and thus constitutes a type system for components interaction. The purpose of these works is quite different from our. They check that two components have compatible interfaces if a legal environment letting them correctly interact there exists. Each legal environment is an adaptor for the two components. They provide only a consistency check among components interfaces. That is, they do not deal with automatic synthesis of components interfaces adaptors (i.e., automatic synthesis of legal environments). However in [54], they use a game theoretic approach for checking whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. This approach is able to automatically synthesize the converter.

Unlike the work presented in this thesis, a 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 that. Moreover, they are only able to restrict the system’s behavior to a subset of desired behaviors and they are not able to augment the system’s behavior to introduce more sophisticated interactions among components.

7.3 COMPOSITIONAL MODEL-CHECKING OF SOFTWARE COMPONENTS

Other works that are related to our approach appear in the model checking of software components context in which CRA (Compositional Reachability Analysis) techniques are largely used [30, 31].

Also these works can be seen as an instance of the general problem formulated in this thesis. In the case of these approaches the treated problem can be formulated as follows: given a component $C$ and a desired behavior $B$, find an environment $E$ for $C$ in such a way that $E(C) \equiv B$ under an appropriate notion of equivalence\(^1\). They provide an optimistic approach to software components model checking. In this approach when a component is model-checked against a property, the algorithm returns one of the following three results: i) the component satisfies the property for any environment; ii) the component violates the property for any environment; or finally iii) an automatically generated set of assumptions that characterizes exactly those environments in which the component satisfies the property. This approach suffers the state-space explosion problem. However this problem is raised only in the worst case.

The difference with our approach is that they automatically synthesize the assumptions that represent the weakest environment in which the component satisfies the specified properties. That is, they deal with only two components: i) one actual component and ii) its environment. Moreover, by focusing on technological aspects of this approach, unlike our work, they do not treat the problem of automatically synthesizing the actual code that provides a component with the environment required to satisfy a given property by letting the component completely unchanged. That is, although automatically synthesizing the actual code implementing the “correct” environment might be possible, this approach is not currently able to deal with a black-box components setting.

7.4 PROTOCOL ADAPTOR SYNTHESIS

Our research is also related to work in the area of protocol adaptor synthesis [75].

\(^1\) $E(C)$ is the set of behaviors of the system formed by the component $C$ and the environment $E$ where $C$ and $E$ interact with each other one of them.
The main idea there 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 extended reliability behavior that our synthesis approach supports. Moreover, they require a formal specification of the adaptor dictating, for example, a mapping function among events of different components. Although this kind of specification represents a useful mean to formally and systematically characterize an adaptor, it is not user-friendly as much as possible and hence its application to industrial settings is quite limited. Moreover even though using other kinds of techniques to specify the adaptor might be possible, providing this adaptor specification requires to know so much implementation details by missing part of the goals of the work presented in this thesis.

### 7.5 Wrapper formalization

In other work in the area of wrappers formalization and specification, it is showed how to use formalized protocol transformations to augment connector behavior [61, 62].

The key result was the formalization of a useful set of connector protocol enhancements. Each enhancement is obtained by composing wrappers. This approach characterizes wrappers as modular protocol transformations. The basic idea is to use wrappers to enhance the current connector communication protocol by introducing more sophisticated interactions among components. Informally, a wrapper is new code that is interposed between component interfaces and communication mechanisms. The goal is to alter the behavior of a component with respect to the other components in the system, without actually modifying the component or the infrastructure itself.

While this approach deals with the problem of enhancing component interactions, unlike this work it does not provide automatic support for wrapper synthesis, or for automatically eliminating interaction deadlocks.

### 7.6 Behavioral types and component adaptation

In other work in the area of component adaptation, it is showed how to automatically generate a concrete adaptor from: (i) a specification of components interfaces, (ii) a partial specification of the components interaction behavior, (iii) a specification of the adaptation in terms of a set of correspondences between actions of different components and (iv) a partial specification of the adaptor [12, 13, 14, 17, 16].

The key result was the setting of a formal foundation for the adaptation of heterogeneous components that may present mismatching interaction behavior.

Although this work provides a fully formal definition of the notion of components adaptor, its application domain is quite different than the our. Since, in specifying a system, we want to maintain a high abstraction level, assuming a specification of the adaptation in terms of a set of correspondences between methods (and their parameters too) of two components requires to know so much implementation details (about the adaptation) that we do not want to consider in order to synthesize the adaptor.
In this thesis we have described a coordinator-based architectural approach to the automatic composition and adaptation of software (black-box) components. Our approach focuses on:

• detecting and recovering a restricted class of mismatches at the level of interface signature (i.e., incompatible or non-existent method names);
• detecting and recovering possible concurrency conflicts (i.e., deadlocks);
• enforcing coordination policies on the interaction behavior of the functional components constituting the system to be assembled;
• enhancing the current communication protocol of the composed system in order to introduce more sophisticated interactions among components.

A key role is played by the software architecture structure since it allows all the interactions among functional components to be explicitly routed through a synthesized coordinator. By imposing this software architecture structure on the composed system we isolate the functional components interaction behavior in a new component (i.e., the synthesized coordinator) which is subsequently inserted into the composed system. By acting on the coordinator we have four effects:

• the components interaction behavior can satisfy the coordination policies specified for the composed system;
• the global system becomes flexible with respect to specified coordination policies;
• a (possibly composite) coordinator represents initial glue code that can be enhanced, in a compositional way, to augment the system behavior by adding more interactions without changing the existing components and their interfaces;
• our approach is compositional in the automatic synthesis of the enhanced coordinator; that is, each enhancement is treated as a modular protocol transformation so that we can apply coordinator protocol enhancements in an incremental way by re-using the code synthesized for already applied enhancements;

Our approach requires us to have a bMSC and HMSC specification of the system to be assembled from which we automatically derive descriptions of the components interaction behavior. Since these specifications are common practice in real-scale contexts, this is an acceptable assumption. Moreover we assumed to have a Büchi Automata specification of the coordination policies that have to be enforced. Finally, we also require a bMSC and HMSC specification of the protocol enhancements that have to be applied.
We implemented the whole approach (which is formalized in Chapter 4) in our SYNTHESIS tool (which is presented in Chapter 5). SYNTHESIS should be considered a prototype and it is available at the following URL: http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.html. So far, we validated and applied SYNTHESIS for assembling only Microsoft COM/DCOM components. The coordinator code synthesized by SYNTHESIS refers to Microsoft Visual Studio with Active Template Library (ATL) as reference development platform.

The architectural approach presented in this thesis and implemented in the SYNTHESIS tool - which is also presented here - has to be considered as a support for developing component-based systems out of a set of already implemented heterogeneous components by ensuring the correct functioning of the assembled system at level of component interaction protocol. Moreover, it also provides the developers with a mean for evolving/maintaining the assembled component-based system by allowing them to add new functionality and, in the same time, continue to maintain (or to be consistent with respect to) the behavior of the original system.

Possible limits of the approach are: i) we completely centralize the coordinator logic and we provide a strategy for deriving the corresponding actual code that produces a centralized implementation of the coordinator component. This implies that we must deploy the coordinator component on the same machine where its encapsulated servers run; ii) we assume that an HMSC and bMSC specification for the system to be assembled is provided. Although this is reasonable, it is interesting to investigate testing and inspection techniques to directly derive from a COTS (black-box) component some kind of (possibly partial) behavioral specification; iii) we assume also a Büchi Automata specification for the coordination policies to be enforced. It might be interesting to find a more user-friendly coordination policy specification; for example by extending the HMSC and bMSC notations to express more complex interaction behaviors.

8.1 Future work

The complexity of the synthesis and analysis algorithm is exponential in both space and time. This value of complexity is obtained by considering the unification process complexity and the size of the data structure used to build the coordinator graph.

Future work related to the synthesis algorithm concerns:

- the study of better data structures for the coordinator model in order to reduce its size;
- by referring to the automata based model checking [21], it might be also useful to investigate on-the-fly analysis performed during the coordinator model building process.

The current version of SYNTHESIS is a prototype produced by following an evolutionary approach to systems development. Thus SYNTHESIS is still subject to further extensions.

As future work related to SYNTHESIS, we plan to:

- develop more user-friendly specification of both the desired behaviors and the protocol enhancements (e.g., UML2 Interaction Overview Diagrams and Sequence Diagrams);
- validate the applicability of the whole approach to large-scale examples;
- combine our approach with the approach of adaptors synthesis described in [14]. Depending on the logic implemented by the wrapper, this might allow us to automatically synthesize the wrapper from an its partial specification rather than built it by scratch or acquire it as a pre-existent (COTS) component;
8.1 Future work

- derive a distributed implementation of the coordinator in order to not require the coordinator and its encapsulated serves to be registered in the same machine;

- derive a distributed model of the coordinator without producing its centralized model. The distributed model might be formed by a set of component filters (each of them local to each component) that, looking at local information, validate global behaviors.

Further future work is related to the application of SYNTHESIS in a context more general than the one considered in this work:

- in order to introduce SYNTHESIS in a systematic and engineering approach to component adaptation beyond automatically generating adaptors in such a way that the adapted system exhibits a specified functional behavior, we should provide the SYNTHESIS’s user with support to predict the impact of the introduced adaptors on the extra-functional behavior of the adapted system. This might be done by adding information to the component specification - given as input to our method - in order to perform an adaptor generation process that simultaneously produces QoS prediction models. This implies to extend also the behavioral models used by SYNTHESIS in order to take into account this additional information, e.g., with timing information, quantitative information about resource usage or, in other words, usage profiles. The main issue, here, is to define a formalism which is expressive enough to be efficiently used but - in the same time - good enough to be efficiently analyzed in order to detect/recovery mismatches and simultaneously estimate the impact of the recovery strategy on the extra-functional behavior of the composed system;

- the generated prediction models might be mechanically elaborated to assist the SYNTHESIS’s user in selecting the best (with respect to QoS) adaptor among all possible. How to provide automation for the selection process is not still clear and, hence, the adaptor selection would constitute a relevant open issue. ”Knowledge acquisition” approaches might be investigated in order to try to face this problem;

- the kind of adaptor generated by SYNTHESIS matches with the well known “Adapter” pattern [40]. In order to be able to bridge an increased variety of component mismatches (e.g., QoS mismatches), SYNTHESIS should be extended to generate adaptors whose structure and logic comply to many kinds of adaptor such as, e.g., the “Cache” pattern or others discussed in [40];

- if many kinds of adaptor might be generated then another relevant open issue is concerned to the selection of the right prediction method, e.g., whether to use simulative, analytic or experience-based methods. Especially, it is unclear if the best method depends on the actual pattern being used by the generation process;

- AOP (Aspect-oriented programming) techniques/approaches are becoming more and more important in the context of component adaptation since their advantages with respect to traditional approaches. Namely, they provide transparent adaptation through obliviousness, represent lightweight methods of implementing adapters, allow non-functional behavior reuse, and are suitable for mobile computing since there exist advanced AOP platforms that can add/remove aspects at run-time. Despite all these advantages, the current AOP approaches still lack of automation for generating aspects. Thus, it might be useful to investigate the integration of our approach with AOP approaches. Our approach would provide one with a support for automation. In contrast, AOP approaches would provide SYNTHESIS with capabilities that are currently missing, e.g., performing adaptation at run-time by adding/removing/updating the generated aspect.
REFERENCES


[3] ITU telecommunication standardisation sector, ITU-T recommendation z.120. message sequence charts. (msc’96).


[34] HORWICH, P. Wittgenstein and kripke on the nature of meaning. Mind and Language 5 (1990), 105–121.


REFERENCES


