VRIJE UNIVERSITEIT BRUSSEL

Faculty of Science – Department of Computer Science

November 2001

PACOSUITE

Component Composition Based on Composition Patterns and Usage Scenarios

Dissertation presented to the Vrije Universiteit Brussel in partial fulfillment of the requirements for the degree of Doctor in Science

Author: Bart Wydaeghe

Promoter: Prof. Dr. Viviane Jonckers

Abstract

Components come in a variety of formats, designs and implementations. Components can be designed to work together or their designs can be totally incompatible. This influences greatly the amount and kind of composition work that is to be done. In this work, we build on the work of architectural description languages to improve current visual component composition environments.

This is done using the concept of composition patterns. A composition pattern describes an interaction between a set of roles using an extended sequence chart. It serves as a bridge between the design and the implementation. We further propose a component documentation using the same kind of extended sequence charts. The concept of composition patterns lifts the abstraction level of current composition techniques to the same level of the components. I.e. composition patterns are first class objects that can be defined, stored and reused independently of the components.

We further develop and implement algorithms to perform automatic compatibility checking based on finite automata theory. We also developed tool support that gives feedback in case of a mismatch. This includes a visualization of the matching process and the generation of adaptors. The latter is a new technique based on the adaptive programming library and the results of Reussner. Finally, we describe how glue code can be generated that constrains incompatible and unwanted behaviors of components based on the constraints specified by the composition pattern. This glue code allows us to use a more flexible compatibility check that leads to more generic and more reusable composition patterns.

We implemented these checks and mismatch feedback in our prototype of a visual component composition environment. This tool shows how the formal checks can be hidden for the user to provide an intuitive and easy to use component composition process. We demonstrate this by showing how a small exam construction kit is provided and used in this prototype.

This work is mainly useful to build very flexible construction kits. It allows the developers of such a kit to provide default composition patterns together with their set of components without touching the ability of the users of these construction kits to build very complex compositions that were not foreseen by the developers.

Acknowledgements

Research is often considered to be a very solitary activity. The typical image of a weirdo scientist doing all kind of obscure things in an isolated lab is still wide spread.

However, time has changed. When I started to know computers (about 20 years ago), most of the successful computer games where made by one programmer. Today most of the successful computer games involve hundreds of man-years. The same revolution took place in computer science research. Current research has just become too complex and involves too much resources and time to be done by one person. Successful research today has become a very cooperative business.

So, how do you say thank you to everybody whom you worked with, lived with and talked with in the last six years? The list of people that contributed to this work one way or another is endless. Therefore, I like to thank here all the people I forgot to mention below.

I was guided to a research position while doing my master thesis on parallel algorithms for singular value decomposition. My promoter for this work was Prof. Dr. Pieter De Groen. During one of the meetings for this thesis a PhD student dropped in with some kind of numerical analysis question for the image reconstruction of 3D PET scans. When I showed my interest for that kind of research, he brought me in contact with Prof. Dr. Michel De Vriese. The result was that I worked for half a year on image reconstruction for 3D PET scans on his lab. Without their support, I would never even have started doing research at all.

After that time, I was invited by my current promoter Prof. Dr. Viviane Jonckers and my current colleague and friend Dr. Kurt Verschaeve to discuss a research opportunity at the software engineering lab. This time I did get a grant of the Belgian government to support my research so I ended up for six year at the System and Software Engineering Lab. Needless to say that I owe these two people a big “thank you”. My promoter must have spent countless hours to correct all the papers and texts I have written during my research. She is (often to my despair) awfully good in pinpointing both reasoning errors as well as spelling errors in my work. Kurt Verschaeve was already a friend before we became colleagues and working together only deepened this friendship. He has been my main sounding board and many of the ideas in this work would not be what they are today without him.

Of course, I thank all other colleagues at the SSEL lab and the PROG lab for their support and the fine environment they provided. I like to thank especially Luc Goossens for his friendship and his incremental algorithm idea. I also want to thank Bart Michiels who contributed to the code generation research mentioned in this work. Finally, I want to express my gratitude to our new colleague Wim Vanderperren. He did his master thesis on this subject and joined our lab about nine months before this work was finished. He is responsible for a lot of the implementation effort and accelerated this research considerably with his insights. He is currently working on his own PhD and I am sure that he will do a *very* good job at it.

I further want to thank Prof. Dr. Dirk Vermeir for his interesting comments on this work and Prof. Dr. Karl Lieberherr for his introduction to adaptive programming and the very pleasant stay at his lab in Boston. I also owe my gratitude to Ralf Reussner, who gave me interesting feedback regarding adapter generation.

Doing my studies and working on my PhD has been the best period of my life until now. This is due to the great group of friends I encountered during this time. I thank everybody in the 60s for their friendship. I also like to thank Rene Vergaerde for introducing me in the world of medical websites and the pleasant times we spent together.

My parents made all this possible in the first place. They always supported me in what I was doing without ever pushing me into it. They taught me to communicate with an open mind. They never ceased to show me the other side of the story when I once again drew my conclusions to fast. Thank you for everything.

I finish these acknowledgments with a special thank to my wife Marleen Vandepitte and my little son Robin. When living together one often forgets to say explicitly how much the others mean to you. Therefore Marleen: thanks for your support, your love and for just always being there for me. And for my little son Robin: while you do not understand all these complex sentences, I am sure that you do appreciate this big bunch of pages to cut into pieces or to convert into beautiful drawings.

This research was conducted under grant number IWT 951222 and with the support of the Flemish Government Project: Advanced Internet Access (ITA II)

Table of Contents

1 INTRODUCTION **11**

1.1 PROBLEM STATEMENT 12
1.2 APPROACH 14
1.3 A MOTIVATING EXAMPLE 17
1.4 CONTRIBUTIONS 20
1.5 OVERVIEW OF THE DISSERTATION 21

2 CONTEXT **22**

2.1 STATE OF THE ART 23
2.1.1 Visual Programming Environments 23
2.1.2 Documentation 24
2.1.3 Compatibility 27
2.1.4 Design Patterns 29
2.1.5 Architectures 30
2.1.6 Formal Specifications 31
2.1.7 Higher Level API's 31
2.1.8 Separation of Concerns 32
2.1.9 Conclusion 33
2.2 COMPONENT MODEL 34
2.2.1 Introduction 34
2.2.2 Our Opinion 35
2.3 WHY COMPONENTS? 37
2.4 WHY COMPOSITION PATTERNS? 39

3 DOCUMENTATION **41**

3.1 INTRODUCTION 42
3.1.1 Observation 1: Usage Differs from Functionality 42
3.1.2 Observation 2: Components Have More Than One Use 43
3.1.3 Observation 3: Not Everything Is In A Name 44
3.1.4 Observation 4: Composition Patterns Need to be First Class 44
3.2 REQUIREMENTS 45
3.3 EXISTING DOCUMENTATION TECHNIQUES 46
3.3.1 Communicating Sequential Processes (CSP) 46
3.3.2 Petri Nets 47
3.3.3 State Diagrams 47
3.3.4 Sequence diagrams 48
3.3.5 Collaboration diagrams 49
3.3.6 Summary 50
3.4 OUR DOCUMENTATION 51
3.4.1 Introduction 51

3.4.2 Scenarios 51
3.4.3 Participants 52
3.4.4 Messages 53
3.4.5 Control Blocks 57
3.4.6 Mapping 58
3.4.7 Documenting Components and Composition Patterns 59

4 MATCHING 61

4.1 INTRODUCTION 62

4.2 MAPPING MSC'S ON AUTOMATA 64

4.3 DEFINITIONS: COMPONENT AUTOMATA AND COMPOSITION AUTOMATA. 66

4.4 OPERATIONS ON COMPONENT AND COMPOSITION AUTOMATA 68
4.4.1 Projection of a composition pattern automaton 68
4.4.2 Applying the role map function 68
4.4.3 Parallel Composition 69
4.4.4 Laws for the Parallel Composition Operator. 71

4.5 COMPATIBILITY 72
4.5.1 Local Compatibility 73
4.5.2 Global Compatibility 75

4.6 LOCAL CHECK 79
4.6.1 Calculating the Intersection 79
4.6.2 Check for a Start-Stop Path in the Intersection 79

4.7 GLOBAL CHECK 81
4.7.1 Classic 81
4.7.2 Optimization using Asymmetric Cross Products 84

4.8 ROLE/COMPONENT MAPPING 86
4.8.1 Overview of the process 87
4.8.2 Calculating the intersection without role/component mapping 87
4.8.3 Select All Traces That Have Non-Contradicting Comp/Role Mappings 88

4.9 ROLE/ENV MAPPING 93
4.9.1 Mapping Rules for the "ENV" Participant 94
4.9.2 Checking the "ENV" participant mappings 95

4.10 MESSAGE HIERARCHY 97

4.11 NON-DETERMINISM 98
4.11.1 When Does Non-Determinism Occur? 98
4.11.2 How to Treat Implementation Non-Determinism? 99
4.11.3 How to Treat Scenario Non-Determinism? 101

5 MISMATCH FEEDBACK 102

5.1 INTRODUCTION 103
5.1.1 Problem Statement 103
5.1.2 Approach 104

5.2 ANNOTATING COMPATIBILITY ON MSC'S 106
5.2.1 From Automata to MSC's 106

5.2.2 Discussion 107

5.3 ADAPTER GENERATION 109
5.3.1 Reussner Adapter Generation 109
5.3.2 The Adaptive Programming Library 111
5.3.3 Link Adapter Generation and Adaptive Programming 119
5.3.4 Calculating Adapters 119
5.3.5 Conclusion 126

6 CODE GENERATION 128

6.1 INTRODUCTION 129

6.2 PREPROCESSING THE GLUE CODE AUTOMATON. 131
6.2.1 Remove Non-Valid Traces 131
6.2.2 Collapsing Out/In pairs. 131
6.2.3 Parameter Mappings 132

6.3 GENERATING JAVA EVENT HANDLING CODE 134

6.4 GENERATING CODE 140

7 THE EXAM CONSTRUCTION KIT 142

7.1 INTRODUCTION 143

7.2 DOCUMENTATION 144
7.2.1 Components 144
7.2.2 Composition Patterns 149

7.3 THE COMPOSITION PROCESS 151

7.4 CONCLUSIONS 155

8 CONCLUSIONS 156

8.1 CONTRIBUTIONS 157

8.2 APPLICABILITY 159
8.2.1 Influence on Visual Component Composition 159
8.2.2 Scalability 159
8.2.3 Influence on Other Research 160

8.3 FUTURE WORK 162

9 PRODUCTS AND SPECIFICATIONS 164

10 REFERENCES 165

List of Figures

Figure 1: PacoDoc Screenshot 14
Figure 2:PacoWire prototype visual composition tool...16
Figure 3: Toggling Composition Pattern...17
Figure 4: Typical use of a Juggler Component 17
Figure 5: Typical use of a standard JButton Component...17
Figure 6: Composition Tool 18
Figure 7: Levels of documentation...25
Figure 8: Summary of the scenario syntax 52
Figure 9: Research by Shank for a minimal set of primitives to bring natural language to a canonical form 53
Figure 10: Set of Primitives...55
Figure 11: Why a hierarchy in primitives is needed...57
Figure 12: Documentation of the driving exam client interface 59
Figure 13: Network Interaction Composition Pattern 60
Figure 14: Converting sequence charts to automata 64
Figure 15: Adding "relative" direction for component usage scenarios. 65
Figure 16: Splitting messages in the composition pattern into a sending and a receiving part..65
Figure 17: Generic observer behavior supporting both polling and notification style 73
Figure 18: Snapshot of the component composition process 74
Figure 19: Why a global check is needed. 76
Figure 20: Adding termination properties...80
Figure 21: Calculating the shuffle automaton...82
Figure 22: Template for a single message in the intersection automaton going from component 1 to component 2..83
Figure 23: The asymmetric cross product. 85
Figure 24: Searching contradicting traces...88
Figure 25: Example state diagram for role/component resolving 90
Figure 26: Finding all role/component mappings using dynamic programming ideas 92
Figure 27: Mapping multiple “env” participants on one role.......................................93
Figure 28: Mapping one "env" participant on multiple roles..94
Figure 29: Illegal mapping of "env" participants...94
Figure 30: Mapping one "env" participant on multiple roles..95
Figure 31: Combining two equal messages with hierarchy...97
Figure 32: Implementation non-determinism 98
Figure 33:Example of scenario non-determinism 99
Figure 34: In black box components, ALT blocks represent non-deterministic behavior 100
Figure 35:Indicating the mismatch?.. 103
Figure 36: Feedback process by anotating MSC's... 107
Figure 37: Marking incompatible traces?.. 108
Figure 38: Adapting a simple CD player.. 109
Figure 39: The generated adapter. Transitions that are not marked with (M) are needed adaptations.. 110
Figure 40: Bus simulation class graph. Squares and hexagons denote classes (concrete and abstract, respectively), regular arrows denote fields and

are labeled by the field name, and bold arrows (labeled with ◊) denote the subclass relation (for the shading, see text) 112
Figure 41: Evolved bus simulation class graph... 112
Figure 42: Strategy graph corresponding with the "From Busroute through BusStop to Person" strategy. 113
Figure 43: Example class graph and strategy graph used for the calculation of a traversal graph. In this example e_1 and e_2 have no attached constraints, the constraint attached to e_3 specifies that the edge going from A to D should be bypassed and the constraint attached to e_4 specifies that A and all incident edges to A should by bypassed. 113
Figure 44: Applying the traversal graph algorithm on the example in the previous picture 115
Figure 45: Class Graph and Strategy Graph with Corresponding Automaton 116
Figure 46: Applying the traversal graph algorithm to the component at the left hand side and role R1 of the composition pattern at the right hand side, results in a traversal graph specifying that the component fits with the role if message C is inserted between B and D 120
Figure 47: Example for Adapter Generation.. 121
Figure 48: Converting the Component and Composition Pattern specified in Figure 47 to a Class Graph and a Strategy Graph................................. 122
Figure 49: Marking the Traversal Graph for the example depicted Figure 47 123
Figure 50: Resulting Traversal Graph for the example in Figure 47........................ 123
Figure 51: The usage scenario of the component at the left hand side is not compatible with role R1 of the composition pattern at the right hand side. 124
Figure 52: How to Adapt? Inserting the first B or the second?................................. 125
Figure 53: Using the Name Map function to select the adapter. 126
Figure 54:Template for messages with their implementation mapping in the resulting global check automaton.... 131
Figure 55: "Collapsing" Out/In pairs. 132
Figure 56: Parameter mapping dialog 133
Figure 57: DrivingExamGUI usage scenario.. 145
Figure 58: Typical use of a standard JButton Component 147
Figure 59: Usage scenario for the network client component................................... 147
Figure 60: Documentation of the driving exam server component............................ 148
Figure 61: Description of a generic network server component 149
Figure 62: LaunchClientNetwork Composition Pattern describing the interactions between a launcher, a client and a network role 150
Figure 63: Two basic composition patterns... 150
Figure 64: Snapshot of the development of an exam application using the exam construction toolkit.. 151
Figure 65: Compilation of screenshots of the exam service application................... 152
Figure 66: Usage scenario for a Console component.. 152
Figure 67: UseNetwork composition pattern... 153
Figure 68: Extending the exam application with chat functionality. 154
Figure 69: Generic usage scenario and composition pattern.................................... 157
Figure 70:Composition Adapter: CONTEXT specifies where to apply and ADAPTER specifies what to do. 162

Figure 71: Applying the composition pattern adapter to the original composition pattern on the left hand side results in the composition pattern at the right hand side. 163

1 Introduction

"I don't have a solution, but I certainly admire the problem"

- Ashleigh Brilliant

"For every problem, there is one solution which is simple, neat and wrong."

- Henry Louis Mencken (1880-1956)

1.1 Problem Statement

Component technology becomes increasingly important in the software industry. A large community believes that components and associated component models help in reducing the cost and the development time of new applications. The following quote is prototypical for this way of thinking:

"Component Based Development is an approach to application development in which prefabricated, pre-tested and reusable pieces of software are assembled together thereby enabling very flexible applications to be built rapidly. It all sounds engagingly simple, to the extent that often the child's toy Lego is used as a metaphor, appropriate right down to the need for standardization - visualize those neat little plugs and sockets each brick has, irrespective of size and shape. If an application designer can draw upon a pool of software objects, each plug-compatible with another, and whose form and function are as obvious as Lego bricks, then the vision has been realized. A change in the requirements? No problem, just snap out one brick or sub-assembly, and snap-in another"

[Short, 1997]

Components come in a variety of formats, designs and implementations. Components can be designed to work together or they can be obtained from very different sources. Bringing these building blocks together results in an application. Needless to say, that just bringing components together is not enough to obtain a working application. In the real world, application programmers write a lot of glue code to make components work together. Today the cooperation between components is mainly defined at the implementation level. At that level there exists standards so that programs written in different programming languages and on different platforms can understand and use each other (e.g.[COM, 1999;CORBA, 2000;DCOM, 1998]). These technical possibilities provide component connectivity. Connectivity on its own does not imply interoperability. The following analogy of a "plain old telephone service" shows clearly the difference.

"Thanks to a worldwide numbering scheme and the interconnection of all the national telephone networks, it is fairly easy to establish a telephone connection between two arbitrary points in the world to carry sounds between these two points. This is worldwide connectivity. If, however, people want to interoperate (do meaningful things) through the telephone, they should not only be able to exchange sounds, but also to understands each others sounds, i.e. to use a common language."

[Tiberghien]

We believe that just as in the telephone example current approaches to component technology and component composition have a serious problem in assuring component interoperability i.e. "the components plug, but they don't play". We need documentation on how to implement a typical cooperation between components i.e. what interaction scheme is needed between a given set of components to obtain a wanted application with a given design. Today this is solved using experienced developers.

Current component documentation techniques enable humans to determine functionality. However, the documentation provided to *use* a component is very weak in most cases. It is common practice that developers start experimenting with a component to find out how it should be used. This means that composing components requires a lot of technical knowledge and a lot of effort. This stands in sharp contrast with the selection of suitable components for your application. It is usually not necessary to know all the technical details to identify a component as a possible candidate for your application. Just browsing a catalogue with natural text descriptions will do in most cases. The knowledge needed to select suitable components is knowledge about the application domain. We state that the main reason of this situation lies in the lack of abstraction at the composition level. Even the latest tools to compose components, force you to really code (be it manually or visually) every connection between a set of components. Composition code is spread around in your application and cannot be reused or considered to be black box.

This work tries to lift the abstraction level of the compositions to the same level as the components. I.e. we want it to be as easy to select and apply a suitable component composition, as it is to select and use a suitable component.

1.2 Approach

Our approach is based on four concepts: composition patterns, component usage scenarios, compatibility checking algorithms and glue code generation. A composition pattern formally specifies how a set of roles interacts, while a component usage scenario specifies how a component interacts with a set of environments. We use a special kind of Message Sequence Charts (MSC's) [MSC, 1993] [Rudolph, 1996]to do this. Each component is documented with a set of MSC's. Each MSC describes a scenario for one of the functionalities supported by this component. The main difference with standard MSC's lies in the kind of signals sent. We developed a compact set of primitives with a predefined meaning. Instead of using API calls we use these primitives to model the components behavior thus avoiding the confusion that stems from the use of API calls for the signal labels. The syntax is mainly the MSC syntax. It contains a set of participants, a set of signal sends between these participants and a set of control blocks and structuring mechanisms.

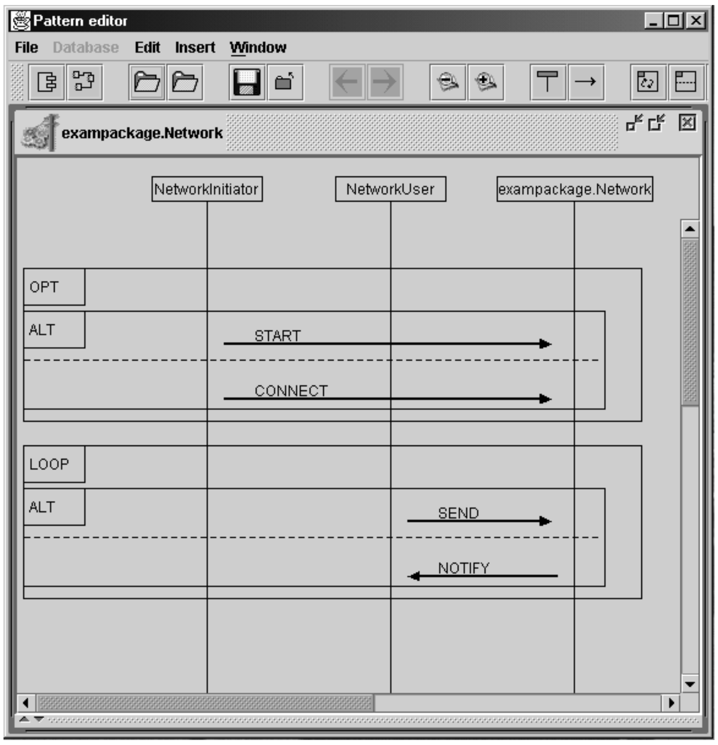


Figure 1: PacoDoc Screenshot

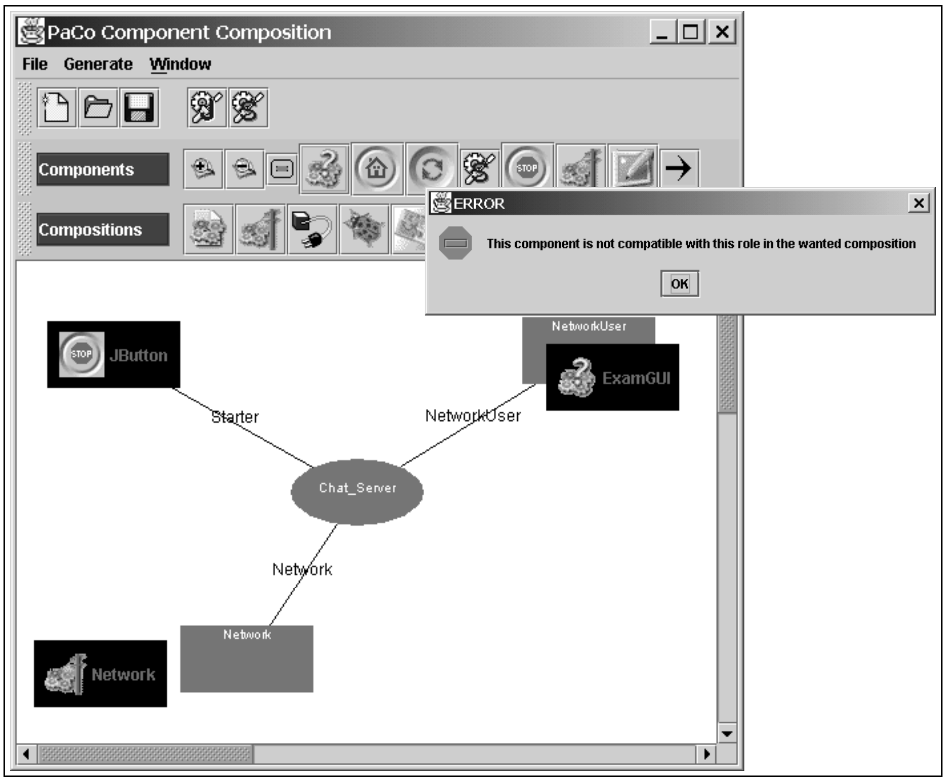
The idea is to document how components should be used. Composition patterns are documented in a similar way using the same compact set of primitives. We developed a prototype editor called PacoDoc (Figure 1) to browse and edit this kind of documentation.

As the composition pattern and the component documentation are expressed in the same formalism it becomes possible to check compatibility between the protocol offered by the components and the protocol as specified by the composition pattern.

We developed algorithms to check compatibility based on finite automata theory. Our compatibility definition allows components to offer more functionality than what the composition pattern asks for, as is the case in most compatibility definitions found today in literature. However, we view the composition patterns as first class reusable entities, implying that also composition patterns can be more general than what the component offers. A good example is an observer composition pattern specifying both polling behavior as well as notification behavior. This composition pattern thus describes in general that two roles should be connected using some kind of observing scheme. We do not want to force any component using this composition pattern to implement both possibilities. Restricting the composition pattern to only one option, on the other hand, would render a less generic and less reusable composition pattern. In short, we declare a set of components and a composition pattern to be compatible if there exists at least one common trace (that reaches an end state) over all components that is also specified by the composition pattern. Technically, we calculate the intersection between the protocols as specified by the parallel composition of all components with the protocol specified by the composition pattern. The result is a new state machine describing the "compatible" behavior between this set of components, constrained by the protocol specified by the composition pattern. If this automaton is not empty, we generate the source code that implements this result automaton. This code is then used as the glue code between the components. This is necessary as our compatibility definition only assures that there is a common trace. It does not guarantee that it is the only trace, or that the components follow this trace at runtime. The generated glue code ignores all non-valid behavior of the components and allows only the common traces between the components and the composition pattern. The details of this process are further explained in chapters 4 and 6.

If our compatibility checks returns a mismatch we offer the developer several possibilities to analyze the incompatibility. This includes tools that mark the compatible behavior and indicate were the compatibility check fails, as well as a tool that suggest "fixing" scenarios for both components and composition patterns.

All these technical matters are transparent for the user. During component composition, a user works with a visual composition editor that contains a palette of components and a palette of composition patterns. We developed a prototype of such a tool called PacoWire. This tool allows dragging a component on a role of a composition pattern. The drag is refused when the component does not match with the selected role. It is possible to drag a component on more than one role, so that the same component can be shared among different composition patterns. When all the roles of a composition pattern are filled, this tool checks the components against the composition pattern and vice versa. If the check succeeds, glue-code is generated.



This component is not compatible with this role in the wanted composition

Figure 2:PacoWire prototype visual composition tool

This implies that we have a new way to start the development of component-based applications. In current visual composition tools, the developer selects a set of components first and tries to implement this design afterwards using these components. There is no feedback whether these components fit in the design or not. Composition patterns can be selected based on a design document (typically use cases) and search for compatible components based on these composition patterns.

1.3 A Motivating Example

As a kind of "quick preview", we describe here a small example that indicates the problem and shows how we try to solve it. Our work is mainly targeted towards Java Beans. One of the tutorials for the Java Bean Box describes how to build a Juggler application so that a click on one button starts the juggling on the Juggler component and a click on a second button stops the juggling on the Juggler component. The construction of this application only requires visual wiring. We show how we use our approach to build a similar application but instead of one start button and one stop button, we try to use the same button component as a "toggle" button. This is not possible in the Bean Box (nor in any other commercial visual wiring tool I know) without manual adaptation of the code, because current tools only generate code for fixed event/action pairs. I.e. one event always results in the same API call.

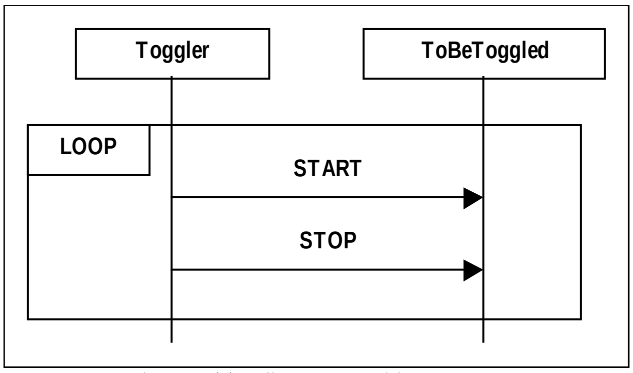


Figure 3: Toggling Composition Pattern.

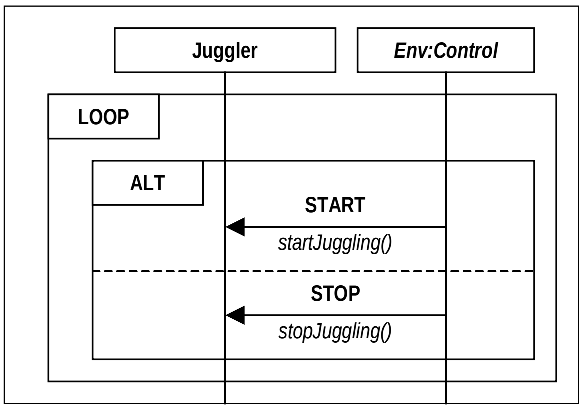


Figure 4: Typical use of a Juggler Component

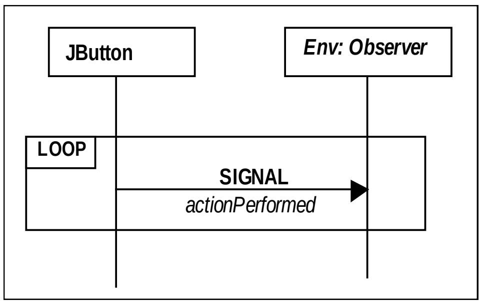


Figure 5: Typical use of a standard JButton Component

In our approach, we add a pallet of standard composition patterns on top of the pallet of components found in current tools. The documentation for a typical usage scenario for a standard Java Button and for the Juggler component is shown in Figure 4 and Figure 5. The documentation for a toggle composition pattern is depicted in Figure 3. This documentation is made by the developer of the construction kit and is transparent for the user. The user uses a tool as shown in Figure 6. He or she selects the toggle composition pattern from the palette with composition patterns and he or she fills the Toggle role with the Button component and the ToBeToggled role with the Juggler component. Our tool then checks compatibility of the components with the composition pattern. If the check succeeds it generates all code necessary to build the application. In case the check fails gives mismatch feedback. In this small example, the documentation of the JButton component is checked against the Toggler role. This is done based on the primitives (START, STOP, SIGNAL…) and the sequence diagrams. We define a basic hierarchy on the primitives where SIGNAL matches with any other primitive. It is easy to see that the JButton component matches with the Toggler role and that the Juggler component matches with the ToBeToggled role (for an exact definition of our compatibility definition see section 4.5)

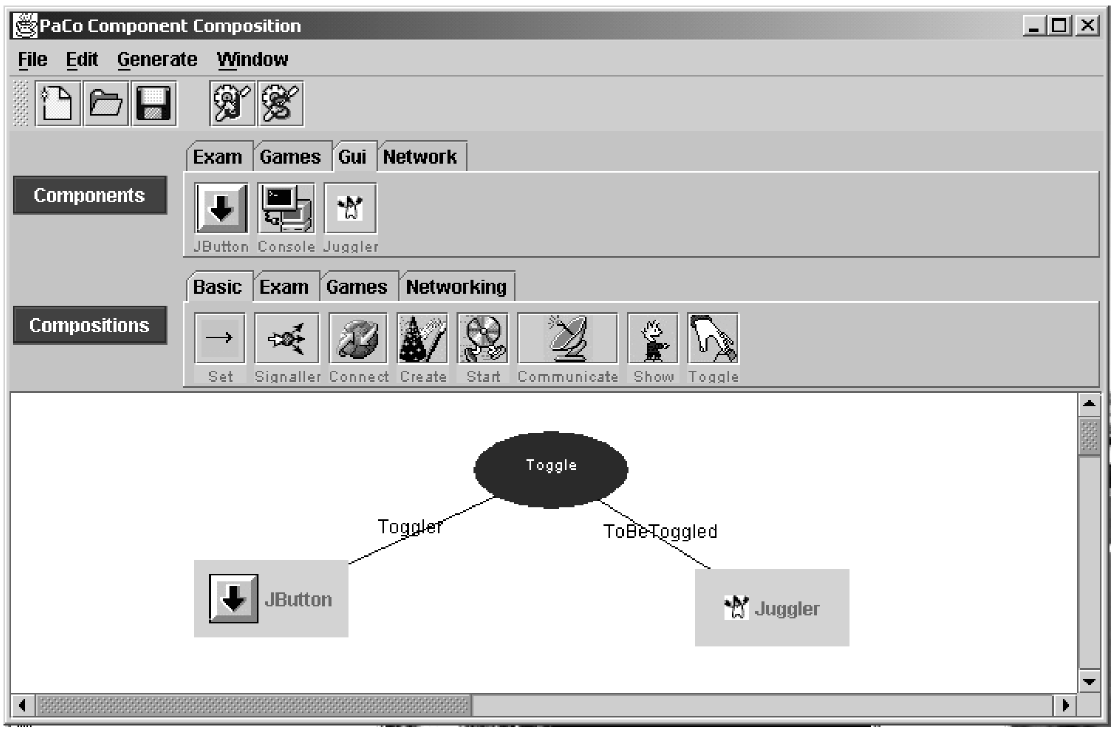


Figure 6: Composition Tool

In this case, the tool performs the compatibility check and generates a main application that instantiates a button and a juggler. It also generates a small state machine component listening for JButton "actionPerformed" events and calling the "startJuggling()" and "stopJuggling()" methods on the Juggler component alternately. This means that all the necessary glue code is generated and that the tool does not require any programming knowledge from the user.

It is clear that this approach is not limited to binary composition patterns only, nor is it limited to a single composition pattern. However, this very simple example already indicates the main principles and benefits of the approach. This research improves the code generation process found in current commercial visual component composition tools by generating full protocols between components instead of mere event/action pairs. The main advantage of this approach lies in the reuse of the composition patterns, the shift of expert knowledge from the application developer towards the component and construction kit builder and in the introduction of compatibility checking algorithms.

1.4 Contributions

The main contributions of this dissertation are:

1. Improvement of current visual component composition environments. This is done using the concept of composition patterns. This concept lifts the abstraction level of current composition techniques to the same level as that of the components. I.e. composition patterns are first class objects that can be defined, stored and reused.
2. Automatic compatibility checking based on finite automata theory using a compatibility definition that allows components to offer more than what the composition pattern asks for and allows composition patterns to be more general than what the components offer.
3. Glue code generation that both forces components to follow only compatible traces and restricts the composition pattern to what the components can offer. This allows us to use more generic and more reusable composition patterns than what is currently available.
4. Improved feedback at composition time allowing the composer to find out mismatches and suggesting fixing scenarios.
5. Support for "composition based" construction of component based applications because components can now be selected on their compatibility with a composition pattern.

This work is mainly useful to build very flexible construction kits. It allows the developers of such a kit to provide standard composition patterns together with their set of components without touching the ability of the users of these construction kits to build very complex compositions that were not foreseen by the developers.

1.5 Overview of the Dissertation

The next chapter (Chapter 2) defines the context of this work. It describes the state of the art of current component composition techniques and deals with the important issue of the component model. It further elaborates on our motivation and view on components. The questions: "Why do I propose to do component based development?", "What is a component in this dissertation?" are answered in this chapter. Finally, it gives the background behind the concept of composition patterns.

Chapters 3 till 6 discuss the technical details of our approach. The documentation to define usage protocols for components and interaction protocols for composition patterns is introduced in chapter 3. To do this we first make some observations about the nature of this kind of documentation. This leads us to a set of requirements that we want to be satisfied by this documentation. This is followed by an overview of how existing documentation techniques perform in view of these requirements. Finally, we present the details of the documentation itself.

The next chapter (Chapter 4) describes the matching process. It starts with our view on compatibility. It mainly describes two different situations. A local matching process that checks a component against a role in a composition pattern and a global matching process that checks if there exists cooperation between a set of components and a specified composition pattern. It further describes algorithms to perform automatic matching of components and roles. I.e. during the normal component composition process the developer drags the right component on the right role of the composition pattern. We devised an algorithm that searches the most likely role for a given component based on compatibility. Finally, we go into more detail on how we handle environments and sub-typing.

Chapter 5 describes techniques and tools to provide feedback in case of incompatibility. This varies from tools that annotate partial compatibility to tools that hint at possible fixing scenarios. The technical part stops with a discussion on the code generation process (Chapter 6) itself. This is followed by a small example where a driving exam construction kit is built (Chapter 7) to illustrate our approach.

At the end of this dissertation, we state our conclusions.

2 Context

"When you steal from one author, it's plagiarism; if you steal from many, it's research."

- Wilson Minzer (1876-1933)

2.1 State of the Art

Component composition is a long-standing issue in software engineering. Many, quite different sub-fields of software engineering research are applicable to this problem. For this work, we use the results and the ideas from very different areas such as visual programming, documentation techniques, patterns, API definitions and formal compatibility checks. We also use ideas and terms coming from the separation of concerns research. In this chapter, we try to give an overview of the relevant state of the art in these domains.

2.1.1 Visual Programming Environments

Current visual programming environments offer a variety of component composition possibilities. Today we see three main classes of visual component composition schemes:

1. The Intelligent Network Approach (LabVIEW [Wells, 1997], Java Studio [Weaver, 1998])
2. Overwriting Event Handlers (Visual Basic [VisualBasic, 2001], Delphi [Delphi, 2001], Visual Java [VisualJava, 2001],….)
3. Visual Wiring (NetBeans [NetBeans, 2001], Visual Age [VisualAge, 2001], BeanBox [BeanBox, 2001],….)

Here the intelligent network approach is the eldest composition scheme and visual wiring constitutes the most recent addition. However, this does not mean that the older approaches are outdated. In fact, all these approaches are heavily used today and have all found their own share of the market.

2.1.1.1 Intelligent Networks

This approach allows to visually script components together. Every component has a very strict interface. A component can be started and throws a (set of) event(s) when finished. Output events of one component can be connected with the start API call of another component. This renders a system that is a close visual representation of a normal programming language. Every wire can be viewed as one instruction. All this means that the wiring is a local process. It is not possible to define global behavior over a set of components. The wiring is certainly not a first class object. It needs to be done over and over again. There is no possibility to store typical wiring except in combination with a set of components.

2.1.1.2 Overwriting Event Handlers

This is the approach found in the highly popular visual development environments such as Visual Basic, Visual C++, Visual Delphi, etc. These environments allow drag and drop facilities of components on a form and let the interaction between these components be defined by "overriding" the event handlers with free code. I.e. for every component, the programmer has the possibility to select an event and to provide code for it. A button component for example allows the "overriding" of the "onClick" event or the "onMouseOver" event. Typically, code that is written for such events involves calls to other components methods (API). This means that the programmer has to know which methods to call, what these methods do and in what order these methods should be called to accomplish the task at hand. The glue code also tends to be scattered around and is therefore difficult to maintain.

2.1.1.3 Visual Wiring

With the advent of Java Beans and more precisely with the Java Bean Box [BeanBox, 2001], a higher abstraction level was introduced. This environment allows the same drag and drop facilities as the classic environments but is extended with a code generation wizard for the basic interactions. These tools allow to drag and drop components on a form and to connect them. Connecting two components pops up a dialog box where an event of the source component can be selected and connected with a method call of the target component. The corresponding code is then automatically generated by the environment. This is the first step towards a higher abstraction level for the glue code. However, even very little experiments show that the user still needs to know how the component API needs to be used to obtain the wanted behavior. It is also very hard to introduce global synchronization.

2.1.1.4 Conclusion

Visual component composition tools improved a lot in the last decade. The abstraction level raised and they became very user friendly. However, composition information is still spread around in the resulting applications and cannot be reused, nor saved independently. This forces the developer to rewrite the glue code over and over again and to know in detail how the components should be combined in new application

2.1.2 Documentation

In [Beugnard, 1999] a good overview of state of the art component documentation is presented. In this paper, the authors introduce four different abstraction layers to describe contracts for components. These four layers are shown in Figure 7. The first level, basic, or syntactic, contracts, is required simply to make the system work. It typically contains the API of the components, possibly with the definition of the data passed between the components. The second level, behavioral contracts, deals with the effect of one call on the component. It typically describes in what context a given call produces valid results. The third level, synchronization contracts, improves confidence in distributed or concurrency contexts. This level typically describes protocols between components and their environments and synchronization issues. The fourth level, quality-of-service contracts, quantifies quality of service.

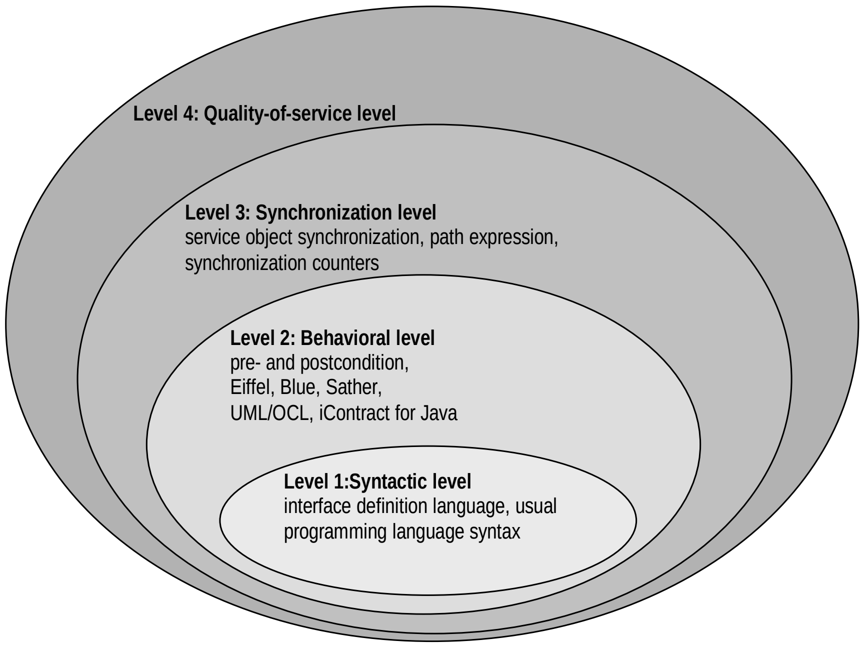

Level 4: Quality-of-service level
Level 3: Synchronization level
service object synchronization, path expression, synchronization counters
Level 2: Behavioral level
pre- and postcondition, Eiffel, Blue, Sather, UML/OCL, iContract for Java
Level 1:Syntactic level
interface definition language, usual programming language syntax

Figure 7: Levels of documentation

Together with the raise of the abstraction level, suitable documentation becomes less and less obvious.

2.1.2.1 The Syntactical Level (Level 1)

At the syntactical level, there are plenty of possibilities. Standard programming languages are typically used to describe the method signatures. More recently we the Interface Description Language (IDL) [IDL, 2001] was introduced to abstract away from specific programming languages. This documentation is also backed by plenty of checking tools (typically performed by the compiler or the interpreter).

2.1.2.2 The Behavioral Level (Level 2)

At this level we find more recent efforts such as design by contract [Meyer, 1992] (Eiffel), contracts by Helm [Helm, 1990], contractual obligation by Lamping[Lamping, 1993], OCL [Warmer, 1999], iContract for Java [Kramer, 1998], etc…. These all try to describe the effect of one operation. I.e. at this level, a procedure or a method call is considered to be a transaction. These contracts are typically added to the programming language (using asserts or pre and post conditions) and are therefore checked automatically. The checking now typically occurs at runtime.

2.1.2.3 The Synchronization Level (Level 3)

At his level, we find more formal approaches that are typically checked in a separate environment. Campbell and Habermann's [Campbell, 1974] introduced the idea of augmenting interface descriptions with sequence constraints already in 1974 (using path expressions). In the following years much work was done to capture this kind of constraints. Most of these approaches build on a formal base such as Communicating Sequential Processes [Hoare, 1985] (see for example the architectural description languages Wright [Allen, 1997] and C2 [Taylor, 1995]) or state machines [Harel, 1987](see for example work of Yellin and Strom [Yellin, 1994a;Yellin, 1994b] and Zaremski [Zaremski, 1997]). This documentation has still not reached the same level of acceptance as the documentation techniques found on level 1 and level 2.

2.1.2.4 The Quality of Service level (level 4)

This level is typically not covered today. If it exists it is typically a natural text description that describes the parameters that influence the quality of service, together with some test results. Recent work at this level of documentation includes the CORBA 2.4 specification by the OMG group (see following quote).

"CORBA 2.4 includes several Quality of Service specifications, which are intended for managing and selecting various underlying transport choices based on application needs. Specifically, this version contains the Asynchronous Messaging, Minimum CORBA, and Real-Time CORBA specifications as well as revisions made by several RTFs and FTFs, including those responsible for the Interoperable Name Service, Components, Notification Service, and Firewall specifications."

Excerpt from the OMG web page at [CORBA, 2000]

2.1.2.5 Conclusions

In our work, we try to express compatibility constraints on the usage protocol of components. This clearly asks for documentation at level 3. Current documentation techniques at this level are not in wide spread use today. We believe that this is mainly caused by the formal notations. Therefore, we try to use a notation that is already accepted to make the acceptance less difficult. The documentation introduced in this work complements other documentation and is only used to perform automatic compatibility checks and glue code generation. It is not suited to find out neither what a component does nor what quality of service it delivers. It only describes how the component should be used.

A more specific and focused overview of possible documentation techniques is given in section 3.3. There we focus specifically on existing documentation techniques to specify interactions between software artifacts.

2.1.3 Compatibility

Research related to compatibility checks and definitions is mainly found in the field of architectural description languages. The architectural mismatch problem was first recognized and described by Garlan and co. in [Garlan, 1995]. This launched a whole new research field around Architecture Description Languages [Allen, 1997;Luckham, 1995;Taylor, 1996] (building on Module Interconnection Languages [DeRemer, 1976;Parnas, 1972]). An architecture in this context is generally considered to consist of components and the connectors (interactions) between them. As the existing architectural descriptions are often informal and ad hoc, this research tries to formalize this documentation. We too try to formalize cooperation between components.

These approaches nearly all define compatibility between a composition specification and a set of component. However, most of this works defines compatibility in a different context than ours.

The RAPIDE system [Luckham, 1995] differs from our approach in their use of unidirectional protocols only. I.e. components are used as a class library where functions are called and output is never actively sent. In RAPIDE, compatibility is deduced from a simulation. I.e. the system is simulated to see whether the components work together as specified.

Yellin and Strom [Yellin, 1994a;Yellin, 1994b] use state machine descriptions to define component compatibility. Their approach is however restricted to two parties. The component composition model used in their approach allows an interface in one component to be bound to an interface in a second component. It does not allow an interface in one component to be bound to multiple interfaces (in several components) or to check a component against a role in a composition specification, as our system does. They define components to be compatible if they are deadlock free and agree on the followed trace. Note that this does not mean that the components may not be able to follow different traces in general. As long as divergent behavior never occurs in the specified composition specification, the components are declared compatible.

The architectural language Wright [Allen, 1997] uses the same concepts as we do. I.e. they have components, composition specifications and roles. Informally they define compatibility as follows: “A component may offer more than what the composition specification asks for but it forces the components to implement at least all the behavior that is specified in the composition specification.” We argue in section 4.5 that our definition is more generic.

Other interesting work regarding compatibility checking can be found in protocol conversion literature [Reussner, 2000] and the interface adaptors of Thatte [Thatte, 1994]. In this work, protocols are used to specify interfaces and an algorithm is described that synthesizes a converter given the protocols and the specification. However, the goal of this work is to generate converters from one protocol to another rather than checking compatibility.

Reussner also uses finite automata theory in his “Coconut” project [Reussner, 1999] to perform component matching. His work is very similar to ours as far as the local check is involved. At the moment, he does not perform a global check. He uses the incremental algorithm to generate adapters for mismatching components. We also share another the asymmetric cross-product algorithm (see section 4.7.2). It is used by Reussner to calculate adapters to combine two non-compatible components. The problem is indeed similar. He needs to follow a common trace as long as possible, once he reaches a point where one component no longer implements the same behavior he keeps the behavior of the first component until he reaches a point where the behavior is compatible again. We do the same where we just keep the traces of the composition pattern for those parts that are not concerned with the component at hand.

2.1.3.1 Conclusion

All work we know adopts a similar definition of compatibility. That is:

- The system is simulated to deduce compatibility or,
- Compatibility is defined for exactly two parties as having common behavior or,

- A component needs to be a refinement of the role it is playing in a composition pattern.

We already argued that it is better to allow both a component to offer more than what the composition pattern asks for and a composition pattern to be more general than what the component offers.

2.1.4 Design Patterns

Around 1995 research on reusable software solutions for recurring problems gained a lot of attention. This research is known in literature as the pattern research [Alpert, 1998;Gamma, 1995;Lajoie, 1994;Riehle, 1997]. Depending on granularity they are respectively called: Idioms, Cliches, Design Patterns and Architectures. Design Patterns and Architectures both launched a new research field. As they both greatly influenced our proposal, we discuss these in further detail.

The design pattern research was launched by the publication of the so-called Gang-of-Four Book [Gamma, 1995] in 1995. The authors were inspired by the book "A timeless way of building" [Christopher Alexander, 1979] the architect Cristopher Alexander tried to write down his knowledge about architectural problems in a fixed format. The GOF book describes twenty recurring design problems with their solution. All twenty problems are written down in the same format (called a Design Pattern Language). This format contains a combination of informal text, UML sketches and implementation snippets. The description is informal but proves to be of great value for developers. A lot of the design patterns are concerned with the cooperation of two or more "participants" in an application. The GOF book [Gamma, 1995] for example contains the observer pattern (cooperation data-view), the visitor pattern (cooperation data-traversal function), the chain of responsibility pattern (typically event cooperation between interface elements), the bridge pattern (kind of indirect cooperation between sender and receiver of a message), the factory pattern (another kind of indirect cooperation between sender and receiver of a message) and many others. Therefore, patterns are good candidates as specification on how components work together in a component based development approach.

The success of these patterns proves that the same kind of compositions is used over and over again. With our usage scenarios and composition patterns, we try to capture this "composition" information. However, as we try to build automatic tool support based on this information, we need to use a more formal notation. Formalizing design patterns tends to diminish the power of it as it usually restricts its generality. It is still an open question in the research community if formalizing is the way to go or not [Coplien, 1996]. We do not have the answer to this question, however we think that formalizing a specific interpretation of a pattern brings many virtues. Components are built with a certain composition scenario in mind. This scenario is not at all general. It involves a very specific order of API calls. We state that formalizing such protocols brings many advantages without the loss of generality that occurs in formalizing design pattern.

2.1.5 Architectures

The architectural mismatch problem was first recognized and described by Garlan and co. in [Garlan, 1995]. This launched a whole new research field around Architecture Description Languages [Allen, 1994b] [Luckham, 1995] [Shaw, 1978] (building on Module Interconnection Languages [DeRemer, 1976]). An architecture in this context is generally considered to consist of components and the connectors (interactions) between them. As the existing architectural descriptions are often informal and ad hoc this research tries to formalize this documentation. These ADL's seek to increase the understandability and reusability of architectural designs, and enable greater degrees of analysis. We too try to formalize cooperation between components. However, most of the architectural description languages today specify component composition in the context of the component model rather than the application context. We start from the assumption that the component model is already chosen and stable. We consider the component model as a necessary precondition before we start specifying our composition patterns. Another known problem with a formal approach like architectural description languages is that they are difficult to learn and to use and suffer from scalability problems. We believe that choosing a set of primitives is a promising approach to bridge the gap between the lack of any semantics and a full-fledged formal specification as in architectural description languages. Closely related work can be found in both Allen and Garlan's work [Allen, 1994b] as well as in the work on contracts by Helm et al [Helm, 1990].

In both models, components may have one or more interfaces, each with its own formal specification based on finite state protocols. Their connectors are first-class, reusable components in their own right and can support n-party interactions. They use a stronger compatibility rule that allows them to deduce deadlock and livelock freedom using the local checks alone. We extended this work with glue code generation that allows a more flexible compatibility check leading to more generic and more reusable composition patterns. Using a compact set of known terms to document components and composition patterns also improves the reusability of our connectors.

2.1.6 Formal Specifications

2.1.6.1 Description

There also exists significant work in specifying the behavioral constraints on components. Assertion languages, pre- and post condition specification, design-by-contract (from Eiffel), Contracts, [Helm, 1990] [Meyer, 1992] [Beugnard, 1999], have all been focused on ensuring behavioral match between a system using a component and the component itself. The main topic is how components can be trusted and what can be done if a component behaves unexpectedly. The proposed solution is to provide every component with a contract that specifies what the component does.

2.1.6.2 Relation with This Work

There exist clear similarities in our work with this approach. Our usage scenarios explicitly define constraints on the interaction scheme between components. The difference between this work and our approach mainly lies in the concept of composition patterns. In our approach, we not only check compatibility of a set of components but also whether there exist a common trace in the combined behavior of the components with a wanted usage scenario. Another difference is the usage of abstract primitives rather than specific API calls.

2.1.7 Higher Level API's

Interesting related work for the construction of our set of primitives is found in "The DARPA Knowledge Sharing Initiative" [Genesereth, 1992;KIF, 2001]. This project defines the Knowledge Interchange Format (KIF). Currently this language is used (among others) as a communication language for agents. To do this they propose a set of "performatives". These "performatives" are similar to our primitives. We add a sequence diagram and a mapping to the API to show the behavioral aspect and a possible implementation but the basic idea is similar. I.e. defining a set of predefined terms to specify how parts of an application work together.

Another interesting reference in this context is [Mclennan , 1998]. The Austin Product Center in Texas develops oilfield systems and application libraries that are used all over the world in different offices and field units. By performing API usability tests (how easy the API is to learn, what misconceptions or errors programmers make using the API, etc.…), they show that providing code examples successfully supported a deep understanding of the libraries. The code examples give a better understanding of the purpose of the library, the usage protocols and the usage context. We believe that usage scenarios (see section 3.4.7) can bring similar benefits while being more general than examples.

2.1.8 Separation of Concerns

Aspect Oriented Programming, Subject Oriented Programming, Generative Programming, Adaptive Programming, Composition Filters and many others are all research topics that try to solve the separation of concerns problem. This research is under constant evolution, but important milestones in this research include the Law of Demeter, the technical report on separation of concerns by Christina Lopez and Walter Huersch, the PhD thesis by Christina Lopez: "D - A language framework for distributed programming" [Lopez, 1997], GenVoca by Don Battory, the paper "Subject-Oriented Programming (A Critique of Pure Objects)" by William Harrison and Harold Ossher [Harrison, 1993], the position paper "Composition-Filters Based Real-Time Programming" by J. Bosch and M. Aksit and the book: "Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns" by Prof. Dr. Karl Lieberherr [Lieberherr, 1996].

The goal of this research is fundamentally different from what we try to do. "Separation of Concerns" is about building programs by using a specific tailored aspect language for every concern in an application and to weave these aspects or concerns together afterwards. In that sense, it has nothing to do with component composition. However, as attempts were made to introduce aspects in component based applications, as this research also needs to generate "glue code" and as it often uses similar terminology as we do, we discuss a couple of papers out of this research to avoid confusion.

The term composition filters suggests that this research could be relevant for our composition patterns. After all, we try to compose components. However, composition filters should be viewed as "adapters" or "wrappers". I.e. all interactions between a set of components or objects go through a (set of) composition filter(s). These filters can be defined and programmed independently of the application. This allows the developer to localize and modularize crosscutting concerns. Composition filters do not define interactions as our composition patterns do, but they adapt interactions.

The term composition pattern is also used in the Subject-Oriented Programming community. See the paper by Clarke and Walker at ICSE 2001 [Clarke, 2001]. However, the goal of their composition patterns is different. We use composition patterns to describe reusable compositions of components and use them to check if the components match with the corresponding roles described in the composition pattern. They want to use composition patterns to specify crosscutting requirements or aspects independently of a given design. So their goal is separation of concerns, my goal is component composition.

So why do we use the same term? It seems to be very hard to come up with a name that is not "taken" yet. We have used many names to indicate composition patterns (micro-architectures, role cooperation, usage patterns, etc.) but nearly all terms have so many co-notations today that none of them really suits our needs. Therefore, we just decided to stick to the term composition patterns and usage scenarios.

Another related term that needs clarification is AP&PC (Adaptive Plug-and-Play Components) and Aspectual Collaborations. The terms contain the words collaborations and components, but the idea here is that aspects should be treated as components. These components are then woven automatically in the application. The focus is again on separation of concerns and aspect weaving but this time in the context of components. However, this is an interesting idea and a colleague of mine currently tries to apply these ideas to our work.

2.1.9 Conclusion

There exists much relevant work to build on. However, at this point in the text it is difficult to discuss the relevance of this work for our research. Therefore, a lot of the related work discussion is spread over this work. A short overview:

In section 2.2.1 we give an overview of existing component models and the architectural mismatch issue. Section 3.3 gives an overview of existing specification and documentation techniques specifically focused on interactions. The introduction of section 4.5 further discusses related approaches to compatibility specifications and compliance checking. In section 5.1.2 we discuss existing approaches to generate mismatch feedback. Section 5.3.1 and 5.3.2 describe the most relevant algorithms in detail. These sections are mainly focused on adapter generation techniques and adaptive programming research. The latter is not directly research meant to generate mismatch feedback, but we indicate how results obtained in this very different research area can be applied to mismatch feedback.

2.2 Component Model

2.2.1 Introduction

A brainstorm session with people of the industry about the exact nature of components produced very confusing results. It seems that the word "component" covers nearly anything ranging from 2 bytes of memory with some bit manipulation in embedded systems, over XML documents describing large sets of medical images to full blown applications consisting of million lines of code. The real problem however is that all these definitions are valid in their own context. To avoid confusion in this work we give a definition of components for our context.

Szyperski and Pfister [Szyperski, 1997] proposed the following definition at the 1996 ECOOP conference:

Component definition

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"

Formulated at the ECOOP 1996 conference.

This definition describes the kind of components we use in this work. We do not state that this definition is the only right one. In our view, the main property that distinguishes components from other concepts as class libraries is the composability property. A component needs to provide "hooks" to combine the component with other components. This is where the component model comes in. A component model describes the exact way that is used by components to interact. There are plenty of possibilities. A very well known model is "pipes and filters". This technique is known from the Unix world [Bach, 1986]. Unix provides operators in its command line window to pipe Unix applications together. This makes these programs real components in the sense of the definition above. They are independent (you can run them stand alone), they are configurable (most Unix programs have a bunch of command line arguments), they are black box (they are binaries) and provide a number of tasks and services and they provide a standard means to connect them with other components. It is clear that one needs to specify how this is accomplished. Unix programs that are going to be used in pipes and filters connections need to explicitly open pipes, send data to pipes and close pipes using a predefined library (for an introduction to programming pipes in C see for example [Marshall, 1999]).

Java Beans [EJB, 2001] is a more recent example of a component model. This model adheres to the principles as proposed by the Model-View-Control [Krasner, 1988] pattern. Java Beans implement this architecture by providing "addListeners", "removeListeners" and "notifyListeners" methods. This allows them to fire events (call methods) on all subscribed listeners. This is clearly a different composition scheme than the pipes and filters of the Unix components.

We can extend this list with call backs, shared data resources, blackboard systems, layers and many others (see [Garlan, 1993] for a good overview) all having different standards on how components interact. It is not difficult to see that it would be very hard to combine components that are written for different component models (the problems are related both with technical as semantic properties of the different component models). Allen and co. made a similar observation when they tried to build their Aesop system (a tool to support architectural design and analysis [Allen, 1994b]). In [Garlan, 1995] they explain the idea of architectural mismatch or why it is so hard to build systems out of existing parts. They propose four ways to improve this situation: make architectural assumptions explicit, construct large pieces of software using orthogonal subcomponents, provide techniques for bridging mismatches, develop sources of architectural design guidance.

2.2.2 Our Opinion

At the technical level, there exists a wealth of research that tackles the problem of bridging the gaps between several component models, platforms and languages (see for example the ActiveX to Java Bean bridge [BeanActiveXBridge]). These tend to work only for component models that have related semantics (as Active X [ActiveX, 1999] and Java Beans [EJB, 2001]). However, we are confident that this problem is going to be dealt with in a more pragmatic way. More precisely, the first company that constructs a set of "killer" components will not only sell its components but also the associated component model. Indications of this process can be found in the set of component models that exists today. The best known component models all have a background of very successful software companies (or consortiums), like Unix with pipes and filters, Microsoft with ActiveX [ActiveX, 1999], Sun with Java Beans [EJB, 2001], etc… I.e. component models without a very wide background stand no chance to become adopted. Therefore, we think that this issue will be solved one day, either by an international standard or by a defacto standard. Even when this turns out to be more difficult than expected we can still apply our research in a context where we work within the same component model. This work is mainly useful to build construction kits for a given domain. I.e. one component builder provides a set of components and a set of typical composition patterns. It is possible that the component builder reuses a third party component but he will bring these in the same component model and provide a suitable documentation. The user is only confronted with a set of components using the same component model and a consistent documentation.

Therefore, the research presented here ignores component model issues and starts from a world were there exists only one component model. In practice, we use the Java Bean component model throughout this text. We have chosen this component model just for practical reasons. There is no fundamental reason why we could not apply the concepts of this work on other component models that allow complex interactions between components.

The previous explained why we neglect the component model issues in this work. However, the architectural mismatch problem as described in [Garlan, 1995] also exists within the same component model and even in the same construction kit. In the last five years, we saw the advent of many solutions for the mismatch problem. Among them Architectural Description Languages [Allen, 1997;Luckham, 1995;Taylor, 1996], Mismatch detection [Compare, 1999], Design and Architectural Patterns [Alpert, 1998;Gamma, 1995] and Adapter Generation [Reussner, 1999;Shu, 1989;Yellin, 1994a;Yellin, 1994b;Zaremski, 1997].

In this work, we build on this research to provide automatic compatibility checks, mismatch detection of components and roles, and to suggest adapters to overcome the mismatches.

2.3 Why Components?

At the time, I started my computer science studies we saw the advent of the first program generators. I remember me to use Dbase III+ [Weber Systems Inc., 1985] to construct a simple program to keep track of the members of the women organization my mother was leading at that time. It had a simple wizard that constructed a database application with a bunch of input screens, a set of reports and a set of tables and queries. No programming knowledge at all was needed to construct an application that was capable of browsing, editing, printing and displaying a set of records. I naively believed that it would take only a couple of years before the first "generic" program generators would be constructed and that plain programming would disappear. At that time, there was a lot of research in this area but none of the results made the same quantum leap as I noticed in the database world. The problem turned out to be far more complex than expected.

This inspired me to take a look at the reuse community. If it was not possible to get rid of the programming effort, why not try to reuse somebody else's work? I learned that reuse mostly existed at the implementation level and that it was quite difficult to lift the abstraction level because of the weak link between analysis and design documents and their corresponding implementation.

Object Orientation was expected to come to the rescue. We know now that this technique only partially lived up to this claim [Szyperski, 1997]. The main contribution offered by object orientation for reuse is its introduction of modularity in the implementation that corresponds with the modular elements found in the analysis and design models. The main issues why it did not solve the reuse problems are dependencies and granularity. Select a class in an object oriented application that you can reuse elsewhere and you often end up with many other classes to support this class if not the complete application.

These observations lead to the component concept. A component is typically self-contained and specifically designed for reuse. This makes it easier to reuse a component than a class or an object. However, this improved reusability came at a price. The price was adaptability and extensibility. You can either use a component or not, there is no way to "patch" it to work (at least not with the same flexibility as classes and objects can be adapted).

What convinced me to use components is that I found back "non programmers" tools again. Components inspired tool vendors to build visual programming environments. These environments were in the beginning mainly successful for GUI building, but recently visual "wiring" was added to these tools. This allows you to construct applications without writing any line of code. However, this does not mean that a "non programmer" is able to use these tools. The wiring is in fact just the visual counterpart of a line of code and it involves exactly the same knowledge to program the code as to draw the right connections.

It is clear however that components offer the potential to lift the abstraction level of the current programming effort and this work tries to be a little part in this process.

2.4 Why Composition Patterns?

As I explained in the previous section, visual wiring of components still involves a lot of knowledge. I already indicated that you need to be a programmer to use these tools, but things are even worse. It also forces the developer to go into the details on how to use the component. Does the component need to be initialized? What API should be called to get the wanted functionality? To answer these kinds of questions, the developer needs to browse trough piles of documentation and needs a clear understanding of standard programming concepts.

Note that the knowledge on how to use components comes on top of the knowledge needed to know the functionality of a component. There exists a clear difference between knowledge needed to select components and the knowledge needed to construct an application with them. Selecting components for an application can be done by anyone that reads the requirements and reads the documentation of the available components. However, building an application with them involves programming knowledge.

With this work, we want to make the wiring process as simple as the component selection process. It should be enough for a developer to understand the relation introduced by a given wiring on a conceptual level to select and use the right wiring to construct the wanted application.

Several other people in the research community also noticed this problem. Most of the research effort was invested in the promotion of wiring to a first class object. I.e. it must be possible to define, save and reuse wiring independently from the components. However, it was unclear how this could be achieved. One of the promising solutions was the introduction of connectors [Ducasse, 1997;Pintado, 1992]. In very rough terms: "package the glue code in a component to make it reusable". However, this solution tends to shift the problem to connecting connectors with components.

While we were searching for a better solution for the wiring problem, design patterns [Alpert, 1998;Gamma, 1995;Lajoie, 1994;Mikkonen, 1998;Riehle, 1997] came into view. Many design patterns contain exactly the kind of information you would expect in reusable wiring schemes. An observer pattern not only explains that it is a good idea to split a view from its model, but also indicates how these roles should work together to obtain this split. What's more, it does so independently of a specific implementation. This was exactly what we were searching for. However, design patterns have one main problem to make them useful for our problem. Design patterns are not formally defined. They contain natural language descriptions, sketchy UML [UML, 2001] diagrams and implementation examples. To build automatic tool support based on these patterns involves a formalization phase. The question whether this is desirable or not is still an open discussion (see [Coplien, 1996] for a typical discussion between the main players in the field). The main argument of the group against formalization (including the inventors of patterns) is that formalizing patterns make them less reusable. The group in favor of formalization claims that design patterns need to be formalized to define what they are and to build tool support on top of it.

We agree with both points of view. Therefore, we propose to combine both approaches. Informal design patterns are a great way to learn developers how they should design an application. Formally, described design patterns are better to describe the situation in a given application. I.e. to pass experience one should use the informal version, to document existing software artifacts it is better to use a more formal flavor because it allows automatic tool support. As components already exist, it is natural to use a more formal description to document them.

We are only interested in the formalization of the interaction between roles in a design pattern. We are not trying to formalize what the component does, only how the component is used. A formalization of role interactions is what we call a "composition pattern". A composition pattern thus formally describes the interaction between a set of roles. A role can be considered as a “placeholder” for a component. I.e. a composition pattern describes the interaction between an abstract set of roles and these roles can be filled by different components for every other application.

In this view, a role can be considered as an abstract component that is mapped on a real component at composition time. It is clear that the interactions between these abstract roles are also abstract interactions. This means that at composition time not only the roles of a composition pattern are mapped on real components but also the interactions between these roles are mapped on real component interactions.

3 Documentation

"What do you mean? Do you wish me a good morning, or mean that it is a good morning whether I want it or not; or that you feel good on this morning; or that it is a morning to be good on?"

- Gandalf (J.R.R. Tolkien – Lord of the Rings)

"The trouble was that he was talking in philosophy, but they were listening in gibberish."

- Terry Pratchett, Small Gods

3.1 Introduction

In this chapter, we introduce the kind of documentation we use. As explained earlier we want documentation that supports the combination of components based on typical composition patterns and we want this process to be supported with automatic compatibility checking and automatic glue code generation. We first make four observations to aid in the definition of the set of requirements for our component and composition documentation. These requirements are then used to check existing documentation techniques for their suitability. Finally, in the last section of this chapter we present our documentation.

3.1.1 Observation 1: Usage Differs from Functionality

Today most function libraries are documented with a description of the syntax and a natural language description of every API call (this is documentation on level 1 and level 2 in [Beugnard, 1999]). It seems natural to use the same kind of documentation to document components. However, a component differs fundamentally from such a function library as it maintains state. This introduces the need to document the dependencies between the various methods that can be called on a component (this argument also holds for a class in OO languages). To use a component one needs to know the protocol it uses to interact with its environment to accomplish the desired behavior[1]. This new dichotomy was nicely summarized by Allen and Garlan in [Allen, 1994a] were they distinguish between implementation relationships and interaction relationships of software modules or components:

"Whereas the implementation relationship is concerned with how a component achieves its computation, the interaction relationship is used to understand how that computation is combined with others in the overall system".

A typical example of this situation can be found in the client network component we use in our exam construction kit. Table 1 describes the interface of this component.

[1] In general one needs to know a lot more than the interaction protocol and the functionality of a component alone, to use it. Typical extra information includes platform requirements, performance, dependencies, version information, etc. In this section, we concentrate on the difference between usage, functionality and implementation only.

API	Events
SetHost(host)	connected
SetPort(port)	disconnected
Connect()	connectionFailed
Send()	dataReceived(String aString)
DisConnect()	dataSent(String aString)

Table 1: API of the NetworkClient component

Describing the effect of every call gives the user a good idea of what this component does. To *use* this component on the other hand we also need to know that before we call the "connect" method, we need to call "setHost" and "setPort" to specify where we want to connect. After the connect call we need to wait for the "connected" event before we can call the "send" method and so on. A developer typically uses his or her domain knowledge to deduce this kind of usage information. However, we need to document this protocol explicitly to allow automatic composition checks. In general, we need both kinds of documentation to do a meaningful composition. In this work, we focus on the usage aspects.

3.1.2 Observation 2: Components Have More Than One Use

This observation is not about the fact that a component can be used in many different occasions in many different places, but rather that there exist a different usage protocol for each behavior that is supported by a component. A good example is provided by a generic network component we use in our experiments. This component can be used both as network server and as a network client, it supports sending strings or streaming video, it further supports the management of its clients (it allows to add, delete and get info of all network clients). It supports both point-to-point communications and broadcast communication. It is clear that setting up such a component as a client for example differs fundamentally from its set-up as a server. Therefore, good usage documentation should document all kind of uses for the given component. This is clearly impossible. The developer of a component cannot foresee all the uses that are going to be made of this component. However, we believe that providing the most typical scenarios already covers a huge range of uses. Take for example a look at the simple JButton component in the Java Development Kit. The API of such a component is large (11 API calls on its own and more than 150 inherited from its ancestors). The use of such a component however is nearly always the instantiation of the button with a given label, followed by the subscription of the interested components on its "actionPerformed" event.

3.1.3 Observation 3: Not Everything Is In A Name

Relying on the semantics derived intuitively from the name of an API call or its natural language documentation works surprisingly well (it is by far the most often used technique today), but it is clear that there remains a lot to be wished for. A simple search in the JDK class library on the word "update" results in 52 hits. Comparing the real meaning of these API calls reveals that the use of the "update" API covers very different meanings. In graphical components, this call typically results in an update of the user interface, while in security components; update is usually used to recalculate the security certificate. In general, natural language descriptions work fairly well for interpretation by humans, but it is very hard to create automatic support and tools for component composition.

3.1.4 Observation 4: Composition Patterns Need to be First Class

Even with a very good documentation on how to use one component, it still remains difficult to know how to compose a set of components. The success of design patterns [Alpert, 1998;Gamma, 1995;Lajoie, 1994] proves this point for standard program constructs. Even when a developer masters a given class library and programming language, he still needs to know standard compositions. Take for example the visitor pattern. Suppose a given class supports a way to accept a visitor and that this behavior is clearly documented. Another class is a visitor and also describes clearly how it behaves. Without a generic introduction on how a visitor communication works, it will be very difficult to compose these classes in the right way.

This means that composition documentation is more than a mere aggregate of the interaction protocols of the components it composes. This view is not new. It was already recognized in [Ducasse, 1997].

"Describing software architectures in terms of interaction relationships between components brings us closer to a compositional view, and hence a more flexible or open view of an application [Nierstrasz, 1995]. First-class connectors allow us to view an application's architecture as a composition of independent components. We gain in flexibility, since each component could engage in a number of different agreements, increasing the reuse potential of individual components. Separating connectors from the components also promotes reuse and refinement of typical interaction relationships. It opens the possibility of the refinement of connectors and the construction of complex connectors out of simpler ones."

However, while the need for first class compositions was recognized at that time, this idea is still not used in the current practice of component composition and documentation.

3.2 Requirements

In the previous section, we made four observations about current component and composition documentation. In the first two observations, we explain that "usage" or "interaction" documentation for a component differs from documentation on what the component does. We also note that components typically have more than one way of interacting with other components. In the fourth observation, we recognize the need to consider compositions as first class objects. Based on these observations we search for documentation to specify:

- The sound or typical "usage(s)" of a component.
- Known and generic composition patterns.

As we already mentioned we are searching for component and composition documentation to support automatic compatibility checking and automatic glue code generation. More precisely, we search for documentation that allows us to:

- Check compatibility automatically in the sense that the interaction protocol between a set of components complies with a given composition pattern (see further for a more exact definition).
- Generate glue code automatically to obtain a working composition of components as specified by a composition pattern.

This leads us to documentation techniques that specifically describe interactions between software modules or components. Examples of this kind of documentation are found in protocol checking literature. This literature also provides typical compatibility checks. An interesting check in this context is a check for deadlock freedom. We explain further in this work how we distinguish between two different versions of deadlock freedom. A local check where a component is checked against its corresponding role in the composition pattern and a global one where a set of components is matched against a composition pattern. In the following section, we review some of the documentation techniques that are used in the context of protocol checking and interaction specifications. We reflect on how these techniques can be used to document component usage scenarios and composition patterns. We also check if these techniques are suitable to perform the compatibility checking and the automatic glue code generation. Finally, as we believe that many of these techniques are not used because of their formal nature, we also assess the acceptance of the technique as a measure for its ease of use.

3.3 Existing Documentation Techniques

In this section we review some of the documentation techniques that are used in the context of protocol checking and interaction specifications. We reflect on how these techniques can be used to document component usage scenarios and composition patterns. We also check if these techniques are suitable to perform the compatibility checking and the automatic glue code generation. Finally, as we believe that many of these techniques are not used because of their formal nature, we also assess the acceptance of the technique as a measure for its ease of use.

3.3.1 Communicating Sequential Processes (CSP)

The Communicating Sequential Processes (CSP) language [Hoare, 1985] was introduced by C. A. R. Hoare to describe patterns of communication between parallel processes using algebraic expressions. These may be manipulated and transformed according to various laws in order to establish important properties of the system being described. Behind CSP lies a mathematical theory of *traces*, *failures* and *divergence*s. Traces define the operational model of CSP, while failures and divergences define abstract sets representing circumstances under which a process might be observed to go wrong. The model supplies a precise mathematical meaning to CSP processes, and is consistent with the algebraic laws that govern them.

In the standard operational model of CSP, processes are represented by transition systems. There is a close relationship between the operational model of CSP and the Failures-Divergences model, which means that the former may be used to prove properties of a system, phrased in terms of the latter.

CSP is a mathematical notation. However there are a number of concurrent programming languages based on CSP, such as OCCAM [INMOS, 1988] and ADA [ADA, 2001]. Thus, theoretical results derived using this model are applicable to real programming.

CSP is a very general model to describe communication between processes. The price for its expressive power is that many interesting properties (as deadlock freedom) are not decidable in general.

"The problem of determining whether any given concurrent system can ever deadlock is similar to the famous halting problem of Turing machines – it is undecidable. This means that there can never be an algorithmic method for proving deadlock freedom which will work in the general case." [Mairson, 1989]

Deadlock freedom is proven for many special cases and for many restrictions of the general model. However, this mainly reduces the model to a less general formalism. A promising approach in this field is the work done by Jeremy Martin where the formal basis of CSP is used to prove and construct "rules of thumbs" (or design patterns) for developers and designers to construct deadlock free systems [Martin, 1996].

Thus, while it is easy to specify a system in terms of CSP specifications (using ADA or OCCAM for example), it is not easy to perform a compatibility check based on these specifications. These restrictions always render a system with the expressive power of finite state processes. In that case, it seems natural to use state diagrams as notation rather than these restricted versions of CSP.

3.3.2 Petri Nets

Petri Nets is a formal and graphical appealing language, which is appropriate for modeling systems with concurrency. Petri Nets has been under development since the beginning of the 60'ies, where Prof. Dr. Carl Adam Petri defined the language [Petri, 1962]. It was the first time a general theory for discrete parallel systems was formulated. The language is a generalization of automata theory such that the concept of concurrently occurring events can be expressed. It has since then been the subject of extensive study in the academic world.

This model is widely used as a research tool and is backed by a considerable amount of theory. Important properties for our research as deadlock freedom and liveness are at least for standard Petri Nets decidable [Esparza, 1994] although they often require exponential time. There also exists a lot of tool support to analyze Petri nets. This makes it an interesting choice. The only disadvantage is the academic nature of this model. It is very hard to convince developers to document their systems using Petri nets.

3.3.3 State Diagrams

State transition diagrams have been used right from the beginning in object-oriented modeling. The basic idea is to define a machine that has a number of states (hence the term finite state machine). The machine receives events from the outside world, and each event can cause the machine to transition from one state to another.

State transition diagrams were around long before object modeling. They give an explicit, even a formal definition of behavior. A disadvantage of them is that you have to define all the possible states of a system. While this is all right for small systems, it soon breaks down in larger systems, as there is an exponential growth in the number of states. This state explosion problem leads to state transition diagrams becoming far too complex for much practical use. To combat this state explosion problem, object-oriented methods define separate state-transition diagrams for each class. This pretty much eliminates the explosion problem since each class is simple enough to have a comprehensible state transition diagram. (It does, however, raise a problem in that it is difficult to visualize the behavior of the whole system from a number of diagrams of individual classes - which leads people to interaction diagrams).

The most popular variety of state-transition diagram in object-oriented methods is the Harel Statechart [Harel, 1987]. This was introduced in OO modeling by Rumbaugh[Rumbaugh, 991], taken up by Booch [Booch, 2001] and adopted in the UML [UML, 2001]. It is one of the more powerful and flexible forms of state transition diagram. A particularly valuable feature of the approach at the analysis phase is its ability to generalize states, which allows you to factor out common transitions. They also include concurrent state diagrams, allowing objects to have more than one diagram to describe their behavior.

State diagrams have a firm formal basis in finite state machines. This theory provides us with algorithms for equivalence checks, intersection and difference calculation (product automaton) and the calculation of a parallel composition (shuffle automaton) [Hopcroft, 2001].

The weak point of state diagrams to describe compositions of components is it lack of addressing. In general, a state machine receives events and broadcasts events from and to an “environment”. They do not support further identification of this environment or of the communication channels used.

This issue is addressed in many extensions on state machines. Among these, SDL [Ellsberger, 1997] is one of the best-known formalisms. The theory to deal with these addressing issues is also very well covered in the protocol checking, conversion and adaptation literature. One of the best descriptions on how state machines can be extended with addressing and used to check interactions between components can be found in [Brand, 1983]. This work is extended and improved by Zaremski [Zaremski, 1997], Reussner [Reussner, 1999]and Yellin and Strom [Yellin, 1994a;Yellin, 1994b].

State charts comply very well with our requirements except maybe for there ease of use.

3.3.4 Sequence diagrams

A sequence diagram shows a typical interaction between a number of parties over time. It shows the parties participating in the interaction and the messages that they exchange. These messages describe a communication between parties and the receipt of a message is normally considered as an event. Parties can be abstract entities as roles or implementation entities as software modules, objects and components.

Sequence diagrams where first popularized by Jacobson [Jacobson, 1992] but by far the best-known sequence diagram notation is the UML sequence diagram [UML, 2001]. The latter is not the most expressive notation though. The most serious limitation is it lack of support for loops and optional parts and its awkward definition of alternative constructs (this is done by specifying conditions on a message per message base).

A more complete and expressive version of sequence diagrams is Method Sequence Charts (MCS's). These sequence diagrams are typically used for the formal specification of telecommunication protocols (often in combination with SDL [Ellsberger, 1997]). The first formal version called MSC'92 was standardized by the ITU as Z. 120 [MSC, 1993]. MSC'92 had more or less the same expressive power as the UML sequence charts. This version evolved in MSC'96. As this new version supports basic messages, loops and alternatives it has at least the power of regular expressions and improves considerable on the "single trace" semantics of the UML sequence diagram.

This kind of documentation is widely used and fulfils most of our requirements. Sequence diagrams in UML are typically used to define use cases [Jacobson, 1992] and are thus very well suited to describe "usage information". Their correspondence with regular expressions means that we have algorithms to translate sequence diagrams to state machines and vice versa (see [Hopcroft, 2001] pp 91-104). This allows us to use sequence diagrams as notation and use state machine theory to do the compatibility checks.

3.3.5 Collaboration diagrams

In collaboration diagrams example objects are shown as icons. Arrows indicate the messages sent in the use case. A sequence is indicated by a numbering scheme. Simple collaboration diagrams simply number the messages in sequence. More complex schemes use a decimal numbering approach to indicate if messages are sent as part of the implementation of another message. In addition, a character can be used to show concurrent threads. The UML notation guide describes the relation between a sequence diagram and a collaboration diagram as follows:

"A pattern of interaction among instances is shown on an interaction diagram. Interaction diagrams come in two forms based on the same underlying information, specified by an interaction, but each form emphasizing a particular aspect of it. The two forms are: sequence diagrams and collaboration diagrams. Sequence diagrams show the explicit sequence of stimuli and are better for real-time specifications and for

complex scenarios. Collaboration diagrams show the relationships among instances and are better for understanding all of the effects on a given instance and for procedural design." [UML, 2001]

This quote tells us that collaboration diagrams and sequence diagrams have the same expressive power and that the choice between them depends on the context or even personal preference. As both presentations have the same underlying model, it is easy to translate one representation in another.

3.3.6 Summary

The most general of the models described above is CSP. CSP describes protocols as interacting concurrent processes. In this framework, one can specify all protocols and most of their properties. The cost of this generality is the undecidability of most properties [Mairson, 1989]. Therefore, existing methods of analysis use human assistance or take advantage of the fact that many protocol features do not use all the generality available (as is done in Wright [Allen, 1994b] for example). This allows a protocol to be analyzed as if it were described in a less general formalism. A Petri net is a less general model. In this formalism, protocols are more easily analyzable, and most of the properties we need for our research are decidable for Petri nets. However, Petri nets are not easy to use and are therefore mainly used as theoretical background or in very specific domains.

The least powerful model is that of a single finite-state machine describing the system including all the component processes and interconnecting channels. In this model, only certain protocols can be described. (For example, a protocol allowing an arbitrary number of messages in transit cannot be described.) Nevertheless, describable protocols are relatively easy to analyze in the sense that all properties are decidable by exhaustive analysis.

Therefore extending this model to use one finite state machine for each component and adding addressing information to define the component interactions combines the best of both worlds. As state diagrams can be converted in regular expressions (and hence in sequence diagrams or collaboration diagrams) and vice versa, we have the option to use sequence diagrams or collaboration diagrams as notation and use state machine theory to perform the compatibility checks.

3.4 Our Documentation

3.4.1 Introduction

In the previous sections, we introduced a set of requirements describing what we want to document and we described in very general terms what we want to do with it. As the documentation we are searching for is mainly interaction documentation (be it typical composition patterns or typical "usage" scenarios of components) we now turn to the question how exactly we are going to document this information.

Based on the little survey in the previous section we propose to use a special kind of Message Sequence Charts (MSC's) [MSC, 1993] for this goal. Each component is documented with a set of MSC's. Each MSC describes a typical "usage" scenario supported by the component. The main difference with standard MSC's lies in the signal labels. We have chosen a compact set of primitives with a predefined meaning. The use of this standard set of labels avoids the confusion that stems from the interpretation of the meaning of a set of API calls from its intuitive semantics. Using a standard set of labels also allows the reuse of documentation. As the usage scenario is no longer bound to one specific interface, the same scenario can be used to document two different components that have the same behavior but use other interfaces. We explain later in this work how these labels are mapped on real API calls and output events. This "interface mapping" allows us to generate code afterwards.

3.4.2 Scenarios

Figure 8 summarizes our scenario syntax. Our syntax is mainly a subset of the MSC syntax containing a set of participants, a set of signal sends between these participants and a set of control. This section describes these syntax elements and their meaning.

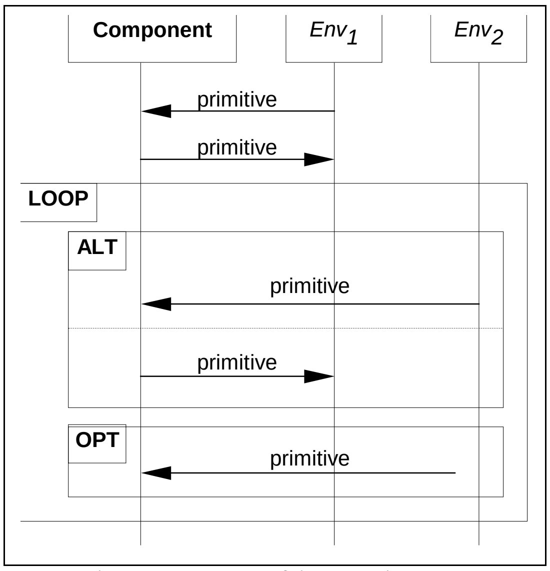


Figure 8: Summary of the scenario syntax

3.4.3 Participants

Sequence charts describe interactions between a number of participants. We want to use this documentation to document both components and composition patterns. A component usage scenario describes the interaction for one “main” role (the component itself) and a set of environment roles. A composition pattern on the other hand describes the interaction between a set of roles. This introduces a small difference in the way we document components and composition patterns.

Participants for Components

For components, we introduce "environment" participants. An environment participant stands for any other cooperating component or glue code. The sequence diagram specifies a contract for any component or glue code that plays the role of this participant. It specifies what kind of messages the component expects from its environment and in what order.

As can be seen a scenario contains exactly one "component" participant. All other participants are "environment" participants. An "environment" participant is labeled ENV_i. A "component" participant is labeled with the component name. It is important to split up the behavior to as many environments as possible. If a message could be sent to or received from a different environment, this should be documented. The reason why is discussed in detail in section 4.10.

Participants for Composition Patterns

For composition patterns, every participant stands for a role that will be filled by a component at composition time. A composition pattern specifies a contract that describes a typical interaction between a set of roles. Thus, composition patterns are described independently of specific components. This makes these patterns reusable in many different contexts.

3.4.4 Messages

3.4.4.1 Introduction

Our documentation uses the standard MSC graphical symbols, but the labels for the signal sends are taken from a compact set of terms with an agreed meaning. Those terms are then mapped on the API of the component. This stands in contrast with standard MSC's messages that are expressed directly in terms of API calls. Building automatic tool support based on concrete API calls is very difficult. The "update" API call in a GUI component for example has not the same meaning as the "update" API call found in a database component. It takes a human and a lot of documentation to distinguish the two. This makes it very difficult to construct automatic tool support. The primitives we propose are used to map API calls from very different sources. Mapping a set of API calls from one component on for example the primitive "CONNECT" indicates that these API calls correspond with a set of other API calls on another component that are also mapped on the primitive "CONNECT".

The idea to use a small set of primitives was inspired by the natural language research done by Schank [Schank, 1973]. The exact nature of the language representation used by Shank is beyond the scope of this work. Suffices it to say that he provides a small set of primitives with a known meaning together with additional syntax to express any natural language phrase in a language independent manner. An example of such a translated sentence is given in Figure 9. “PROPEL” is one of the primitive actions with an agreed meaning. The additional syntax shows that “John” initiates the “PROPEL” action and that “Ball” is the object of the action.

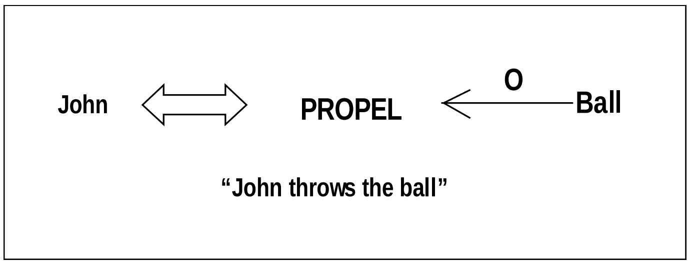


Figure 9: Research by Shank for a minimal set of primitives to bring natural language to a canonical form

The important concept behind this research is the difference between the associative theory of meaning and the constructive theory of meaning. The associative theory of meaning tries to define the meaning of concepts by relating it to other concepts. An often-used analogy is to try to learn a foreign language by tracing the explanation of a word in a dictionary. Many analysis techniques are implicitly based on the associative approach. Take for example an association between two classes in a UML static structure diagram. This diagram implies that there exists a relation between these two classes without defining the meaning of the classes. This diagram informs the developers that the system contains two entities that are related. Without further information, the developer could extract the main structure of the system, but it is doubtful that the same technique can be used to describe the meaning of every entity.

The constructive approach on the other hand starts with a set of known concepts and relates all other concepts to these. The latter approach is the basis of the research of Shank. In many formal systems this approach is used were meaning is based on mathematical concepts. While it is possible to document the meaning of every API call in these formal systems, this is often overkill for the properties we want to check. Therefore, we try to bridge the gap between these full-fledged formal systems and informal documentation, by introducing concepts in between by definition and relate the meaning of API calls to these concepts.

Figure 10 shows the set of primitives we use in our experiments. These primitives are classified in a simple hierarchy. This hierarchy is used during the matching process described further in this text in the sense that we allow subtypes to map on super types and vice versa.

Important note:
This set of primitives is just a proof of concept. We do not claim that this is the only set of primitives or even that it is a good set of primitives. We use this set for our experiments only. However, it gives indications on how such a set should look like and how it can be organized.

From our limited experience in building a set of primitives for our experiments, we learned that it is very hard to come up with a general set that is usable for all kinds of domains. One should rather construct a set of primitives for a specific application domain. Therefore, we state that this approach is especially useful to build "construction kits". It gives developers the opportunity to build a set of components and to document for that set how they should be used and combined. Part of this research is done for the Advanced Internet Access (AIA) project [Wydaeghe, 2000] were we try to build construction kits for Internet services. For this project, we built a construction kit that allows us to build all kinds of distributed exams for the Internet (real time, offline, multiple choice, open questions, authorized, non authorized, with or without multimedia, etc.).

The set we present in Figure 10 proved to be sufficient to document all components and composition patterns in this construction kit. This set was constructed during an iterative process of several months. We started with a basic set of primitives that simply seemed to be reasonable and adapted it based on the feedback from people documenting the exam components and composition patterns.

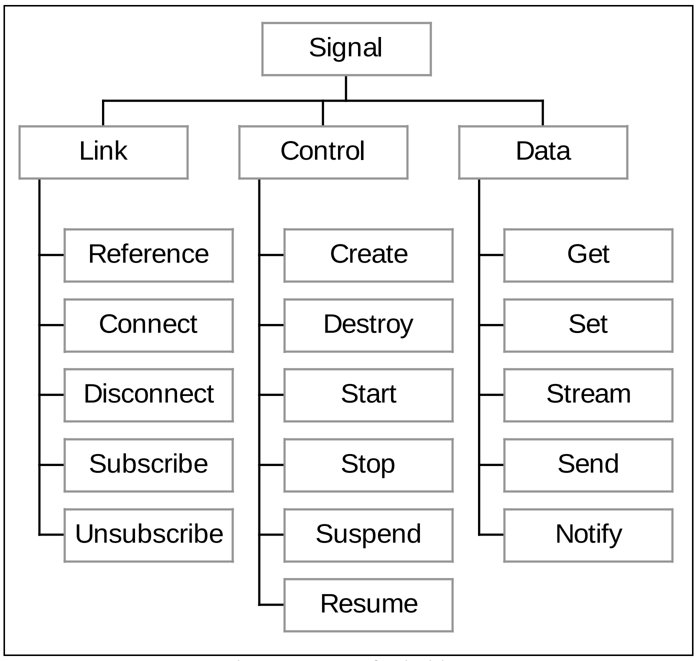


Figure 10: Set of Primitives

The agreed meaning for this set of primitives is given in Table 2. The meaning for the higher order primitives is defined by the aggregate of the meaning of its children. The meaning is given from the viewpoint of the initiator. If a message is labeled with SEND going from party one tot party two, it is clear that one party actually sends data, but the other party receives this data. The latter is not indicated in the table.

LINK PRIMITIVES	
Reference	Acquire an explicit link to another participant
Connect	Initialize a communication link with another participant
Disconnect	Stop a communication link with another participant
Subscribe	Register with another participant.

Unsubscribe	The unsubscribing participant expects that "Notify" messages be no longer sent to him from the other participant.
CONTROL Primitives	
Create	The participant creates a participant.
Destroy	The participant destroys a participant.
Start	Request to start a service on a participant
Stop	Request to stop a previously started service on a participant
Suspend	Request to suspend a service on another participant
Resume	Request to resume a previously suspended service on another participant
DATA primitives	
Get	Fetch data from another participant
Set	Set data on another participant
Stream	Continuously receiving or sending data from or to another participant.
Send	Instantaneously receiving or sending data from or to another participant.
Notify	Inform an other participant

Table 2: Agreed meaning of primitives

The previous table only gives an intuition for the meaning of these primitives. In practice, the meaning and the differences between these primitives are established during multiple iterations and during actual use. It is just a set of agreements between people providing a set of component usage scenarios and their typical composition patterns.

3.4.4.2 Hierarchy of Primitives

As can be seen in the set of primitives presented in Figure 10, we introduced a hierarchy in our messages. The *signal* primitive is the most general one. We want this primitive to match with any other primitive. We recognized the need for this kind of hierarchy while modeling very general components. A typical example is the JButton Java Bean included in the Java Development Kit. This very standard button sends out an event each time it is pressed. Of course, this button also supports a list of other behavior (setting its caption, size, icon, events for mouse clicked and mouse released, etc…), but we restrict ourselves to the typical use of this button: sending out an event whenever it is pressed.

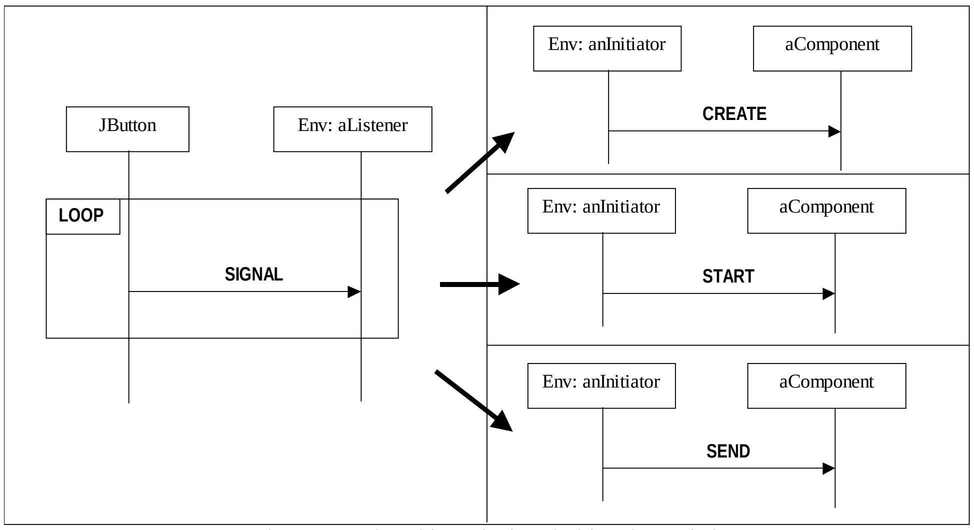


Figure 11: Why a hierarchy in primitives is needed.

Without hierarchy we would document this button with a loop over the NOTIFY primitive. This would mean that a button could only be combined with another component that waits for a NOTIFY message (see section 4.5 for a definition of compatibility). This is not what we want. We want to be able to express that whenever the button is pressed it needs to start another component or that it needs to send a string over a network, or anything we can imagine. The problem is that anything can happen because of pressing a button. Therefore, we model a button as a loop over the SIGNAL primitive (Figure 11). Now the button defined in the left part can be matched with any of the components on the right side of the picture.

The same argument does not hold for messages that are called on a component. While an output event can result in all kinds of actions, a method call always means the same thing. Specifying both the output events and these message calls in generic terms reduces the value of the compatibility check. This leads us to the following rule of thumb:

In general one should document output events of components (outgoing messages from the component participant) using the most general primitive that is applicable, while messages that are called on a component should be documented as specific as possible.

3.4.5 Control Blocks

We use the OPT, ALT and LOOP keywords from the MSC syntax. The OPT keyword means an optional block and the ALT keyword indicates alternatives. The LOOP keyword indicates iteration over a part of the scenario (i.e. zero or more times).

3.4.6 Mapping

For components *only,* we also provide the mapping between the set of primitives we use in the MSC's and the real API calls and output events. This gives the possibility to deduct how a given component implements a required behavior. To be complete, two different mappings should be provided: one to map participants in the composition pattern to components and one to map primitives to API's.

Every participant of a composition pattern should be linked with an existing component. The "environment" participants are mapped later to other components automatically (see the matching process in chapter 4).

The mappings allowed for primitives on API calls are different for outgoing messages as for incoming messages.

We allow every outgoing primitive to be mapped to one event or to a set of alternative events. We allow the latter because it provides a better alternative for non-deterministic documentation. A little example makes this clear. A typical interaction between a user interface and a FTP component is a request to make a connection to a given host followed by a notification of "success" or "failure". If we do not allow the "success" and the "failure" event to be mapped on the same NOTIFY primitive we would obtain non-deterministic behavior at the primitive level. Indeed, we would need to document this interaction as a REQUEST followed by an alternative where both options contain a NOTIFY primitive with the first one mapped to "success" and the second one to "failure". This leads to serious problems in compatibility checking and for the code generation (For a discussion on this kind of non determinism see section 4.11). Now, suppose that the user interface component only wants to print the result of the connect request. In that case, we want to specify that, whatever event is thrown as result we want to invoke some kind of display method on the user interface (if we do feel the need to specify different traces, this means that the set of primitives is not specific enough). Allowing an outgoing primitive to be mapped on several output events allows us to do just that, making the documentation of components much more natural.

We allow incoming primitives to be mapped to one API call or to a sequence of API calls. Allowing incoming primitives to be mapped on a set of alternative API calls makes it impossible to generate code automatically. (In practice, we also allow this option in our tool and we rely on user input during the code generation process to make the choice).

Finally, if a primitive term is contained in a LOOP block this term is mapped only once.

3.4.7 Documenting Components and Composition Patterns

The documentation introduced in the previous sections is used to document both components and composition patterns. The documentation for components is straightforward. For every component a usage scenario describes the interaction of the component with its environment. Thus, our component documentation contains exactly one main participant and a set of environment participants. It also contains an implementation mapping for every message used in this usage diagram. In Figure 12 we depict the documentation for our UserExamControl component. This component provides a user interface for multiple-choice exams. This component is launched by a "setVisible" call. From then, it starts to receive commands and it sends the selections done by the user to an environment.

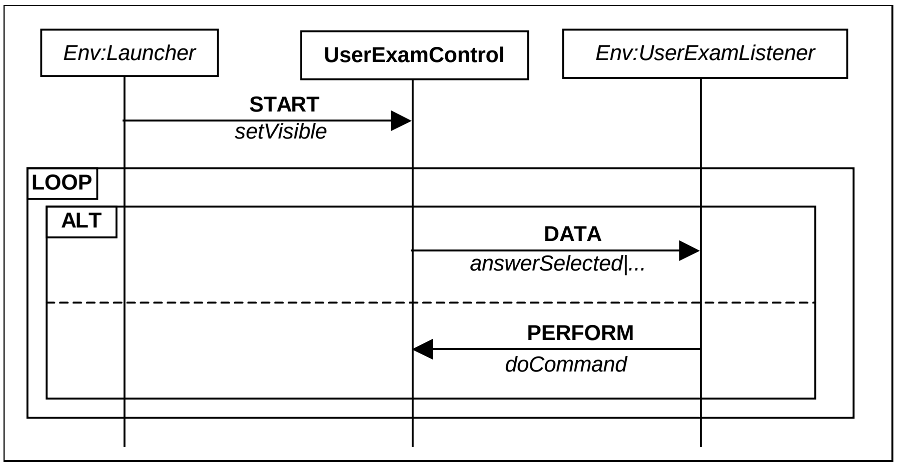


Figure 12: Documentation of the driving exam client interface

Composition patterns are documented in a very similar way. I.e. a composition is also documented using a scenario that uses the fixed set of primitives we introduced. A composition pattern describes the interaction between a set of role and can thus be viewed as a kind of use case for (a part of an) application. As a composition pattern describes an interaction between roles, it does not contain environment participants or implementation mappings. A composition pattern is a high level description of the cooperation between several roles without any indication on how this cooperation will be implemented. Figure 13 depicts a composition pattern describing the interactions between a network role (i.e. a role that provides access to a network) and a role using this. As you can see, the only difference with the component documentation is that the primitives are not mapped on concrete API calls and there is no longer a "main" role depicting a specific component.

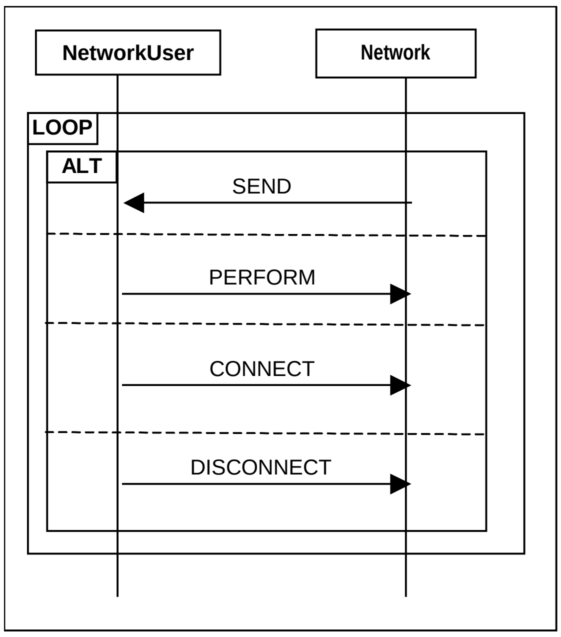


Figure 13: Network Interaction Composition Pattern

4 Matching

"Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication."

- SIGPLAN Notices Vol. 17, No. 9, September 1982, pages 7 – 13

"It is easier to change the specification to fit the program than vice versa."

- SIGPLAN Notices Vol. 17, No. 9, September 1982, pages 7 – 13

4.1 Introduction

This section is about compatibility checking. In the previous chapter, we introduced our proposal for documenting components and composition patterns. The goal of this documentation is to facilitate the component composition process. More specific, we envision a tool were a set of components and composition patterns are selected to build a component-based application. Once the composition pattern and the components are chosen, the developer indicates which components map on which roles in the composition pattern.

The next step is to check the compatibility between this set of components and the composition pattern. This is the focus of this chapter. To do this we use automata theory. This involves transformations from usage scenarios and composition patterns to automata. The resulting automata of these transformations are called component automata and composition automata respectively. On these automata, we then define our notion of compatibility. We distinguish between the compatibility of a component with a role and the compatibility of a set of cooperating components as specified by the composition pattern.

Part of the ideas in this chapter are based on existing theory but an important other part is the result of explorative research. This means that every idea was implemented in our prototype implementation and tested. These tests then inspired improved algorithms and/or new theory. This means that the algorithms and definitions in this chapter present the current status of our research. We only present the latest version and we do not describe the intermediate solutions that lead to the current algorithms and definitions. There are some exceptions however. We present for example two algorithms to perform global compatibility checking. The second version was inspired by the performance problems with the first algorithm. However, we do discuss both versions here because they both have their virtues. The first algorithm is intuitive and easy to implement, while the second alternative is more efficient.

The structure of this chapter is as follows. First, we define the transformation of usage scenarios and composition patterns to automata. Next, we introduce a set of operators on these automata. Using these definitions and operators we then define our notion of compatibility. After that, we present first an algorithm to check local compatibility and an algorithm to check global compatibility. For this global check, we provide two alternatives. In the final sections of this chapter we describe in more detail how the component/role mapping is used and we describe algorithms to search for suitable role/component mappings automatically. We also describe an additional compatibility check, to check the mapping of environment participants on roles. Finally, we explain the effect of the message hierarchy we introduced in section 3.4.4.2 and we discuss the issue of non-determinism.

4.2 Mapping MSC's on Automata

In the previous chapter, we introduced a special kind of sequence charts to document usage scenarios and composition patterns. We want to use this documentation to perform compatibility checking. While MSC's are easy to use for the end user, it is better to work with automata for the actual compatibility checks. In this section, we describe how we translate component usage scenarios and composition patterns to component automata and composition automata. These automata are defined further in this section.

As the scenarios we use are directly compatible with regular expressions, the conversion to a DFA seems to be straightforward. The interested reader can find proves of equivalence between regular expressions and automata in [Hopcroft, 2001]. A summary of the conversion is depicted in Figure 14.

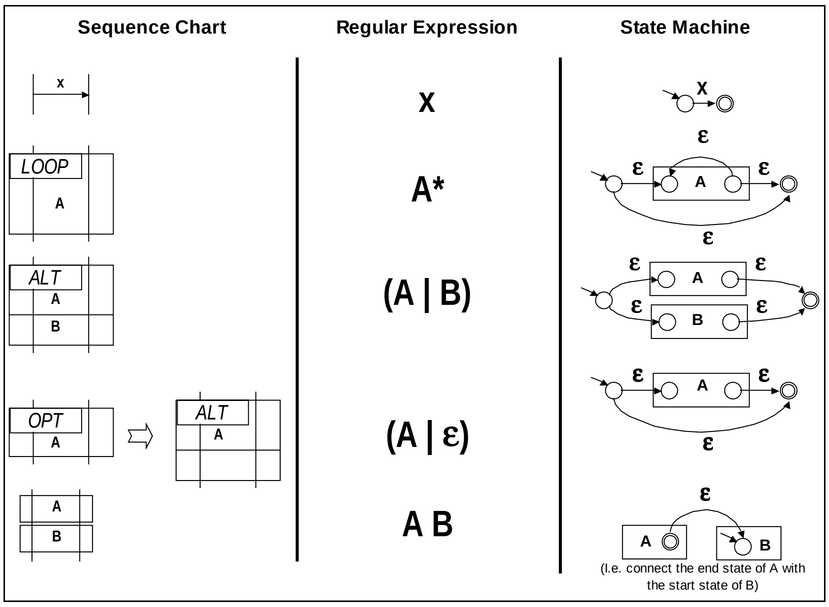


Figure 14: Converting sequence charts to automata

If we do not want to lose information during this transition, we need to label the transitions of the resulting component and compositions automata with more than the message label alone. Especially the direction of the messages is important. Therefore, we add "Out" or "In" to the transitions of a component automaton to indicate incoming and outgoing messages from the viewpoint of the component. An example of this conversion is depicted in Figure 15.

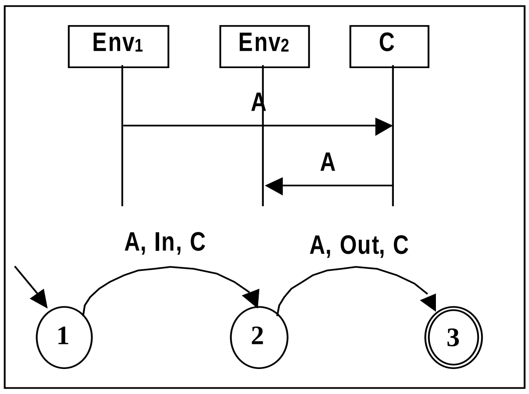


Figure 15: Adding "relative" direction for component usage scenarios.

In this conversion, we loose the environment mapping information. I.e. we do not distinguish anymore between a message A coming from Env_1 and a message A coming from Env_2. We come back on this issue in section 4.9. For now suffices to say that we ignore environment mappings in our compatibility checking algorithms and that we perform an additional check afterwards to deal with them.

To convert composition automata we need to do one extra step. Note that every transition in a component automaton is either an incoming message or an outgoing message. However, if we use again the standard conversion algorithm [Hopcroft, 2001] on composition automata we end up with transitions that specify the sending and receiving of a message in one transition. Therefore, we split every message of the composition automaton in a sending and a receiving part. Thus, every message in a composition scenario results in a sequence of two transitions. The first transition is labeled with the message going out of a role and the second transition is labeled with the message going in a given role. The result is depicted in Figure 16.

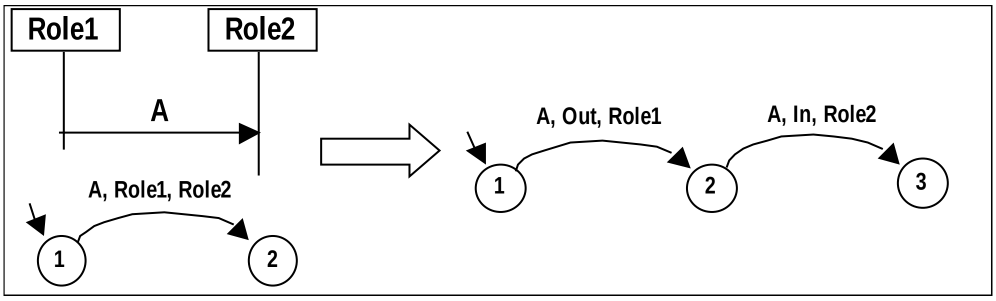


Figure 16: Splitting messages in the composition pattern into a sending and a receiving part

4.3 Definitions: Component Automata and Composition Automata.

The results of the mapping described in the previous section are component and composition automata. These automata are defined as follows:

DEFINITION: Component and Composition Automaton

Component and composition automata are both standard deterministic automata. They are defined by the well known five tuple (S, q_0, F, succ, Σ) where

- S = set of states
- q_0 = initial state $\in$ S
- F = set of final states $\subseteq$ S
- Σ = a set of labels
- succ is a partial function mapping S x $\Sigma \rightarrow$ S. succ(s,α) is the state entered after the label α is accepted in state s.

The difference between the composition automata and the component automata lies in the structure of their labels.

A label of a component automaton corresponds with a message in the component usage scenario. It is a four tuple (name, direction, component, implementation). For a component usage scenario of a component c with a set of environment participants E, this five tuple is defined as follows:

- name $\in$ {signal, notify, set,…} the set of primitive messages for the application domain
- direction $\in$ {In, Out}
- component = c
- implementation is the actual implementation in terms of the interface of the component. This is a set of output events if direction = Out and a list of API calls if direction = In

It is important to note that all component automata need to be complete. Complete here means that the alphabet of a component automaton always contains all possible labels corresponding with a message received or sent by its corresponding component c. Thus $\forall$ name $\in$ {signal, notify, set,…}: $\forall$ direction $\in$ { In, Out}: (name, direction, c) $\in \Sigma$

Likewise, a label of a composition automaton corresponds with a message in the corresponding composition pattern. It is a four tuple (name, direction, source, destination) expressed in terms of the set of roles R of this composition pattern where:

- name $\in$ {signal, notify, set,…} the set of primitive messages for the application domain
- direction $\in$ { In, Out}
- role $\in$ R (r is the role that sends the message if direction = Out and r is the role that receives the message if direction = In)

4.4 Operations on Component and Composition Automata

In the following, we often use a set of operations on the component and composition automata defined above. These are introduced here.

4.4.1 Projection of a composition pattern automaton

The projection of a composition pattern CP to a role R is the restriction of this composition pattern to the interactions that have role R as source or destination.

DEFINITION: Projection of a composition pattern to a role

We denote the projection of a composition pattern CP to a role R as:

$$P_R(CP)$$

It can easily be constructed from the full composition pattern by replacing all transitions that are labeled with a message that does not interact with role R (source and destination $\neq$ R) with an epsilon transition. Calculating the epsilon closure of this automaton calculates the projection. The set of messages of the projection is the original set without these non-interacting messages. The set of roles of the projection is the original set of roles without the roles that are not the source or the destination of any message in the new set of messages.

4.4.2 Applying the role map function

In the following, we often need to replace every occurrence of a role name in the messages of a composition automaton with its corresponding component name. Labels of a composition automaton are four tuples containing source and destination roles. To allow us to compare composition automata and component automata we translate these role names in the component names based on the mapping given by the user. This operation is defined here.

Applying Role/Component Map Function

Let CP = (S, q_0, F, succ, Σ) be a composition automaton

Let MAP be a total function that maps role names on component names if the role is mapped on a component and is the identity function if the role has not been mapped on a component.

Then CP_{MAP} = (S, q_0, F, succ, Σ') denotes the composition automaton after applying the MAP function to CP

$$\Leftrightarrow$$

$$\Sigma' = \{m = (\text{name, direction, component}) \mid \exists\, m = (\text{name, direction, role}) \in \Sigma: \text{MAP(role) = component})$$

4.4.3 Parallel Composition

To provide a more formal basis to our algorithms we introduce the parallel composition operator for component and composition automata in analogy with the parallel composition operator of CSP. We use this operator later to prove equivalence of our compatibility checking algorithms. It describes both the independent execution of each of the components as well as the performance of a joint step.

We define the parallel composition for component and composition automata as follows:

Definition: the parallel composition operator

A parallel combination of two component or composition automata P_1={S_1, q_1, F_1, $succ_1$, Σ_1) and P_2 =(S_2, q_2, F_2, $succ_2$, Σ_2) is described as:

$$P_{1\,\Sigma_1}\|_{\Sigma_2} P_2$$

In this combination, P_1 can perform events only in Σ_1, P_2 can perform events only in Σ_2, and they must simultaneously engage in events in the intersection of Σ_1 and Σ_2. There are two rules that define the possible transitions of a parallel combination. One rule describes the independent execution of each of the components, and the other describes the performance of a joint step.

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_{1\,\Sigma_1}\|_{\Sigma_2} P_2 \xrightarrow{\alpha} P'_{1\,\Sigma_1}\|_{\Sigma_2} P_2} \left[\alpha \in (\Sigma_1 \setminus \Sigma_2)\right]$$

$$P_{2\,\Sigma_2}\|_{\Sigma_1} P_1 \xrightarrow{\alpha} P_{2\,\Sigma_2}\|_{\Sigma_1} P_1'$$

$$\frac{\begin{array}{c} P_1 \xrightarrow{\alpha} P_1' \\ P_2 \xrightarrow{\alpha} P_2' \end{array}}{P_{1\,\Sigma_1}\|_{\Sigma_2} P_2 \xrightarrow{\alpha} P_1'{}_{\Sigma_1}\|_{\Sigma_2} P_2'} \left[\alpha \in \Sigma_1 \cap \Sigma_2\right]$$

This corresponds with a new automaton

(S_1 x S_2, [q_1, q_2], F_1 x F_1, succ', $\Sigma_1 \cup \Sigma_2$) with

succ': (S_1 x S_2) x ($\Sigma_1 \cup \Sigma_2$) $\rightarrow$ S:

- succ'([s_1,s_2],α) = [t_1,t_2] $\Leftrightarrow \alpha \in \Sigma_1 \cap \Sigma_2$ and $succ_1$(s_1, α) = t_1 and $succ_2$(s_2, α) = t_2
- succ'([s_1,s_2],α) = [t_1,s_2] $\Leftrightarrow \alpha \in (\Sigma_1 \setminus \Sigma_2)$ and $succ_1$(s_1, α) = t_1
- succ'([s_1,s_2],α) = [s_1,t_2] $\Leftrightarrow \alpha \in (\Sigma_2 \setminus \Sigma_1)$ and $succ_2$(s_2, α) = t_2

In our definition, we define the result of the parallel composition as a new automaton where the new set of states is the cross product of the set of states of the component and composition automata under composition. This is not obvious. If we allow the component and composition automata to be non-deterministic we need to take the power set of states and we need to define the transition function in terms of state sets (i.e. we need to define the result as the cross product of two non-deterministic automata. See [Hopcroft, 2001] for a definition). Our definition implies that the automata under composition need to be deterministic. This topic is described in detail in section 4.11.

An intuitive view on the parallel composition is that it allows component or composition automata to perform their own transitions as long as no synchronization is required. However, they need to perform a joint step for labels in the intersection of their alphabets. I.e. they need to agree on common steps, but can go along with their own transitions.

This definition implies that the result of the parallel composition of P_1 and P_2 with totally disjoint alphabets results in the shuffle automaton of these two automata. If P_1 and P_2 have exactly the same alphabet, this operation corresponds to the intersection of these automata.

4.4.4 Laws for the Parallel Composition Operator.

We need to prove two properties of this parallel composition operator. These properties are later used to prove equivalence of two algorithms.

Laws for parallel composition

$$P_{1A} \|_{B\cup C} (P_{2B} \|_C P_3) = (P_{1A} \|_B P_2)_{A\cup B} \|_C P_3 \quad \langle \| - assoc \rangle$$

$$P_{1A} \|_B P_2 = P_{2B} \|_A P_1 \quad \langle \| - comm \rangle$$

(The $\|$ operator is associative and commutative) (see p.70 [Hoare, 1985]).

4.5 Compatibility

In this section, we define our notion of compatibility for a set of components. The compatibility definition that is typically found in literature is that every component needs to be a refinement of the role it is going to play in the composition [Hoare, 1985]. An intuitive interpretation means that the set of traces of the component is a superset of the set of traces of the composition automaton (the interested reader is referred to [Hoare, 1985] for a stronger definition of refinement in terms of failures and divergences). In automata terminology, this means that the language accepted by the composition automaton is part of the language accepted by the component automaton. Informally this means that the component at least offers what the role in the composition pattern asks for, but it allows components to offer more than what was asked for.

This seems to be reasonable. Indeed components are explicitly built to be reusable and this implies some level of generality. Components typically offer more services than what is needed in a given application. Our exam construction kit for example contains a generic network component that can be used as server or as client. A given application will nearly never use both functionalities at once. This definition also has the advantage that it is possible to prove deadlock freedom if the property holds that every component is a refinement of its corresponding role. I.e. a local check of every component suffices to know that the global system will not deadlock [Allen, 1997].

However, we also consider composition patterns to be reusable entities. This means that a generic composition pattern often specifies more than what the components offer. Figure 17 shows a typical composition pattern expressing generic observer behavior. This composition patterns tries to specify that two roles should be connected using some kind of observing behavior. It does not care if the observation is done using an active polling scenario or using a notification scenario. Even if the observation is done, by notification the observer role can refresh its own data by getting the new data or it can ignore the new value (an example of the latter is a notification of a simple button press).

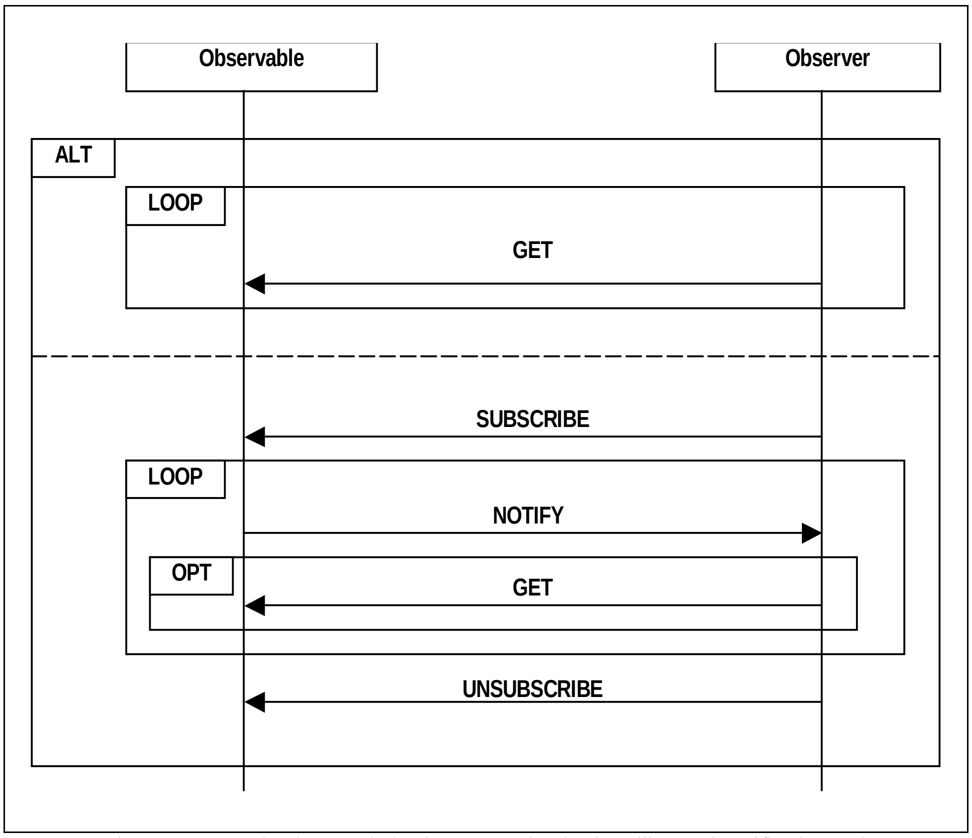


Figure 17: Generic observer behavior supporting both polling and notification style

It is clear that we do not want every component that is used in this composition pattern to implement both possibilities.

However, defining compatibility based on refinement returns a mismatch if a component is used that does not implement everything that the composition pattern asks for. We think that this is too restrictive. Based on this observation we introduce our notion of compatibility in the following sections.

4.5.1 Local Compatibility

Figure 18 depicts a snapshot during the development process of a component-based application using our approach. At this point a composition pattern with three roles is dragged on the canvas (see the “use case” notation consisting of an oval with three connected boxes) and three components are selected to be used in this composition.

By dragging the network component to the "Network" role in the pattern, the developer indicates that this component should fulfill this role in the composition. The provided documentation is used to perform a compatibility check between the component and the composition pattern.

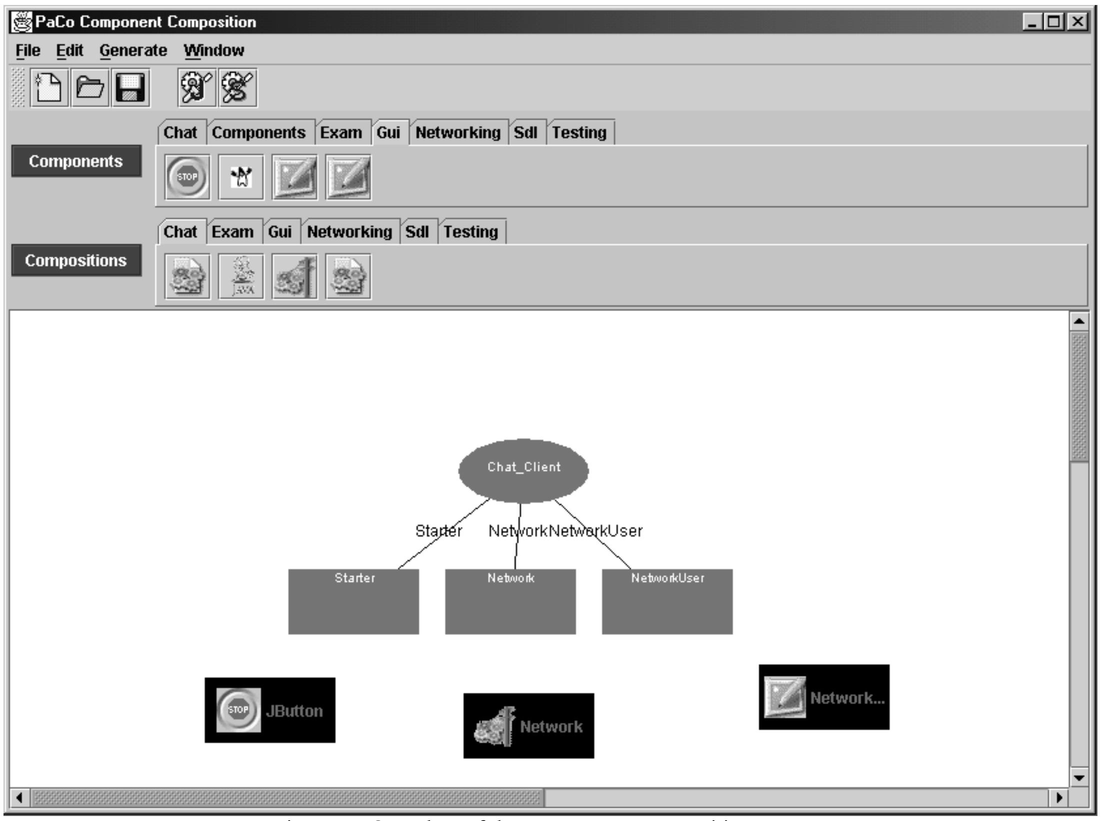

PaCo Component Composition
File Edit Generate Window
Components
Chat Components Exam Gui Networking Sdl Testing
Compositions
Chat Exam Gui Networking Sdl Testing
Chat_Client
Starter
NetworkNetworkUser
Starter
Network
NetworkUser
JButton
Network
Network...

Figure 18: Snapshot of the component composition process

This is a "local" check. I.e. at this point, we can only check the compatibility of one component with a role in the composition pattern. As we explain later in this text, it is possible that all components match with their role in the composition pattern, but fail to match with each other. Therefore, the checking process at this point differs from the global checking process. We now define our notion of local compatibility.

Definition: Local Compatibility

Let C be a component and R be a role of the composition pattern CP. Further let L(C) be the language accepted by the component automaton and let $L(P_R(CP))$ be the language accepted by the automata defined by $P_R(CP)$

Then C is local compatible with R $\Leftrightarrow L(C) \cap L(P_R(CP)) \neq \emptyset$

Informally this means that a component is defined to be compatible with a role if they agree upon at least one trace. This allows a component to offer more than what the composition pattern (the corresponding role) asks for and it also allows the composition pattern (the corresponding role) to specify more than what the component offers.

4.5.2 Global Compatibility

The local compatibility definition however looses the nice property that deadlock freedom follows from the fact that every component is locally compatible with its corresponding roles. The local compatibility definition only guarantees that there is a common trace shared by the composition pattern role and the component, but this trace can be different for every component. An example makes this clear.

Figure 19 depicts a theoretical situation where all the local compatibility checks for the three components at the left hand side succeed but where there is clearly no trace in the three components together that matches with the required trace of the composition pattern. Matching component "C1" with the projection of the composition pattern to role "R1" renders a non-empty intersection. Both the projection and the component send out the message "A" and terminate after that. Matching component "C3" with the projection of the composition pattern around role "R3" also renders a non-empty intersection. Both scenarios accept a message "B" and terminate. The same goes for component "C2" and the projection of the composition pattern to role "R2". The component matches with the second alternative rendering again a non-empty intersection. However, if component "C1" sends the message "A" to component "C2", component "C2" sends out the message "C" to component “C3” this component does not accept this message "C".

The local compatibility check only searches for a non-empty intersection between the component and the role. In this case, the intersections of the components mapped on interacting roles are disjoint (more precisely the intersection of C2 with R2 and the intersection of C3 with R3 are disjoint). Informally this means that it is possible for different components to "select" different traces through the composition pattern that are not necessarily compatible.

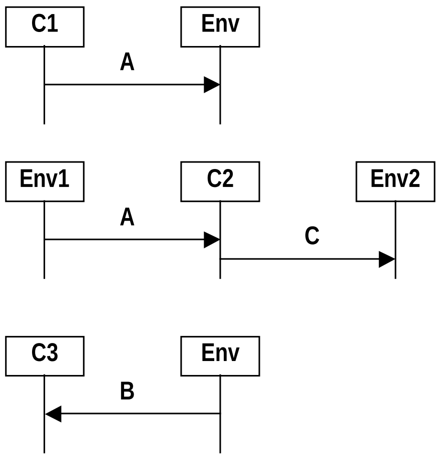
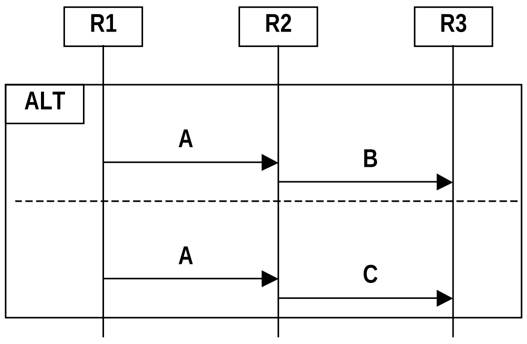
Component Usage Scenarios	**Composition Pattern**
 C1 Env A Env1 C2 Env2 A C C3 Env B	

Figure 19: Why a global check is needed.

From the previous, it follows that we need to take all the components into account to check a full composition. We show that the parallel composition operator does exactly that.

> Let $C_1,\ldots,C_n$ be a set of components and let $CA_1,\ldots,CA_n$ be the corresponding component automata (i.e. $\forall i \in [1..n]:CA_i$ is the component automaton of C_i)
>
> Let CP be a composition pattern with roles $R_1,\ldots,R_n$. Let CPA be the corresponding composition automaton.
>
> Let MAP be a total function such that $\forall i \in [1..n]:MAP(C_i) = R_i$
>
> Then $(CA_1 \parallel \cdots \parallel CA_n)$ describes the concurrent execution of the components, because the alphabets of these components are totally disjoint.

We want to constrain the set of traces in $(CA_1 \parallel \cdots \parallel CA_n)$ to the traces as specified by the composition pattern. We also want to constrain the traces of the composition automaton to the traces supported by $(CA_1 \parallel \cdots \parallel CA_n)$. This means that we need to take the intersection of these automata. I.e. we need to calculate:

$$\text{CPA} \cap (CA_1 \parallel \cdots \parallel CA_n)$$

However, the alphabet of the composition pattern automaton is totally disjoint from the alphabet of $(CA_1 \parallel \cdots \parallel CA_n)$ (which is $\bigcup_{i=1}^{n} alph(CA_i)$) as every label contains a role name and every label in $\bigcup_{i=1}^{n} alph(CA_i)$ contains a component name. Therefore we apply the role/component mapping first giving a new calculation:

$$\mathrm{CPA}_{\mathrm{MAP}} \cap \left(CA_1 \parallel \cdots \parallel CA_n\right)$$

The intersection is however only a special case of the $\parallel$ operator. It is the parallel composition of two automata having the same alphabet. Therefore, we extend the composition pattern with $\bigcup_{i=1}^{n} alph(CA_i)$. Let us denote the extension of the alphabet of the automaton P with the alphabet of the automaton B as $P_{+\mathrm{alph(B)}}$. It is easy to show that $CPA_{MAP_{+\cup_{i=1}^{n} alph(CA_i)}}$ has exactly the same alphabet as $\left(CA_1 \parallel \cdots \parallel CA_n\right)$.

Remember that we made the alphabet of the component automata complete. Thus $\bigcup_{i=1}^{n} alph(CA_i)$ contains all labels describing messages that are sent from, or received in, component C_1 to C_n. As we applied the role/component mapping to the composition automaton, the alphabet of this automaton only contains labels describing messages that are sent from, or received in, the same set of components[2]. Thus *alph*($\mathrm{CPA}_{\mathrm{MAP}}$) $\subseteq$ $\bigcup_{i=1}^{n} alph(CA_i)$ and because we extended the alphabet of $\mathrm{CPA}_{\mathrm{MAP}}$ with $\bigcup_{i=1}^{n} alph(CA_i)$ it follows that $alph\left(CPA_{MAP_{+\cup_{i=1}^{n} alph(CA_i)}}\right) = \bigcup_{i=1}^{n} alph(CA_i)$ and thus

$$\left(CPA_{MAP_{+\cup_{i=1}^{n} alph(CA_i)}} \parallel CA_1 \parallel \cdots \parallel CA_n\right) = \mathrm{CPA}_{\mathrm{MAP}} \cap \left(CA_1 \parallel \cdots \parallel CA_n\right).$$

All this leads to the following definition of global compatibility:

[2] This assumes that all roles are mapped. Our compatibility check fails if this is not the case.

Definition: Global Compatibility

Let $C_1,\ldots,C_n$ be a set of components. Let $CA_1,\ldots,CA_n$ be the corresponding component automata (i.e. $\forall i \in [1..n]: CA_i$ is the component automaton of C_i)

Let CP be a composition pattern with roles $R_1,\ldots,R_n$. Let CPA be the corresponding composition automaton.

Let MAP be a total function such that $\forall i \in [1..n]: MAP(C_i) = R_i$

Then CP is <u>global compatible</u> with $C_1,\ldots,C_n$

$$\Leftrightarrow$$

$$L\left(CPA_{MAP \; + \bigcup_{i=1}^{n} alph(CA_i)} \parallel CA_1 \parallel \cdots \parallel CA_n \right) \neq \emptyset$$

Informally this means that we define a set of components to be compatible with a composition pattern if the composition pattern specifies at least one trace that is part of the set of possible traces resulting from the parallel interleaving of these components.

4.6 Local Check

We introduced our notion of local compatibility in the previous section. This section deals with the practical implementation of this check. The local check is a straightforward implementation of our compatibility definition. The local compatibility check involves four steps:

1. Convert the usage scenario of the component and the composition pattern to a component automaton and a composition automaton.
2. Take the projection of the corresponding role in the composition automaton
3. Calculate the intersection (or the difference) between the projected composition automaton and the component automaton.
4. Check for a start-stop path in the intersection

The first two steps are already explained in the previous sections. Here we take a closer look at step 3 and 4.

4.6.1 Calculating the Intersection

The calculation of the intersection automaton is straightforward using the algorithms described in [Aho, 1985]. This is a standard intersection of two deterministic automata (the component automaton and the composition automaton after the projection to the role corresponding with that component). It is interesting however to discuss the difference between taking the intersection and taking the difference between the component and the projection of the composition pattern.

To check local compatibility as we defined it in section 4.5.1 we need to take the intersection. However, as we discussed above, we often find a more stringent compatibility definition in literature, namely that components need to offer at least everything what the role asks for. It is easy to implement this more stringent constraint in our local compatibility-checking algorithm. To achieve this we need to calculate the difference rather than the intersection in this stage of the algorithm. As this is a simple thing to do, we leave it as an option in our prototype. This indicates how the architectural description as described by Garlan and co. [Allen, 1997] can be checked using automata algorithms instead of using a full-fledged theorem prover.

4.6.2 Check for a Start-Stop Path in the Intersection

Theoretically, a match is found if the product automaton is not empty. However, it is possible for a product automaton to render a result where the start-state is also a stop-state and where this "path" of length zero is the only start-stop path in the intersection. This is obviously not a valid solution in practice because it means that the component fits in the pattern as long as no events take place at all. This raises the question how long a start-stop path should be before we consider the solution to be valid. In the basic version of our prototype, we take a very pragmatic approach and check whether there exists a path of length 1 or more.

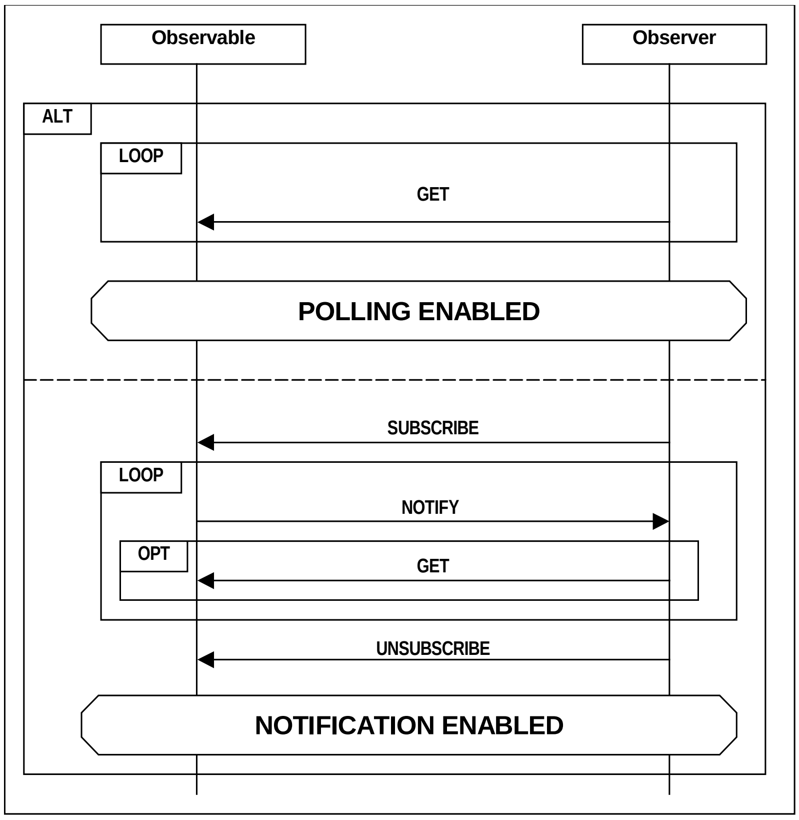


Figure 20: Adding termination properties

A better solution is under implementation. We will annotate the composition pattern with special labels indicating the state of the application if this label is reached. This is illustrated in Figure 20. This composition scenario describes two different observer styles. One based on polling, the other based on notification. If we take these labels with us during the compatibility check, we can check what properties still hold in the resulting automaton. We could then present a list of states that are guaranteed to be reachable to the user.

4.7 Global Check

4.7.1 Classic

We defined global compatibility as:

$$L\left(CPA_{MAP_{+\bigcup_{i=1}^{n} alph(CA_i)}} \parallel CA_1 \parallel \cdots \parallel CA_n \right) \neq \varnothing$$

In our implementation, we split this calculation in two stages. We first calculate the parallel composition of all the components, i.e. $(CA_1 \parallel \cdots \parallel CA_n)$. We then calculate the parallel composition of the result with CP_{MAP}. The last step is the intersection of the composition pattern and the parallel composition of the components. Thus, our implementation of the compatibility check involves the following steps:

1. Convert the usage scenario of the components to component automata ($CA_1 \ldots CA_n$).
2. Convert the composition scenario to a composition automaton CPA
3. Calculate $(CA_1 \parallel \cdots \parallel CA_n)$
4. Apply the role component mapping function MAP to CP to obtain CP_{MAP}
5. Calculate the parallel composition of $CP_{MAP_{+\bigcup_{i=1}^{n} \alpha C_i}}$ and

   $$\left(CA_{1_{+alph\left(CPA_{MAP}\right)}} \parallel \cdots \parallel CA_{n_{+alph\left(CPA_{MAP}\right)}} \right)$$

6. Check for a start-stop path in the resulting automaton of step 5.

Steps one and two are already explained in section 4.2. Step 4 is explained in section 4.4.2. The remaining steps are now further explained.

4.7.1.1 *Calculate the parallel composition of the components*

We now calculate $(CA_1 \parallel \cdots \parallel CA_n)$. As every label on a transition in a component automaton contains the component identification, the alphabets of all component automata are totally disjoint[3]. In that case, the parallel composition operator results in the shuffle automaton of all these components. For performance reasons we implemented the

[3] We do not allow the same instance of a component to be mapped on more than one role.

shuffle automaton algorithm rather than using the more generic parallel composition operation.

Calculating the shuffle automaton itself is a well-known process. It corresponds to the total interleaving of the automata that are shuffled. Figure 9 gives an example for two components with one single message "A". Figure 21 gives an example for two components with one single message "A".

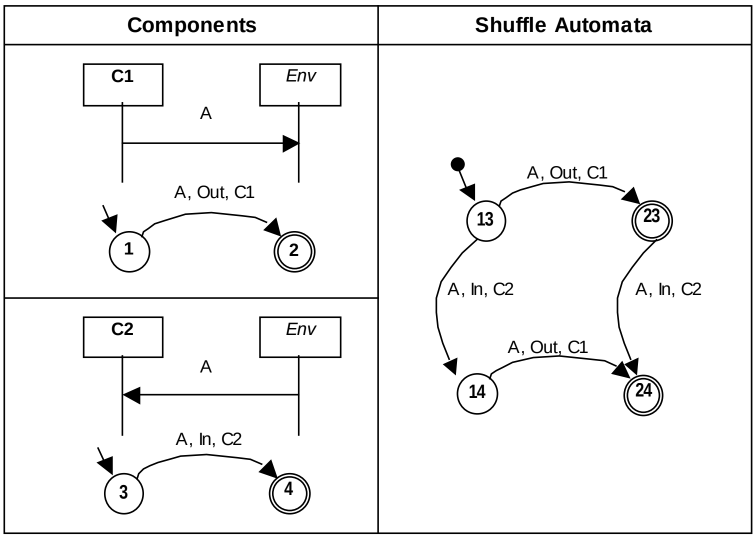


Figure 21: Calculating the shuffle automaton

The result is obtained by advancing in one automaton at the time. For example if we are in the combined state (1,3) we can advance with "A, In, C2" to the combined state (1,4) and we can advance with "A, Out, C1" to the combined state (2,3). Formally:

A shuffle automaton of two automata P_1=(S_1, q_1, F_1, $succ_1$, Σ_1) and P_2 =(S_2, q_2, F_2, $succ_2$, Σ_2) is a new automaton:

(S_1 x S_2, [q_1, q_2], F_1 x F_1, succ', $\Sigma_1 \cup \Sigma_2$) with

succ': (S_1 x S_2) x ($\Sigma_1 \cup \Sigma_2$) $\rightarrow$ S:

- succ'([s_1,s_2],α) = [t_1,s_2] $\Leftrightarrow$ $succ_1$(s_1, α) = t_1
- succ'([s_1,s_2],α) = [s_1,t_2] $\Leftrightarrow$ $succ_2$(s_2, α) = t_2

4.7.1.2 Calculate the parallel composition of the composition pattern and the components

In this step we calculate the parallel composition of CP_{MAP} and $(C_1 \| \cdots \| C_n)$. However, we extend the alphabet of CP_{MAP} with the alphabet of $(C_1 \| \ldots \| C_n)$ to remove all transitions in the resulting automaton of the components that are not compatible with the composition pattern. We also extend the alphabet of every component with the alphabet of the composition pattern to remove all transitions in the resulting automaton of the composition pattern that are not compatible with that particular component. The result is that the parallel composition operator now corresponds with the intersection operator (the alphabets of both automata are equal). Therefore our prototype implements the intersection algorithm directly rather than the generic parallel composition operation. This step is equivalent with the calculation of the intersection during the local check.

4.7.1.3 Check for a start-stop path in the result

Remember that we split every transition in the composition pattern automaton in two separate transitions specifying the send and the receiving of messages. The calculation of $(CA_1 \| \ldots \| CA_n)$ also results in an automaton where every transition corresponds with sending or receiving a message. Thus, the result of the intersection of these automata also contains separate transitions for sending and receiving messages. To check if the system contains traces that terminate we need to find a trace where every outgoing transition labeled (name, Out, $component_1$) is followed by an incoming transition labeled (name,In,$component_2$). Thus the two subsequent transitions need to have the same name and the first transition needs to be outgoing and the second transition needs to be incoming. Figure 22 shows the general template for such an Out/In pair of transitions.

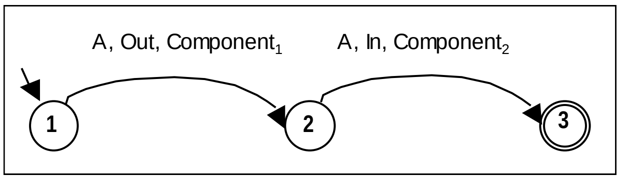


Figure 22: Template for a single message in the intersection automaton going from component 1 to component 2

This template indicates that a component first sends a message and this message is immediately accepted by another component. All traces that send a message out first and receive *another* message afterwards and all traces that receive a message first and send it afterwards are traces that have nothing to do with component interactions.

Therefore, we search for a start stop path where every two subsequent transitions follow the template as specified by Figure 22. If such a path is found, we declare the set of components global compatible with the composition pattern.

4.7.2 Optimization using Asymmetric Cross Products

The previous algorithm is very expensive due to the calculation of the shuffle automaton as this is an exponential process. Now make the following observation.

> As the global checking process ends with the calculation of the intersection between the shuffle of all the component automata and the composition automaton, it is clear that the result needs to be a restricted version of the composition automaton. (Mind that restricted here means accepting a smaller language and not necessarily that the resulting automaton is smaller)

This observation inspired us to the construction of a new algorithm. The idea is to skip the calculation of the shuffle automaton and restrict the composition automaton incrementally with the component automaton. Theoretically, this corresponds to:

$$\left(\left(\left(CP_{MAP_{+alph(C_1)}} \,\|\, C_1\right)_{+alph(C_2)} \,\|\, C_2\right)_{+alph(C_3)} \cdots C_n\right)$$

Thus, the only difference with the previous algorithm is the place of the brackets and where we extend the alphabets. From the associativity and commutativity laws it follows that this renders the same results as the first algorithm. However as the first algorithm first calculates the parallel composition of all components, it will construct a total shuffle first (as the alphabets of the components are totally disjoint) and only restrict this result in the very last step. This leads to a huge intermediate automaton and many useless transitions. The new algorithm on the other hand restricts its result already from the first calculation.

The implementation of this process can be done efficiently using a kind of asymmetric cross product. It involves the calculation of the intersection for all related transitions *only* (i.e. all transitions that are part of the projection of the composition pattern to the role mapped on the component we are intersecting with) and simply adding transitions that are not part of this projection to the result. It is asymmetric in the sense that component transitions are only added when there exists a matching transition in the composition pattern, while transitions of the composition pattern are always added to the result except if they are part of the projection to the role mapped on the component and when they are not compatible with the transitions as specified by the component.

Figure 23 gives an example for a very simple component (named "C1") and a composition pattern. We calculate the asymmetric cross product between this component automaton and the composition automaton. The result contains the transitions "A, Out, C5" and "A, In, C6" because this transition is not part of the alphabet of the component C1. It is thus left intact. It also contains a transition "B, Out, C1" because this transition is related to the component but occurs in both automatons and a transition "B, In, C3" as this transition is not part of the alphabet of the component C1. The transition "C, Out, C1" of the composition automaton is pruned because it is part of the (extended) alphabet of the composition pattern but component "C1" has no corresponding transition and transition "C, In, C2" is kept because it is not part of the alphabet of the component C1.

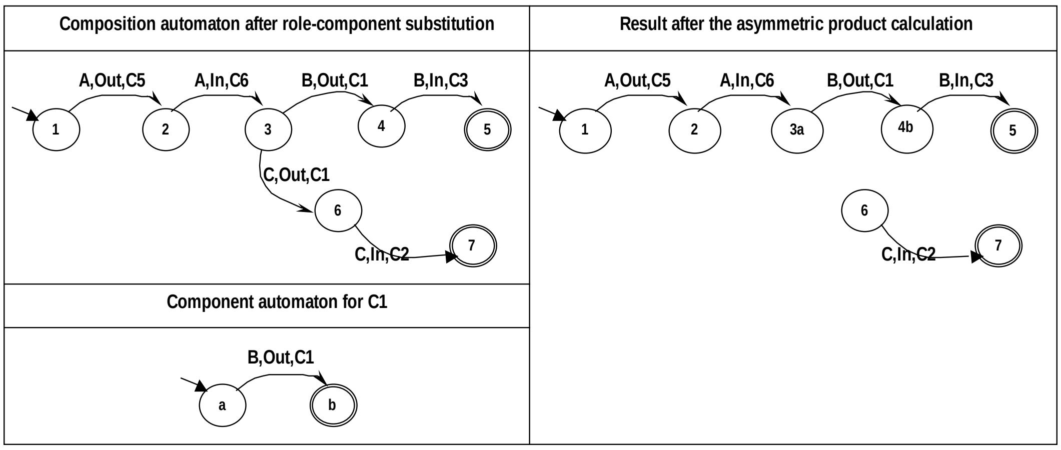

Composition automaton after role-component substitution
Result after the asymmetric product calculation
A,Out,C5
A,In,C6
B,Out,C1
B,In,C3
1
2
3
4
5
C,Out,C1
6
C,In,C2
7
A,Out,C5
A,In,C6
B,Out,C1
B,In,C3
1
2
3a
4b
5
6
C,In,C2
7
Component automaton for C1
B,Out,C1
a
b

Figure 23: The asymmetric cross product.

Another advantage of this algorithm is its incremental nature. This algorithm renders an automaton for partially filled composition patterns. This makes it very well suited for "component generation". I.e. it is possible to take a composition pattern, fill it in partially and use the unfilled roles as new environments for a super component.

4.8 Role/Component Mapping

Before we calculate the parallel composition of a set of component automata with a composition automaton, we use the role/component mapping provided by the user. We use this mapping to bring both automata in the same alphabet. However, if we adapt our equality definition of messages so that we declare two messages to be equal if only their names are equal we obtain a match or a mismatch *without* using the mapping information from components to roles in the composition.

This means that the global check process finds a suitable position for the components automatically.

This makes the presented approach very easy to use. It means that a developer just has to select a composition pattern and a set of components to obtain a working application. He or she does not have to read the composition scenario to find out where every component should be placed.

There are two problems with this approach.

Problem 1: Ambiguous situations occur

One problem occurs for example when a composition pattern specifies two standard button components to launch two different windows. While it does not matter theoretically which button instance is used for which window, it does so in practice. In nearly every component based development tool you are able you to set properties on component instances. A typical property for the button is its caption. Swapping the buttons now renders a very confusion application where the captions of the buttons do not match with the expected behavior. This situation also occurs with two components that have the same usage scenario but different behavior (remember that a usage scenario only describes how to use a component in terms of abstract primitives and not what the component does).

Problem 2: The direction information is not used.

If we do not provide role component mapping information a message send by one component and received by another component can be mapped on a message with the same label going from role X to role Y as well on the same message going from role Y to role X. Thus, the same set of components is compatible with both a composition pattern and the mirrored version of this composition pattern. The situation is even worse because this set of components will also match with the same composition pattern where only some of the messages are mirrored. Without any further checking, our components will happily switch roles during the compatibility check.

We developed an algorithm that deals with these problems. It calculates all "valid" mappings for a set of components and a composition pattern (i.e. all mappings such that all components fit on their mapped role).

The basis of our new algorithm is the first global checking algorithm that calculates the shuffle automaton of the components first and calculates the intersection afterwards described in section 4.7.1 (thus not the asymmetric algorithm). This algorithm uses an explicit role-component mapping to bring the composition automaton and the shuffle automaton in the same alphabet before the calculation of the intersection.

The straightforward algorithm to find these mappings automatically is to calculate the intersection for all permutations of role-component mappings and keep all permutations that render a non-empty product automaton. This algorithm is clearly too expensive. We developed an algorithm based on dynamic programming to circumvent this performance problem.

4.8.1 Overview of the process

We try to obtain all role/component mappings for a given set of components and one composition pattern. This process takes three steps. The first step is already explained in the previous chapters. The second step is the calculation of the intersection with a slightly different equality rule than in the previous. The last step is the proposed algorithm based on dynamic programming. The overview of the full algorithm is thus:

Step 1: Calculate the shuffle automaton of all components as described in section 4.7.1.1

Step 2: Calculate the intersection *regardless* of the role/component mapping

Step 3: Select all traces in this cross product that have non-contradicting component/role mappings

We will now explain step three and four in more detail.

4.8.2 Calculating the intersection without role/component mapping

In this step, we need to calculate the cross product between the shuffle automaton of all the components and the composition pattern. As we do not have a role/component mapping, we ignore role and component names. I.e. we check only the name and the direction of message to decide if they are equal. As the alphabet of the shuffle automaton contains labels expressed in terms of components and the alphabet of the composition automaton contains labels expressed in terms of roles, a joint step in the resulting automaton is a combination of a label of the form (name, direction, component) and a label of the form (name, direction, role). We define the new label for such a step as (name, direction, component/role). The rest of the intersection calculation proceeds as usual. This clearly constructs an automaton with many invalid traces as we allow components to switch roles as they like. In the next step, we remove these invalid traces.

4.8.3 Select All Traces That Have Non-Contradicting Comp/Role Mappings

The labels in the automaton created in the previous step have the form (name, direction, component/role). The semantics of a transition with such a label is that if that transition is followed, we assume that "component" should be mapped on "role". With this observation, we start searching for any path that has non-contradicting mappings.

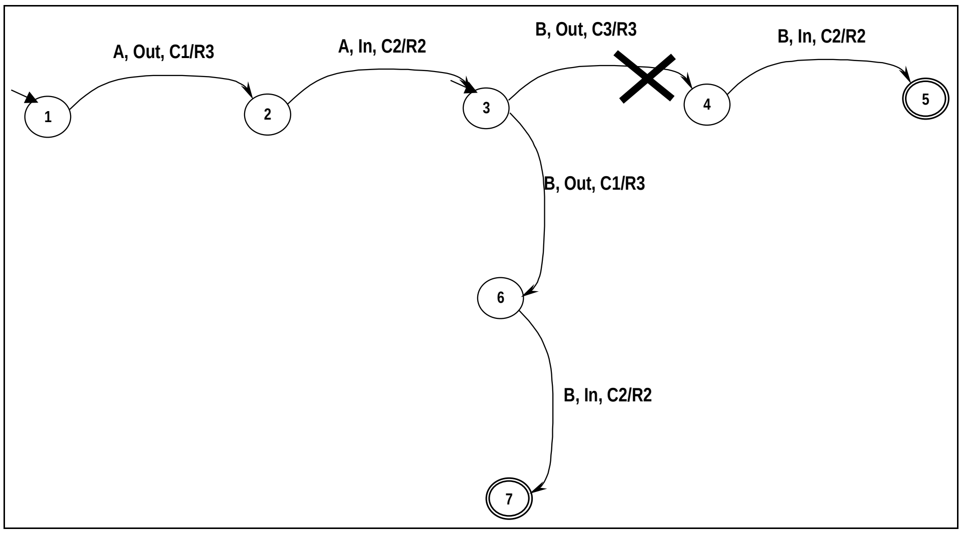


Figure 24: Searching contradicting traces

Figure 24 shows a very small example of the kind of automata we obtain after step 3. The only valid trace in this automaton is from state 1 to state 7 using message "A" and from state 2 to state 3 using message "B". The reason is that following message "A" from state 1 to state 2 implicitly implies that component "C1" is mapped on role "R3". Following message "A" from state 2 to state 3 implies that component "C2" is mapped on role "R2". If we now further follow the trace from state 3 to state 4 using message "B", we read that component "C3" is also mapped on role "R3" and this is not allowed.

We explain further how the resulting automaton is used as a specification for the glue code needed to connect the components. In this context it is acceptable that one component plays multiple roles, but it is not acceptable that one role is played by multiple components (the semantics of the latter are not defined). Therefore, we need to reject the transition from state 3 to state 4 if we first followed state 1 to state 2. This renders the trace: (state1, state2, state3, state 6, state 7) as the only possible trace in this automaton that goes from a start state to a stop state.

To perform an automatic role/component mapping we need to find *all* traces without contradicting mappings in this automaton. It is possible that there exist multiple sets of role/components mappings that render valid traces in the automaton (an example of such a situation is where we use two identical buttons in one composition). In this case, the user should be confronted with the options and he or she has the final choice.

There are a number of possibilities to perform this search. In short:

1. Try all permutations of role/component mappings
2. Perform a breath-first search with history
3. Use a dynamic programming technique

These are now discussed in more detail.

4.8.3.1 Try All Permutations of Role/Component Mappings

While this alternative seems prohibitively expensive, this option still remains open with the observation that we only need to take the permutations of mappings already found in the resulting automaton. Remind that this result automaton is a cross product between the shuffle automaton and the composition automaton based on equality of the messages regardless of their source and destination. If we now enumerate all transitions found in this automaton and read the role/component mappings we will not have that much contradicting mappings. Every mapping will appear in two directions (i.e. if there is a message “A” going from Component 1 to Component 3 and there exists a mapping from Role 2 to Role 4 we will have one mapping Component 1/Role 2, Component 3/Role 4 and one mapping Component1/Role 4, Component 3/Role 2), but chances are low that these roles are also mapped on other components. As mappings that are not included in this list are certain to lead to a dead end, we do not need to consider them. This means that we only need to search for at least one trace in the automata for each permutation of this limited set of mappings. This boils down to enumerating all transitions in the automaton and removing all the transitions that have a mapping that is not included in the permutation we are testing. This is followed by a search for at least one start-stop trace. Any mapping tested this way that returns such a trace is a possible role/component mapping.

4.8.3.2 Perform A Breath First Search with History

This algorithm initially sets out as a normal breath first algorithm. From the start state all outgoing transitions are followed and the destination states are added to the "to do" queue. To avoid that we go round in circles we remember all histories of transitions we took to reach a given state. To proceed we follow all outgoing transitions from the states in the "to do" queue but only if this transition is not yet a part of this transition history.

If the mapping we add to a mapping history contradicts with one of the mappings in there already, we stop following that trace. If a trace comes to a dead end (i.e. all possible continuations follow transitions that are already included in the trace) we check if the trace contains an end state. If so, this trace is a valid solution. In any case, we stop further processing this trace.

The example in Figure 25 shows a possible automaton were we need to resolve the role/component mapping. We will go trough this automaton using the algorithm described above to clarify its working.

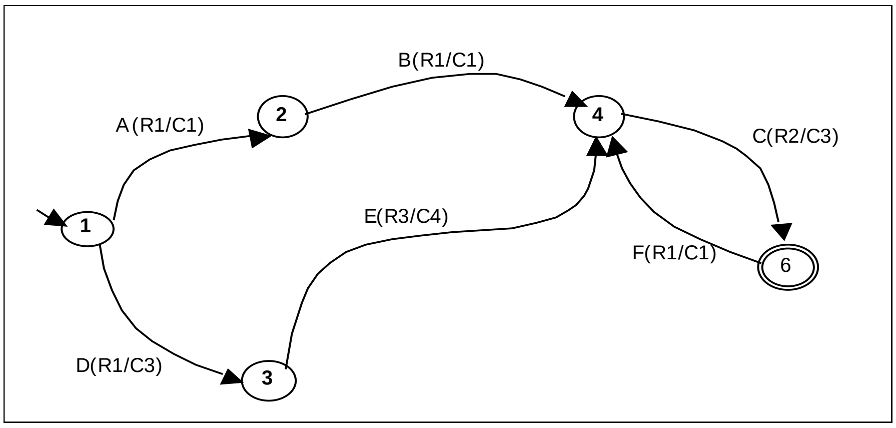


Figure 25: Example state diagram for role/component resolving

In the following we use the symbol Q for the "to do" queue at any given moment. An element in the "to do" queue is a state that needs to be visited and the trace of transitions (together with its corresponding role/component mapping) that has been followed already.

Q={1 (trace: null)}
Pop state 1 and expand.

Q={2 (trace:(A(R1/C1))), 3 (trace:(D(R1/C3)))}
Pop state 2 and expand.

Q={3 (trace:(D(R1/C3))), 4 (trace:(A(R1/C1),B(R1/C1)))}
Pop state 3 and expand.

Q={4 (trace:(A(R1/C1),B(R1/C1))), 4 (trace:(D(R1/C3),E(R3/C4))}
Pop the first state 4 and expand.

Q={4 (trace:(D(R1/C3),E(R3/C4)), 6 (trace:(A(R1/C1),B(R1/C1),C(R2/C3))}
As state 6 is a stop state mark this trace as a solution **(6 (trace:(A(R1/C1),B(R1/C1),C(R2/C3)))**
Pop the state 4 and expand.

Q={6 (trace:(A(R1/C1),B(R1/C1),C(R2/C3)), 6 (trace:(D(R1/C3),E(R3/C4),C(R2/C3))}
Add a second solution **(6 (trace:(D(R1/C3),E(R3/C4),C(R2/C3)))**
Pop first state 6 and expand

Q={6 (trace:(D(R1/C3),E(R3/C4),C(R2/C3)), 4 (trace:(A(R1/C1),B(R1/C1),C(R2/C3),F(R1/C1))}
Pop state 6 and expand.

Q={4 (trace:(A(R1/C1),B(R1/C1),C(R2/C3),F(R1/C1)), 4 (trace:(D(R1/C3),E(R3/C4),C(R2/C3),F(R1/C1))}
We just added state 4 with a trace that contains an incompatible mapping. More precisely R1 is mapped on C3 (A(R1/C1)) and on C1 (F(R1/C1)). As this is not allowed we stop processing this trace any further. I.e. this mapping is just removed from the queue.

Q={4 (trace:(A(R1/C1),B(R1/C1),C(R2/C3),F(R1/C1))}
Now we reach a dead end. If we pop state 4 and try to expand we obtain: **4 (trace: (A(R1/C1) B(R1/C1), C(R2/C3), F(R1/C1), C(R2/C3))**
Note that this trace contains a transition that is already included in the trace (C(R2/C3)), but we are not in a end state. At this point, we check if the history contains at least one end state (as is the case in this example). In that case, we add this trace as a solution. We stop processing it any further.

4.8.3.3 Dynamic Programming Algorithm

The basic idea of this algorithm is to consider every state in the finite automaton as a parallel process. Every process asks all its neighbors (i.e. all states that are connected with this state with an outgoing transition) for a set of mappings such that they can reach a stop state without contradiction. It then checks if the extra mapping needed to reach that neighbor contradicts with that set of mappings. Every mapping that has no contradiction is added to its own set of mappings to reach a stop state. This process iterates until no state receives a new mapping.

In practice, this process is done sequentially instead of parallel. During one iteration, we update every state once. Observe that we only need to update those states that have neighbors who received a new mapping in the previous iteration. As we know that in the first step only those states that have a stop state as neighbor can be updated, we start pushing all stop states on the "to do" queue.

The algorithm then proceeds as follows:

```
while ToDoQueue not empty {
  s = ToDoQueue.pop()
  for every state x that has a transition to s {
    if (isUpdated(x,mappings(s))) ToDoQueue.push(x);
  }
}


boolean isUpdated(State x, State s){
  boolean updated = false;
  Mappings m = s.getMappings();
  for every transition t going from x to s {
    if t.mappings() compatible with at least one element of m {
      Updated = true;
      … update own mapping table …
    }
  }
  return updated;
}
```

Figure 26: Finding all role/component mappings using dynamic programming ideas

This algorithm ends when no state received a new mapping. At this point, the set of possible mappings to reach an end state can be read from the start state.

4.9 Role/Env Mapping

Until now, we only dealt with role/component mappings. These mappings are done manually or automatically, but the "env" participants of a component are always ignored.

Figure 27 and Figure 28 give an example where the "env" participant/role mapping is not one on one. It is easy to see that in both situations the network component on the left hand side of the picture is compatible with the network role in the composition pattern on the right hand side. In the translation of the usage scenario of the network component to a state diagram all information about environment participants is dropped (both in the local as well as in the global check). This means that in Figure 27 implicitly both environment participants of the Network component are mapped on the same "NetworkUser" role in the composition pattern, while in Figure 28 one "env" participant is mapped on the "Initiator" *and* the "User" role of the composition pattern.

All these examples seem to be acceptable at first sight. We want it to be possible that the network component is started by a different component than the one that is going to use the network, but we accept that in some cases the same component creates and uses the network.

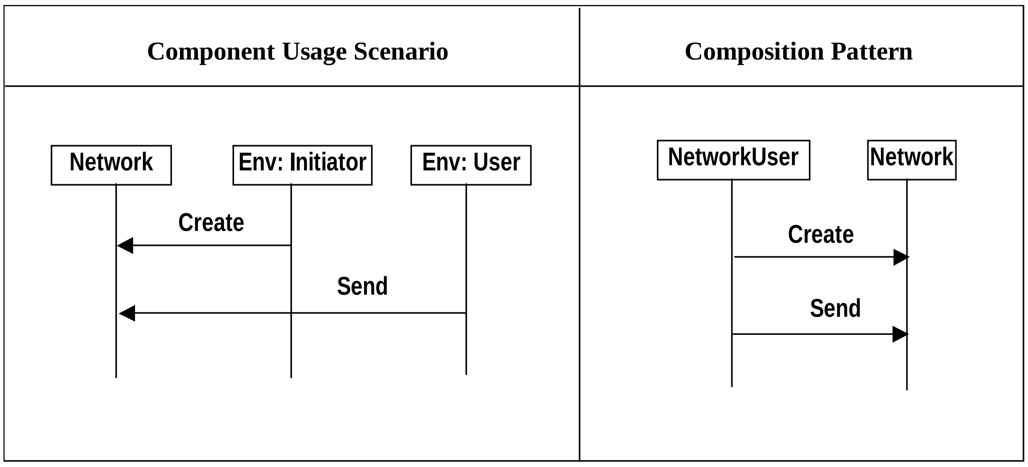


Figure 27: Mapping multiple "env" participants on one role

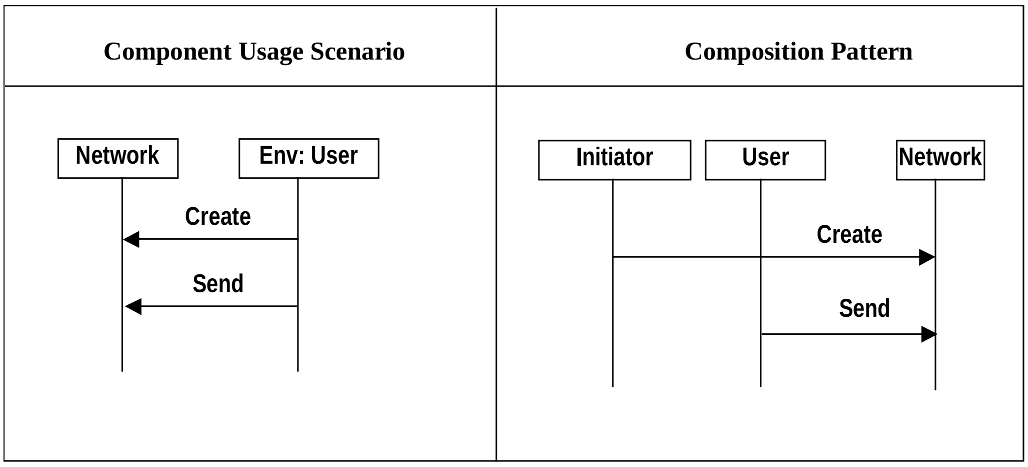


Figure 28: Mapping one "env" participant on multiple roles

Figure 29 shows a situation that is not acceptable. The component has two “env” participants. It expects one “env” participant to give a create message, waits until another environment uses the network to send something and then notifies the sender (for example to tell that the send worked). The composition pattern gives a different interpretation. Here the notification after the first send is given to the initiator (for example to tell the initiator that the connection was used for the first time). However, the current checking algorithms will not complain.

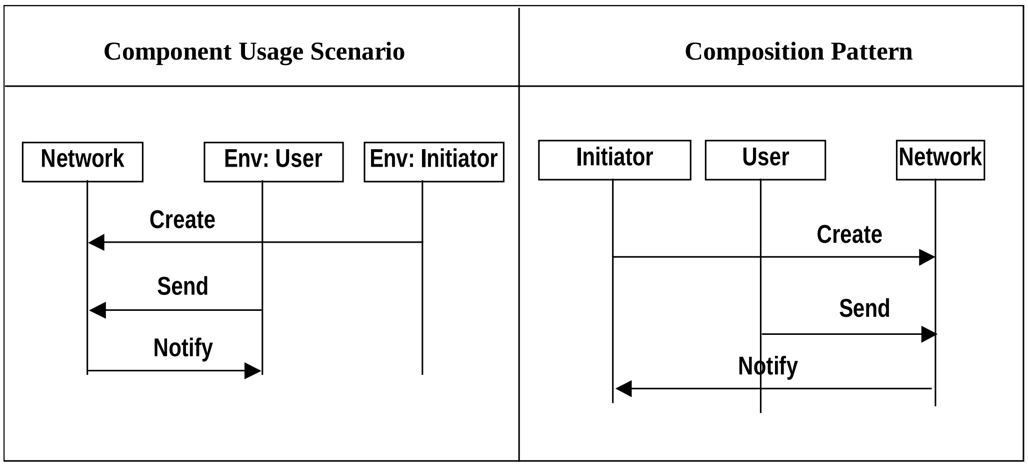


Figure 29: Illegal mapping of "env" participants

4.9.1 Mapping Rules for the “ENV” Participant

It is hard to come up with a rule that accepts the mapping of multiple roles on one “env” participant (as in Figure 28) and the mapping of multiple “env” participants on one role (as in Figure 27), but rejects the situation in Figure 29. The reason is that we expect some kind of *identity* in the example of Figure 29. We want the *same* “env” participant that initiated the send on the network component to receive the notification.

A closer look at the example in Figure 28 learns that we also give up this identity property of the "env" participant in this example. A simple extension of the example makes this clear.

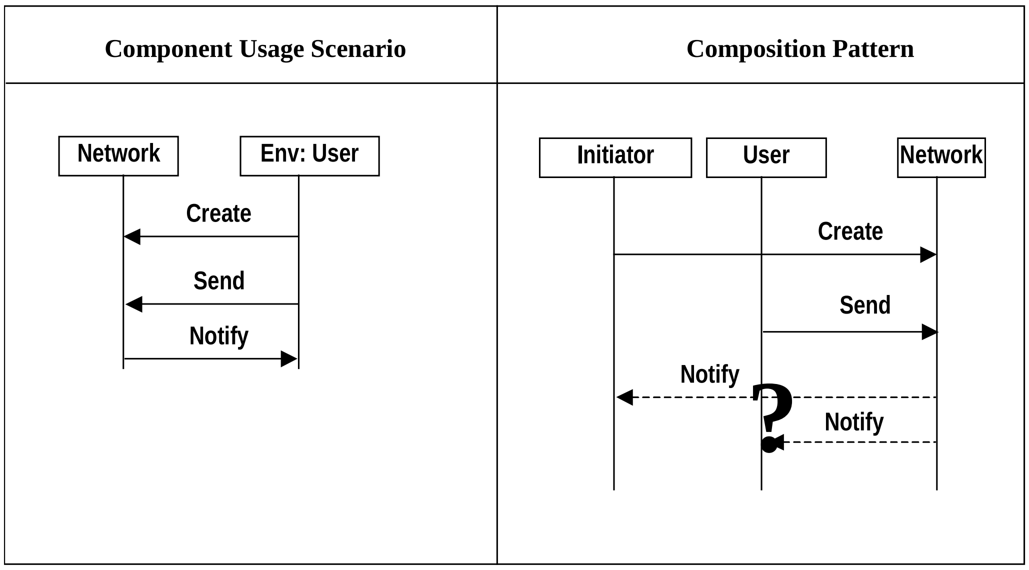


Figure 30: Mapping one "env" participant on multiple roles.

Figure 30 is an exact copy of Figure 28 where we added a notification message in the component usage scenario. The composition pattern uses two different roles to model the behavior of the "env: user" participant. Now we see that it is not well defined where we expect the "notify" message to map. We want it to return to the same role that issued the CREATE and the SEND messages, but this role has split in two different roles in the composition pattern.

These observations led us to the following mapping rule:

Mapping "env" participants

One "env" participant is mapped to exactly one role.

One role can be mapped on many "env" participants.

4.9.2 Checking the "ENV" participant mappings

The algorithms to check local compatibility and global compatibility ignore all information concerning "env" participants. In section 4.9.1 we defined the valid mappings for environment participants. These mappings need to be checked. This is very similar to the problem of finding all non-contradicting mappings in the automatic role mapping process (4.8.3). Thus, we can ignore all environment information and calculate the resulting glue code first. In a next step, we use any of the proposed algorithms for finding non-contradicting mappings (try all permutations, breath first search and dynamic programming) to check the environment mappings. The only difference is that we maintain a list of “env” participant/role mappings instead of role/component mappings. We throw away all traces that imply a mapping of one “env” participant on multiple roles.

4.10 Message Hierarchy

In section 3.4.4.2 we introduced a hierarchy of primitives for the composition patterns and component usage scenarios. While we ignored this in the previous, we need to take a closer look at the impact of this decision on our definition of equality of messages and on their combination. In the following, we call the parent primitive of any sub tree in our hierarchy a "super type" and its descendants a "sub type". It is clear that we want super types to match with sub types and vice versa. Thus in all the previous algorithm we consider two names of different messages to be equal if they are identical or if the first name is the super type of the second name or vice versa.

This has an impact on the construction of the intersection automaton. If we consider two names n_1 and n_2 to be equal if n_1 is a super type of n_2 or if n_2 is a super type of n_1, we need to specify the resulting label in the intersection automaton. Figure 31 gives an example.

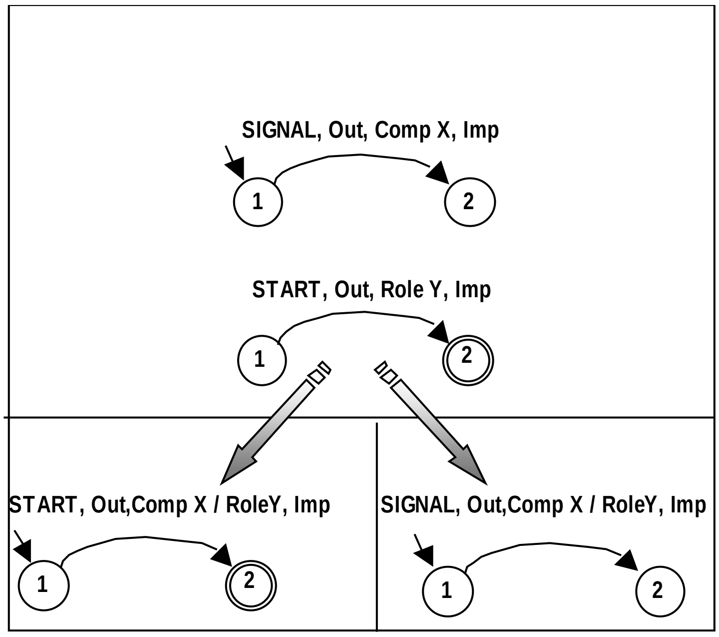

SIGNAL, Out, Comp X, Imp
1
2
START, Out, Role Y, Imp
1
2
START, Out,Comp X / RoleY, Imp
1
2
SIGNAL, Out,Comp X / RoleY, Imp
1
2

Figure 31: Combining two equal messages with hierarchy

We take the subtype as the new label because the subtype constrains the super type.

4.11 Non-Determinism

As we explain further in this text we use the resulting automaton of the global check as the intermediate glue code that is generated between a set of components. It can be argued that there is no problem if this glue code behaves in a non-deterministic way (in CSP for example non-determinism is built-in). However, as we show in the following, non-determinism is sometimes introduced as a side effect resulting in unexpected non-deterministic behavior. In this section, we handle these cases in a bit more detail.

4.11.1 When Does Non-Determinism Occur?

There are two possibilities to introduce non-determinism. The first one is by having the same implementation mapping for different primitives. Figure 32 shows a small part of the resulting automaton that will be used during the code generation. This automaton is deterministic at the level of the primitives but it renders a non-deterministic behavior when implemented. This automaton means that if component C2 sends an "actionPerformed" event it can either call the "Init()" on component C1 or the "Quit()" method on component C3. The only basis we have to make the decision is the primitive. However, this primitive is only part of the documentation. At runtime, a component only sends the "actionPerformed" event and we no longer know whether this was meant to be START or STOP. This is what we call *implementation non-determinism.*

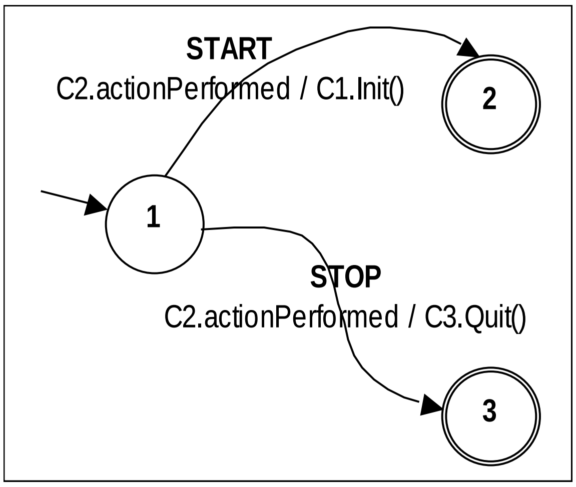

START
C2.actionPerformed / C1.Init()
2
1
STOP
C2.actionPerformed / C3.Quit()
3

Figure 32: Implementation non-determinism

The other possibility to introduce non-determinism is in the documentation itself. It is possible to draw non-deterministic scenarios using the MSC syntax. Figure 33 gives an example of such non-deterministic documentation.

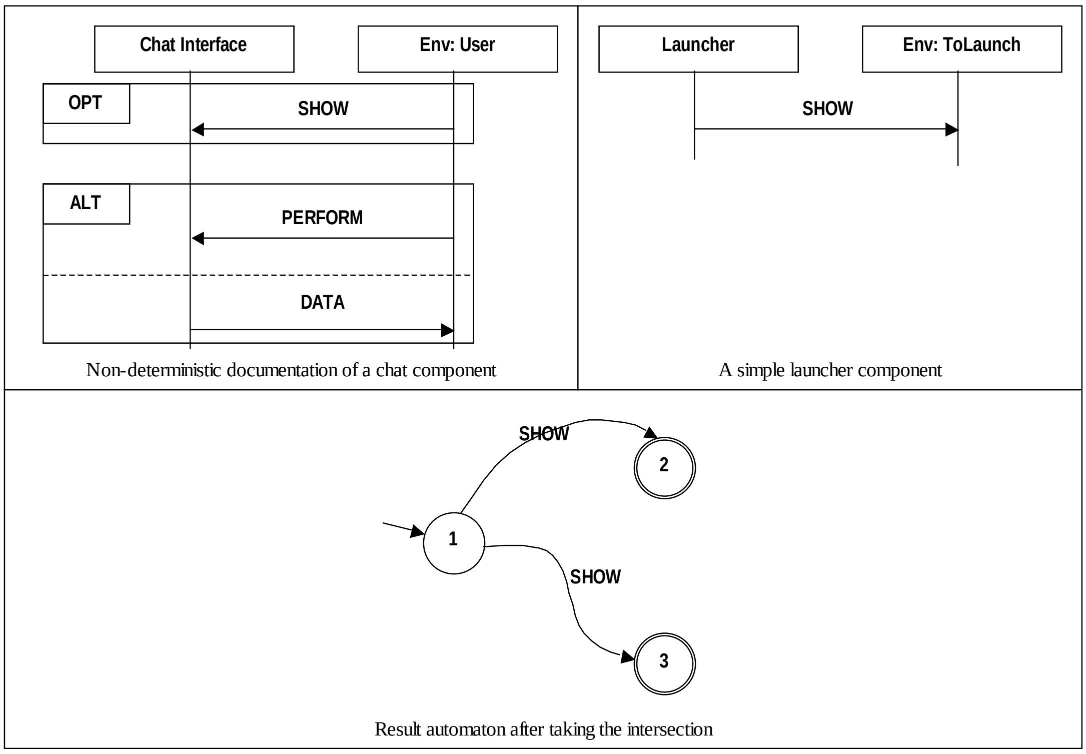

Chat Interface
Env: User
OPT
SHOW
ALT
PERFORM
DATA
Non-deterministic documentation of a chat component
Launcher
Env: ToLaunch
SHOW
A simple launcher component
SHOW
2
1
SHOW
3
Result automaton after taking the intersection

Figure 33:Example of scenario non-determinism

As the first SHOW message of the chat interface component (upper left corner in Figure 33) is optional it supports both a SHOW and a PERFORM message as it first message. Now suppose that this component is combined with a simple component that only sends a SHOW message (upper right corner in Figure 33). As SHOW is a subtype of PERFORM, the SHOW of this component will be matched with both the SHOW and the PERFORM message of the chat interface component. We explained higher that if a subtype is matched with a super type we label the resulting transition with the more specific one. This results in the automaton depicted in the lower part of Figure 33. This automaton is clearly non-deterministic. This is in fact not surprising as the documentation of the chat interface component was non-deterministic to begin with. However, the example also indicates that it is not always easy to recognize these situations. We call this kind of non-determinism *scenario non-determinism*.

4.11.2 How to Treat Implementation Non-Determinism?

Before we go into further detail on how to treat implementation non-determinism, we take a closer look at the cause of this non-determinism. We consider components to be black box entities. I.e. we do not have an understanding on their inner working. We have no way to predict what choice will be taken based on the documentation alone. Therefore, any choice is non-deterministic for us. Figure 34 shows a component that sends either a

START message or a STOP message. These messages are mapped on the same output event "actionPerformed" (this seems strange but this is the way one would typically indicate that the same event could result in either a START of another component or a STOP). In that sense an ALT block in our MSC documentation, behaves just as the CSP guarded command.

Thus, if this component sends the "actionPerformed" event it is unclear if this was meant to be a "START" or a "STOP" message. One possibility would be to add an adapter to every component that labels every output event of a component with the right primitive message based on the state of the component. However, once a component reaches an alternative block where all alternatives have the same implementation mapping; there is no way to know for the component what alternative corresponds to the output event sent by the component.

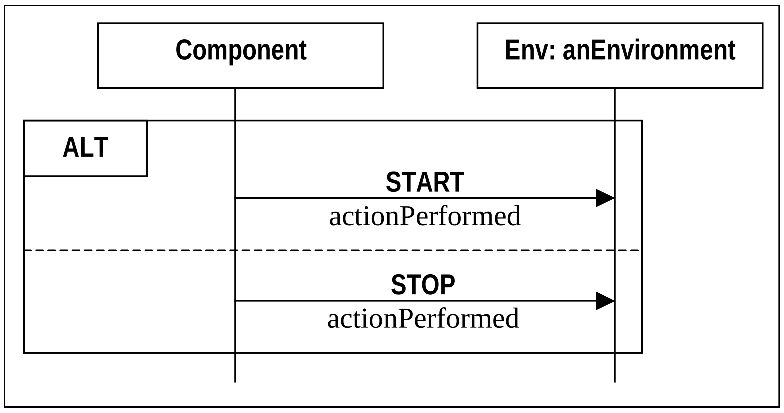


Figure 34: In black box components, ALT blocks represent non-deterministic behavior

This leaves very little room to handle these situations. We have no automatic means to make the decision.

One could also argue that this kind of documentation is wrong. As indicated higher this documentation is how one typically indicates that the same event could be mapped on several primitives. This is an indication that the hierarchy of the primitives is not well defined. It should be possible to find a super type that combines these primitives. This would solve the problem. In case we do not want to alter the documentation, we can only resort to user intervention. As these situations are easy to detect (the generated code contains a state with different actions based on the same event), we can do this just before the code for these states is generated.

4.11.3 How to Treat Scenario Non-Determinism?

Scenario non-determinism can occur when a component contains a state with several alternatives that have a common super-type. This means that there is a chance on this kind of non-determinism in every alternative as all primitives are a sub-type of the SIGNAL primitive. This does not mean that this is a very common situation. If the alternatives that would result in non-determinism deal with different environments, there is no problem. As we check that any environment is mapped on exactly one role (see 4.9.1), we know that two different environment will be matched on two different components. In that case adding the source of the message makes the state-machine deterministic again.

To check this at runtime we need to be able to check the source of an event. This is easily done by adding adapters to every component that adds a pointer to the source of every event (or when we have access to the components, by introducing a super-class for all output events that supports a "getSource()" method as it is done in for example in Visual Age for Java (IBM)). Our prototype tool uses the wrapper solution.

Nevertheless, we have no automatic solution if two messages come from the same component with one message a super type of the other. In that case, we require user intervention.

5 Mismatch Feedback

“Feedback is the breakfast of champions. “

- Kenneth Blanchard - In the Ultimate Success Quotations Library, 1997

5.1 Introduction

5.1.1 Problem Statement

What if a compatibility check fails? The idea of this work was to guide the developer as much as possible during the composition process. Therefore we want to give feedback on where the mismatch occurred and if possible how to cure it. There are two main problems here.

The first problem is to define the kind of feedback we want. It can be argued that in the most general case it makes no sense to provide mismatch feedback except for a simple warning “total mismatch”. Suppose we try to match a component with a role in the composition pattern that does not fit at all. Do we really want feedback saying that we need to adapt the component such and such to make it work or do we want the algorithm to come back with an error saying that this component is not compatible with the selected role? And if so where do we draw the line? In Figure 35 we give an example of such a mismatch.

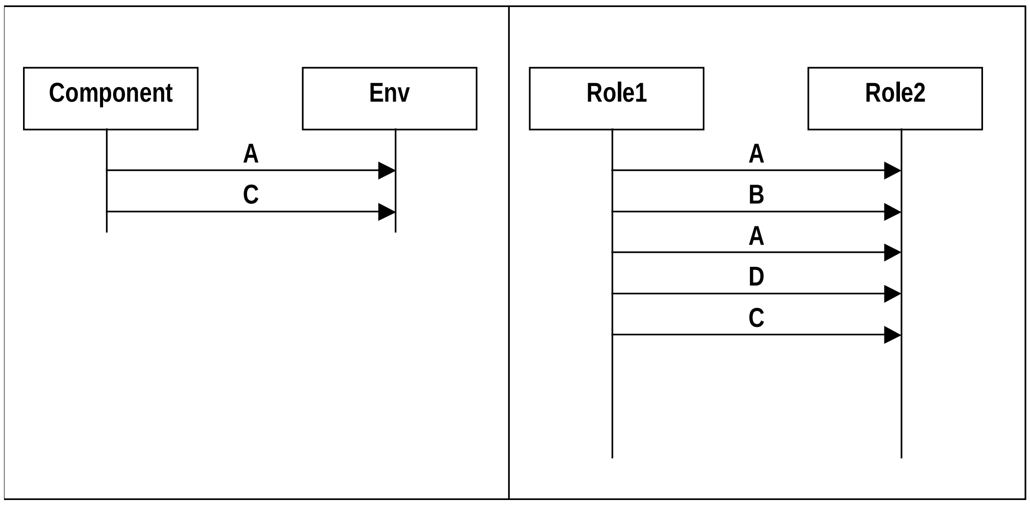


Figure 35:Indicating the mismatch?

How do we decide in this example if the message “A” of the component matches with the first or the second “A” of the composition pattern? Moreover, suppose it matches with the first “A”. Do we really want the program to show that everything between this first “A” and the message “C” is incompatible? Do we want feedback that is centered on the component or rather around the composition pattern? I.e. do want to adapt the component to the composition pattern or the composition pattern to the component? It is clear that there are no straight answers to this kind of questions.

The second problem is the conversion of MSC's to automata. A mismatch is detected during operations on automata. The documentation is provided in the form of MSC's. It seems reasonable to provide the mismatch information as annotations on MSC's. This involves a conversion from automata to MSC's. As an MSC corresponds to a regular expression, we can use the well-known algorithm based on the elimination of states [Hopcroft, 2001]. However, the result of this conversion is not always clear. In general, it turns out to be difficult to see the link between the generated MSC's and the original MSC's used in the compatibility check.

5.1.2 Approach

Therefore, we take the pragmatic approach. We provide a number of tools that can be used by the developer to find out what is going wrong when the algorithms detect a mismatch and to help the developer to find a solution. We developed two classes of tool support. The first class consists of tools to indicate the mismatch. More precisely, we annotate on the component MSC how far the scenario matches with the composition pattern and this both starting from the first message as starting from the last message. We just present the first possible match (i.e. we do not try to skip messages to find better matches). The tool is used as a feedback tool only and needs human interpretation to judge the results. We also developed a similar tool to annotate on the composition pattern how far it matches with the set of components used in this composition pattern, again both starting from the first message as well as starting with the last message.

By far the more interesting approach is the generation of adapters. Adapters are automata that can be used to adapt the protocol specified by one automaton to the protocol specified by another. There exist a whole field of research about adapter generation for finite automata [Schmidt, 2000;Yellin, 1994a;Zaremski, 1997]. Building on this research, we propose two different solutions for the problem. First, we describe the results of Reussner as his solution is prototypical for the field and because the asymmetric cross product is used to calculate the result, which indicates a similarity to our approach to compatibility checking.

Next, we introduce the traversal strategies research done by Lieberherr and co. [Lieberherr, 1997]. The latter has nothing to do with adapter generation. The goal of this work is to check a given class graph (i.e. a dependency graph of classes and objects for a given application) against a traversal specification. A traversal specification describes a path in the class graph possibly with more intermediate classes or objects. We noticed that the proposed algorithm to check a traversal specification (called a strategy) against a class graph has many similarities with the algorithm proposed by Reussner. A closer look reveals that the traversal strategy algorithm can be used as a more flexible and more efficient adapter generator than the algorithm proposed by Reussner. We show how this can be achieved.

5.2 Annotating Compatibility on MSC's

A mismatch is detected in an automaton. We would like to show this mismatch on the original MSC's. The most obvious approach is to trace the automaton and convert this automaton back in a MSC. However, this conversion is difficult and ill defined. There are many ways to convert one automaton to a MSC and it could be very hard to see that the generated MSC is in fact the same MSC you started form.

5.2.1 From Automata to MSC's

To cope with the automata-MSC conversion problem, we skip the conversion and we simply maintain the link between a transition in the automaton and the original message in the component or composition scenario. As we are calculating cross products and perform all kind of other operations, we end up with one transition that is linked to a number of messages (for example one transition in the result is linked to at least the message in the component and its corresponding message in the composition pattern). We implemented this for all operations on our automata. A simple traversal over the automaton now allows us to show all messages that are reachable from the start state and all states from where the stop state can be reached. Mind that we need to calculate a *full* cross product to indicate states and transition reachable from a stop state. In our optimized algorithms we calculate the cross product starting with the two start states until we are stuck. This way we never generate other traces. Therefore, we regenerate the cross product with a non-optimized algorithm (add the cross product of states and add transitions for every state in this cross product) before we start tracing the result. The whole process is depicted in Figure 36.

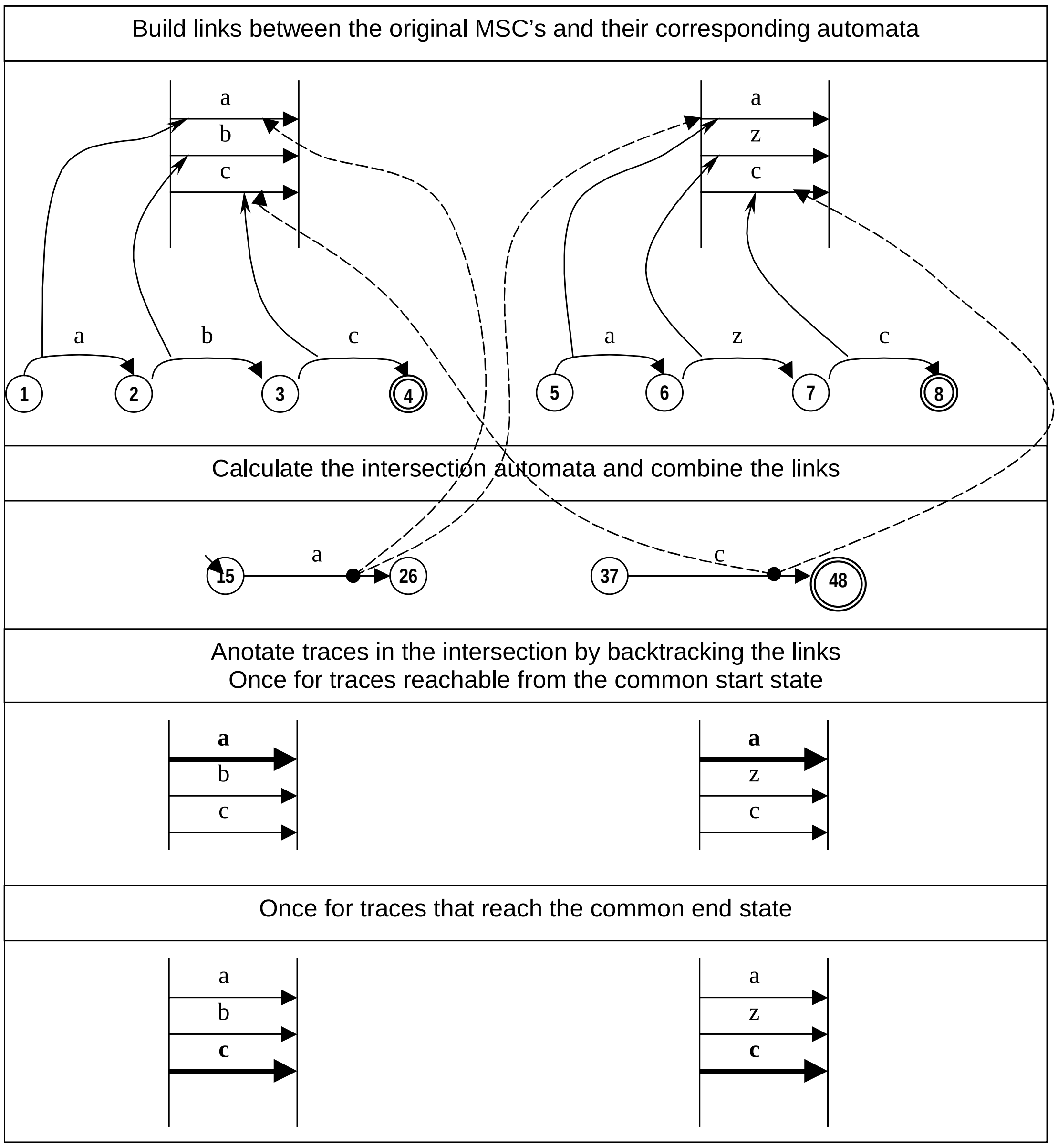

Build links between the original MSC's and their corresponding automata
a
b
c
a
z
c
a
b
c
a
z
c
1
2
3
4
5
6
7
8
Calculate the intersection automata and combine the links
a
15
26
c
37
48
Anotate traces in the intersection by backtracking the links
Once for traces reachable from the common start state
a
b
c
a
z
c
Once for traces that reach the common end state
a
b
c
a
z
c

Figure 36: Feedback process by anotating MSC's

5.2.2 Discussion

The disadvantage of this approach is that the user has to distinguish him or herself between real incompatible traces and traces that are "optional". I.e. only compatible traces are marked, but as we calculate the intersection, this does not mean that all non-marked traces are incompatible. In Figure 37 we try to map the component on the left hand side on role1 of the composition pattern.

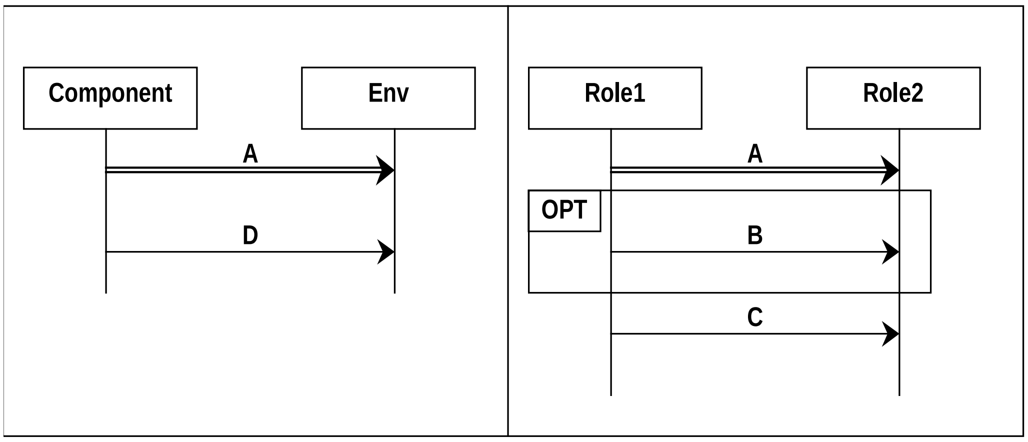


Figure 37: Marking incompatible traces?

As this component is not compatible with this role, this generates a mismatch. The mismatch feedback described above now marks all traces that are compatible. In this case, only the message “A” is compatible (indicated by the double line).

However, it could be argued that the optional part (with the message “B”) in the composition pattern is also compatible. Changing anything inside the optional part will not make the composition pattern compatible or incompatible. If we remove message “D” in the component scenario and message “C” in the composition scenario, we have two compatible scenarios, but the same check of compatible traces will still only highlight the message “A”.

5.3 Adapter Generation

5.3.1 Reussner Adapter Generation

5.3.1.1 Introduction

In this section, we discuss the algorithm introduced by Reussner. We notice that the asymmetric cross product (see section 4.7.2) is used to calculate: "a changing protocol adapter". In [Reussner, 1999] Reussner presents the example depicted in Figure 38.

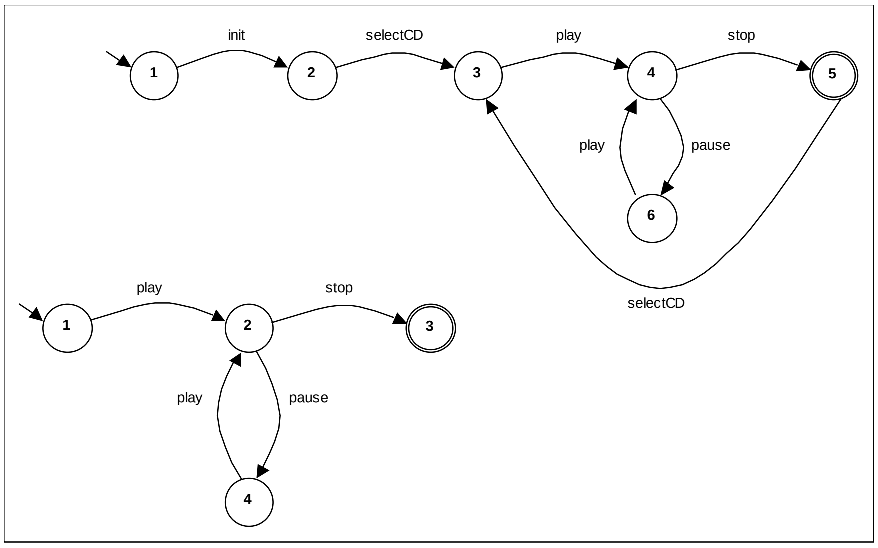


Figure 38: Adapting a simple CD player.

This example describes the usage behavior of the GUI interface of two Compact Disc players. The first player is a more sophisticated player supporting multiple disc play, while the second player describes a standard CD player allowing only one CD to be played. Now suppose our composition pattern specifies the behavior of the multiple disc player and our GUI component specifies a usage scenario corresponding with a single disc player. It is clear that we need to prefix the simple component usage scenario with the "init" and "selectCD" transition to make it work. Reussner explains how he uses the asymmetric cross product to generate these prefixes.

5.3.1.2 Adapter Generation

The main step to compute these adaptations is to create the asymmetric cross product automaton. This algorithm starts with one "master" automaton and one "slave" automaton that will be adapted to the "master" automaton (in our case one "slave" component automaton that needs to be adapted to the "master" composition automaton). The set of states in the resulting automaton is a subset of the Cartesian product of the states of the "master" automaton and the states of the "slave" automaton. The general idea is that the result contains two kinds of transitions: marked and unmarked transitions.

Marked transitions go from a state pair (*sm*, *sl*) containing one state from the "master" and one state from the "slave" with an input "i", where in the "main" the input "i" is handled in state *sm* and the "slave" automaton handles input "i" in state *sl*.

In an unmarked transition the input "i" is only handled in state *sm* (i.e. in the main automaton)

This algorithm is asymmetric in nature. Transitions with labels accepted in the "master" but not in the "slave" are added to the result automation, while transition with labels accepted in the "slave" but not in the "master" are not added to the result automaton. Figure 39 shows the result for the CD example. For more details on this process see [Schmidt, 2000].

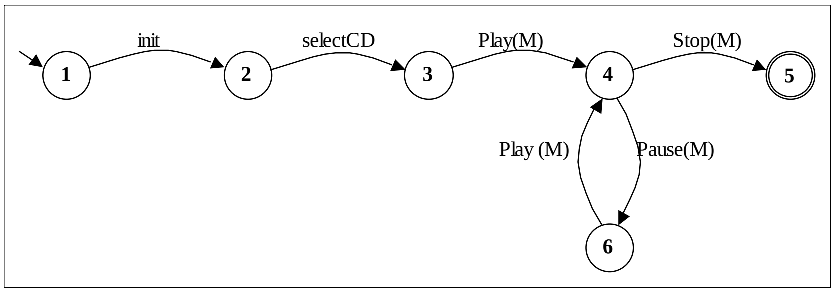


Figure 39: The generated adapter. Transitions that are not marked with (M) are needed adaptations.

This algorithm thus generates a solution on how to adapt a component to a role in the composition pattern. Using the component automaton as "master" and the composition automaton as "slave" on the other hand renders a suggestion on how to adapt the composition pattern to the component. This does not mean that this algorithm is only useful to generate an adapter between one component and a composition role. We also use this adapter generation technique to generate an adapter for the global case. To do this we consider the composition automaton as the "master" automaton and the automaton resulting from the parallel composition of the components as the "slave" automaton. This way an adapter is calculated for a given set of components against a given composition pattern.

As a last step, we need to convert the resulting adapter back to MSC's. We face again all the problems mentioned in section 5.2.1. Therefore, we go for the same solution. I.e. rather than converting the resulting adapter automaton back to a MSC, we keep track of the links with the original automata. Simulating the generated adapter on the MSC's shows when messages of the composition pattern need to be inserted for given scenarios or vice versa.

5.3.1.3 Conclusion

This algorithm comes up with only one solution. It favors early matches over later matches (in Figure 35 this solutions assumes that the component matches with the first A in the composition pattern) and adapts one party to the other instead of adapting both parties. However, it gives a good indication on what is missing to make them compatible and is thus useful as a feedback mechanism.

5.3.2 The Adaptive Programming Library

5.3.2.1 Introduction

In this section, we show the connection between the traversal strategies research described in [Lieberherr, 1997] and adapter generation. The following example is used in [Lieberherr, 1997].

Consider the class graph depicted in Figure 40, which defines a data structure describing a bus route. A bus route object consists of two lists: a list of bus objects, each containing a list of passengers; and a list of bus stop objects, each containing a list of people waiting. Suppose that as a part of the simulation, we would like to determine the set of person objects corresponding to people waiting at any bus stop on a given bus route. The group of collaborating classes that is needed for this task is shaded in Figure 40. To carry out the simulation, an object-oriented program should contain a method for each of these shaded classes.

The idea of traversal strategies is that this could be solved in a much more elegant way. Below are two possible traversal specifications (called traversal strategies) that choose the desired set of classes:

(1) from BusRoute through BusStop to Person;

(2) from BusRoute bypassing Bus to Person;

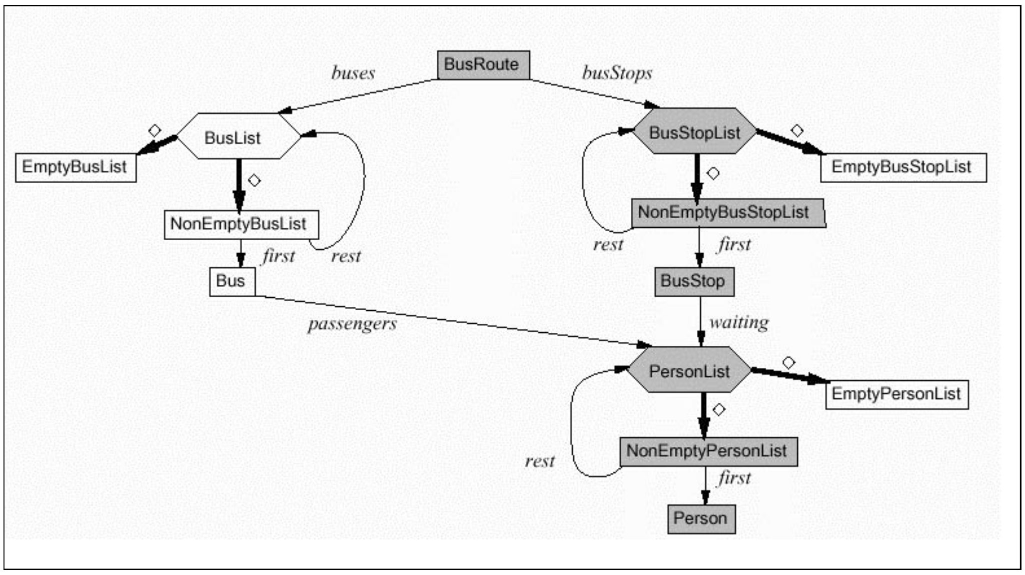


Figure 40: Bus simulation class graph. Squares and hexagons denote classes (concrete and abstract, respectively), regular arrows denote fields and are labeled by the field name, and bold arrows (labeled with ◊) denote the subclass relation (for the shading, see text)

Suppose now that the bus route class has been modified so that the bus stops are grouped by villages. The revised class graph is depicted in Figure 45. To implement the same requirement of finding all people waiting for a bus, an object-oriented program must now contain one method for each of the classes shaded in Figure 45, and thus the previous object-oriented implementation becomes invalid. The traversal strategies (1) and (2), however, are up-to-date and do not require any rewriting.

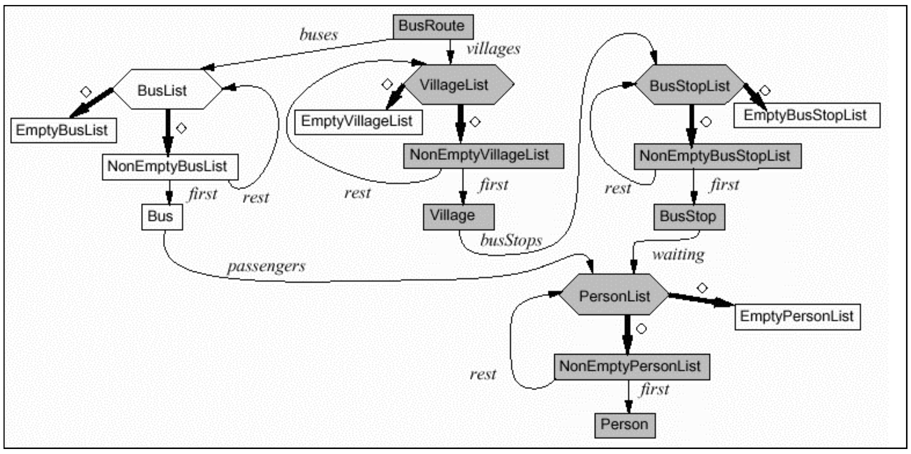


Figure 41: Evolved bus simulation class graph.

The basic idea of traversal strategies is that under a name map N (mapping concepts of the traversal strategy on concepts in the class graph), a path in the strategy graph is an abstraction of a set of paths in the class graph. This is done by viewing each strategy-graph edge a → b as representing the set of paths in the class graph starting with node N(a) and ending at node *N*(b). An example of such a strategy graph is depicted in Figure 42. To be complete every edge in the strategy graph can have a set of traversal constraints attached to it. These are typically used to express bypassing constraints. I.e. if we need to specify that we go from BusRoute to Person *bypassing* BusStop we construct a strategy graph with two nodes BusRoute and Person and an edge going from BusRoute to Person with an attached constraint that BusStop should be bypassed. For the formal definition of these constraints and strategy graphs the user is again referred to [Lieberherr, 1997].

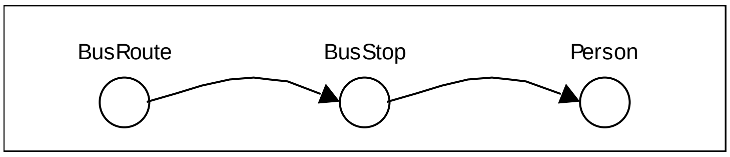


Figure 42: Strategy graph corresponding with the "From Busroute through BusStop to Person" strategy.

Their work presents an algorithm to construct a traversal graph from a given strategy graph and a given class graph. This algorithm is called algorithm 1 in their paper [Lieberherr, 1997]. This graph is in fact an expanded version of the strategy graph using nodes from the class graph such that the resulting graph contains only paths that comply with the given traversal strategy.

In the next section, we explain in a bit more detail the inner workings of this traversal graph construction algorithm.

5.3.2.2 The Traversal Graph Algorithm

In this section, we give an informal definition of the algorithm using a running example used in [Lieberherr, 1997]. For the formal definition of the algorithm, we refer to the same paper. Suppose we want to calculate the traversal graph for the class graph and the strategy graph depicted in Figure 43.

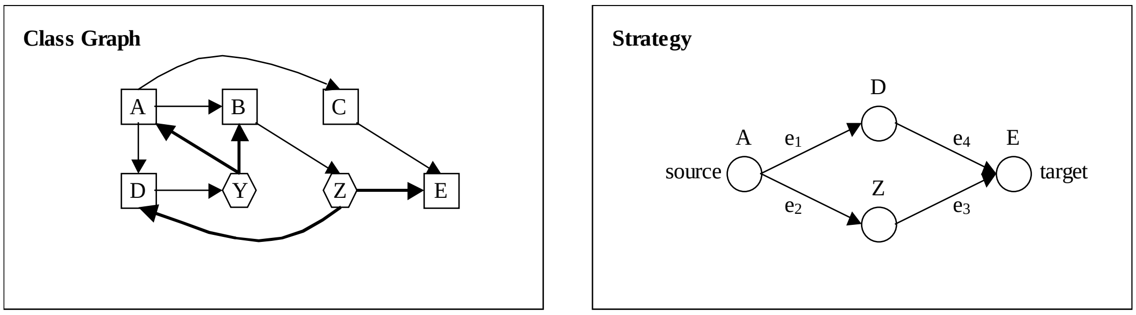


Figure 43: Example class graph and strategy graph used for the calculation of a traversal graph. In this example e_1 and e_2 have no attached constraints, the constraint attached to e_3 specifies that the edge going from A to D should be bypassed and the constraint attached to e_4 specifies that A and all incident edges to A should by bypassed.

The general idea is to expand the strategy graph by inserting a copy of the full class graph for every edge in the strategy graph. Informally the algorithm proceeds as follows:

1. Copy the class graph as many times as there exist edges in the strategy graph
2. Apply any constraints attached to an edge in the strategy graph to the corresponding class graph. I.e. in the copy of the class graph corresponding to e_4 in the example, remove class A and all incident edges (the result of step 1 and 2 is depicted in Figure 44 square 1).
3. Connect these copies as specified by the strategy graph. In the example, edges e_1 and e_2 are connected by node D. In this step we connect the copy corresponding to e1 with the copy corresponding with e_2 by inserting an edge from class D in the first copy to class D in the second copy (the result this step is depicted in Figure 44 square 2). As this results in class D showing up twice in the traversal of the connected graph we replace this new interconnection edge with a set of edges connecting all classes that are directly connected to D in the first copy with class D in the second copy (the result this step is depicted in Figure 44 square 3).
4. Finally mark all classes in the class graph copies corresponding with edges in the strategy graph that are connected with the initial node as start classes and mark all classes in the class graph copies corresponding with edges in the strategy graph that are connected with the final node as final classes. (the result of this step is depicted in Figure 44 square 4, where the little arrows indicate a start class and the doubled bordered classes indicate final classes).

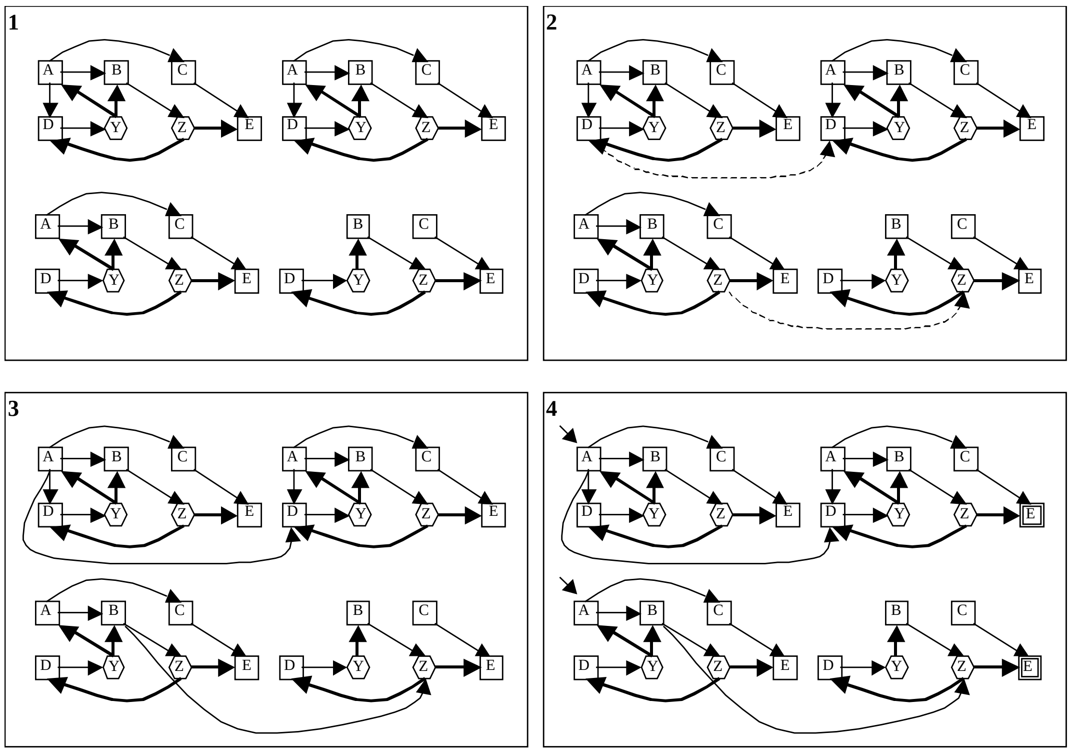


Figure 44: Applying the traversal graph algorithm on the example in the previous picture

5.3.2.3 Generating Adapters

In our research, the local check verifies if a given component C can be used to implement a given role R. Both the component and the role are documented with an automaton that represents the state transitions of the component and the role. The component is compatible with the role if the component and the composition pattern share at least one trace from a start state to a stop state. Typical mismatches occur when the component does have such a trace except that it sends one or more message in between (so it has one or more transitions that are not found in the role description, but it does have all the transitions as specified by the role). The traversal graph algorithm called algorithm 1 in [Lieberherr, 1997] recognizes all these situations and builds an automaton describing all possible adaptations for the component so that it becomes compatible with the role or renders an empty automaton if it cannot be done. To do this we consider the component automaton as the strategy graph and the role automaton as the class graph. The traversal graph is empty if there is no adaptation possible to make the component compatible. Otherwise, the traversal graph contains all possible adaptations. To fully understand the analogy we need to go into more detail on the connection between NDFA intersection and the calculation of a traversal graph.

5.3.2.4 *Connection Calculation Traversal Graph with the Intersection of NDFA's*

During a discussion at ICSE 2000, Prof. Dr. Karl Lieberherr mentioned the connection between the calculation of the traversal graph and the intersection of two NDFA's. A slide show explaining this connection can be found at [Lieberherr, 2001].

The idea behind calculating the traversal graph is to check if a start-stop path specified by the strategy graph also exists in the class graph, allowing the class graph to use more internal transitions. I.e. a strategy specifies where to start, where to stop and what transitions should certainly be passed going from start to stop. This corresponds to traversal specifications following the template: FROM x_1 VIA $x_2 \ldots x_{n-1}$ TO x_n. Note that this is a restriction of the general traversal specification as defined by Lieberherr et al. They also allow specifying what transitions are *not* allowed. For now, we stick to this restriction for simplicity reasons. It is easier to see the connection with the intersection of NDFA's with this restriction. We come back to this later because adding constraints as to what transitions are not allowed makes it possible to constructs adaptation views (see section 5.3.4.4 for details).

This analogy indicates that after each transition as specified by the strategy graph, we can have any number and any kind of transitions in the class graph as long as we go on with the transitions as specified by the strategy. An example makes this clear. Take the class graph and the strategy as specified in the left hand side of Figure 45 (and assume the name map to be identity). The strategy graph means: Traverse FROM a VIA c TO e. It is clear that the class graph supports this strategy. The resulting traversal graph is identical with the class graph.

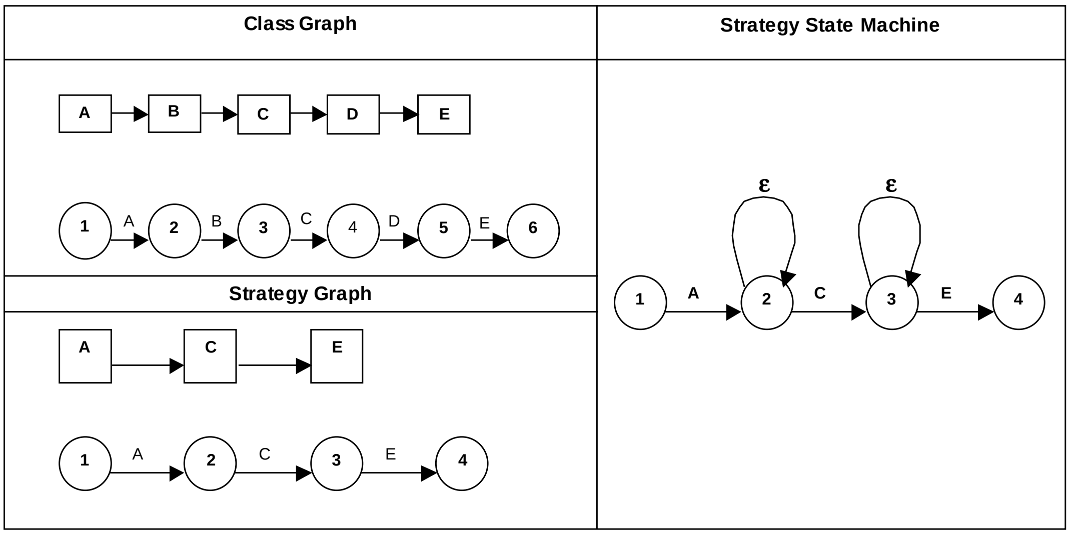


Figure 45: Class Graph and Strategy Graph with Corresponding Automaton

To illustrate the connection with NDFA intersection the corresponding automaton for the class graph and the strategy graph is shown just below them in Figure 45 (the correspondence is easy to see if you compare the strategy automaton with the specification FROM a VIA c TO e). Calculating the intersection between the class graph automaton and the strategy graph automaton as specified on the left hand side of Figure 45 returns an empty automaton. Remind now that the strategy graph allows more internal transitions in the class graph. This means that during the calculation of the intersection we should be able to proceed at wish in the class graph automaton until we find a common transition again. This is easily accomplished by adding epsilon transitions loops after every internal transition in the strategy graph automaton. The result is depicted at the right hand side of Figure 45.

It is important to note here that the traversal graph calculation algorithm does not perform a general intersection of two NDFA's. The correspondence works the other way round. I.e. it is possible to translate the traversal graph and the class graph (for the restricted class of strategies defined above) to NDFA's where the intersection of these NDFA's corresponds with the traversal graph obtained by the algorithm as defined in the Adaptive Programming library. It is in general not proven that two NDFA's can be converted to a class graph and a strategy graph so that their traversal graph corresponds with the intersection of these NDFA's.

5.3.2.5 Connection Calculation Traversal Graph with Parallel Composition

A closer look at the special kind of NDFA intersection introduced above reveals that we obtain the same result using the parallel composition operator. To see this we first define this special kind of NDFA intersection formally:

Definition: Asymmetric NDFA Intersection

An asymmetric NDFA intersection of two component or composition automata $P_1=(S_1, q_1, F_1, succ_1, \Sigma_1)$ and $P_2=(S_2, q_2, F_2, succ_2, \Sigma_2)$ is described as:

$$P_1 \cap_A P_2=(S_1 \times S_2, [q_1, q_2], F_1 \times F_1, succ', \Sigma_1 \cup \Sigma_2)$$

where

- $succ'([s_1,s_2],\alpha) = [t_1,t_2] \Leftrightarrow \alpha \in \Sigma_1 \cap \Sigma_2$ and $succ_1(s_1, \alpha) = t_1$ and $succ_2(s_2, \alpha) = t_2$
- $succ'([s_1,s_2],\alpha) = [t_1,s_2] \Leftrightarrow \alpha \in (\Sigma_1 \setminus \Sigma_2)$ and $succ_1(s_1, \alpha) = t_1$

A quick comparison between this definition and the definition of the parallel composition operator shows that the only difference between them lies in the definition of the transition function. The asymmetric NDFA intersection defines a result for transitions labeled with a label that is part of the alphabet of the first automaton only and for joint steps. The parallel composition operator additionally defines a result for transitions labeled with a label that is part of the alphabet of the second automaton only. I.e. the parallel composition operator allows both automata to proceed for transitions that are labeled with a label that is not part of the other's automaton alphabet, while the asymmetric NDFA intersection only allows this for one of the automata.

Thus to get the same result from the parallel composition operator we need to prevent that there exist labels that are part of the alphabet of the second automaton and that are not part of the alphabet of the first automaton.

This is easily accomplished by extending the alphabet of the first automaton with the alphabet of the second automaton.

All this means that the traversal graph can also be calculated using the parallel composition operator (using the conversion of class graphs and strategy graphs as explained in section 5.3.4.1).

Using the Parallel Composition to Calculate Traversal Graphs

Let CGA be the automaton corresponding to a given class graph CG

Let SGA be the automaton corresponding to a given strategy graph SG

Then $TG = CGA_{+\alpha_{SGA}} \parallel SGA$ is an automaton that has the same start-stop paths as the traversal graph resulting from the traversal graph calculation for the class graph CG and the strategy graph SG as specified in [Lieberherr, 1997]

5.3.3 Link Adapter Generation and Adaptive Programming

We state that the asymmetric cross product used by Reussner and the Traversal Graph Generation Algorithm as defined by Lieberherr are closely related. In the previous we indicate that extending the strategy graph with epsilon transition loops in every state and performing a standard NDFA intersection between the thus extended strategy graph with the class graph has the same result as applying the traversal graph algorithm directly to the traversal graph and the class graph. Yannis Smaragdakis first indicated this link. We also indicated that the asymmetric cross product used by Reussner to generate what he calls "a changing protocol adapter" could also be obtained by performing a standard NDFA intersection where the "slave" automaton is extended in the same way as the strategy graph in the traversal graph algorithm. I.e. by adding epsilon transition loops in every state. As the result of both algorithms can be obtained by the same process (i.e. adding transitions and calculating the NFA intersection) we feel that both algorithms are essentially the same, although used for very different purposes. Hence, our intuition that the traversal graph algorithm can be used as an adapter generator.

The next section explains in detail how we use the traversal graph generation algorithm to generate adapters. This result is not formally proven, but at least the new algorithm works for all the examples we tried.

5.3.4 Calculating Adapters

It is clear from the previous that the traversal graph algorithm calculates the intersection between a specific subset of NDFA's. So, what does it mean if we just convert the role automaton to a strategy graph and the component specification to a class graph? The traversal graph algorithm first inserts ε-transition loops after every internal transition in the class graph automaton (now corresponding with the role specification). This allows the component to proceed until it finds a compatible transition in the role specification. The traversal graph shows how the role R1 can be traversed as specified by the component in Figure 46. Indeed, considering the role specification as the strategy means that we go from A via B to D. Our component specification allows this with an intermediate transition C.

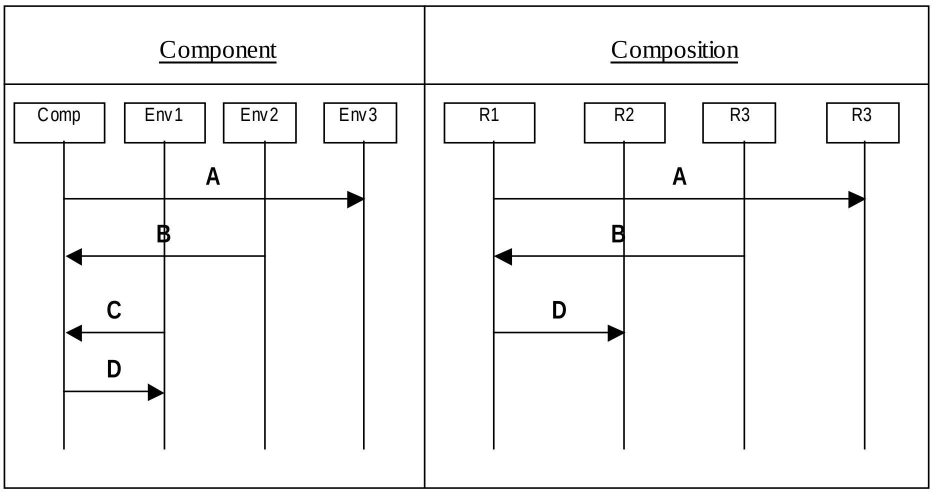


Figure 46: Applying the traversal graph algorithm to the component at the left hand side and role R1 of the composition pattern at the right hand side, results in a traversal graph specifying that the component fits with the role if message C is inserted between B and D

All this means that we only need to convert the automata of the component to a class graph, the role specification to a strategy graph and that the traversal graph algorithm then calculates a traversal graph that corresponds with a automaton describing all traces that renders the component to be compatible with the role. Any trace that is found in the traversal graph corresponds with a solution of how we can adapt the component to the role. Switching the inputs returns all possible adaptations of the role to the component.

To explain the technical details of the adapter generation using the traversal graph approach we start with a small example. In Figure 47 we show a component usage scenario and a composition pattern. It is clear that the intersection of these scenarios is empty. They both engage in A, but fail to proceed any further as the composition pattern expects B while the component only offers F or C.

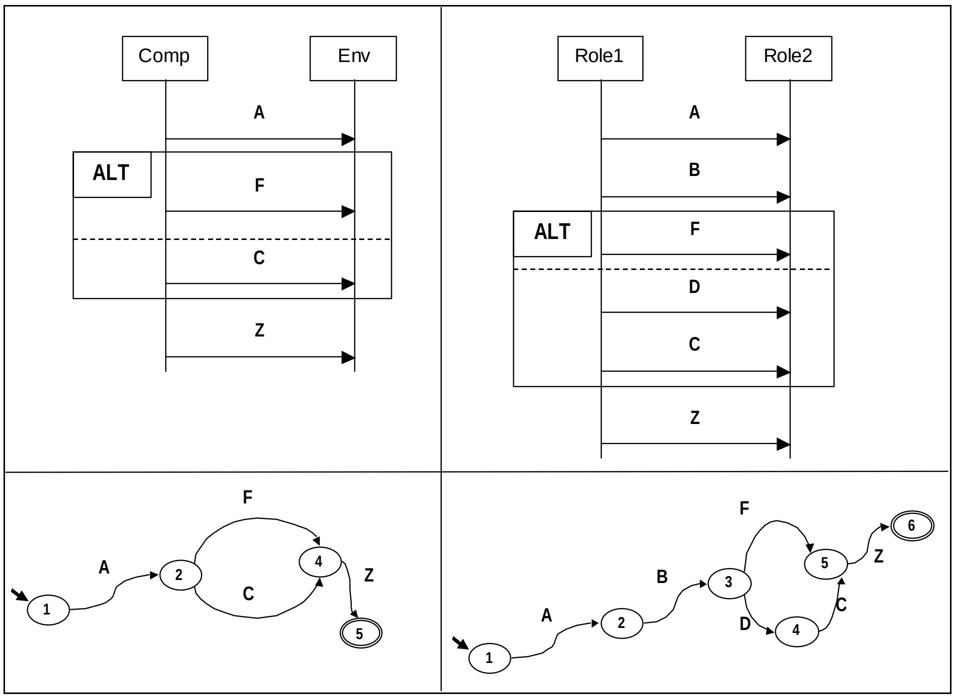


Figure 47: Example for Adapter Generation

Now we apply the traversal graph algorithm to generate an adapter for the component to make it compatible with the composition pattern.

5.3.4.1 Converting to a Class Graph and a Strategy Graph

To achieve this we convert the composition pattern to a class graph. We do this by constructing a class for every transition in the state diagram and adding dependency arcs for every incoming transition to every outgoing transition. State 3 of the composition pattern for example has an incoming transition B and outgoing transitions F and D. Therefore we add dependency arcs from the class B to class F and class D. The classes are labeled with the label of the transitions.

Next, we convert the component to a strategy graph. The process is exactly the same as the conversion to a class graph, except that we construct labeled nodes instead of classes.

The result of both conversions is depicted in Figure 48.

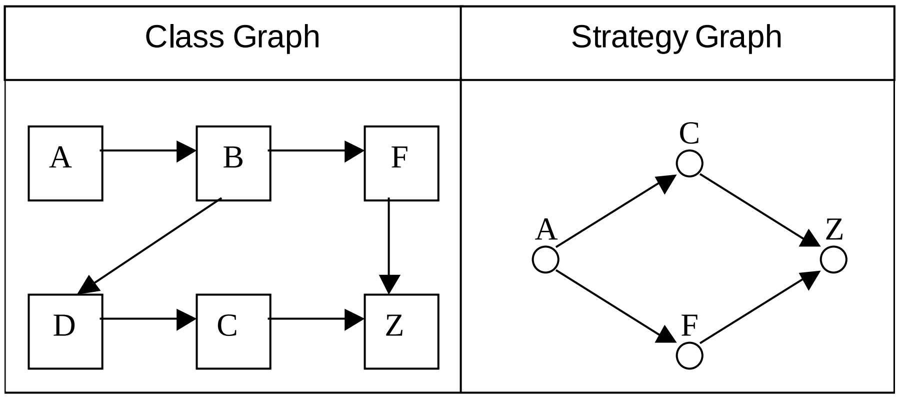


Figure 48: Converting the Component and Composition Pattern specified in Figure 47 to a Class Graph and a Strategy Graph

An alternative approach is to use the equivalence of NFA intersection and the traversal graph algorithm. To do this we add epsilon transitions to every state of the composition automaton and calculate the intersection of this new automaton and the component automaton. However, as an efficient implementation of the traversal graph algorithm is readily available in the AP library [Lieberherr, 1997] we stick to this version.

5.3.4.2 Marking the Traversal Graph

Now we calculate the traversal graph. We need to do something extra to distinguish the "adaptations". If we calculate the traversal graph directly, the result does not distinguish between "intermediate" classes in the class graph and the classes that correspond directly with the strategy (technically this correspondence is defined by the name map function in [Lieberherr, 1997]). We show that it is easy to adapt the traversal graph algorithm so that it marks the classes that correspond with the strategy.

Remind that the traversal graph algorithm "replaces" every arc in the strategy graph with a copy of the class graph. The idea behind this is that we can traverse any link in the class graph to proceed from the source to the destination of one as specified by the strategy graph. Cast in our terminology this means that we can traverse any message in the composition pattern between the messages specified by the source and the destination in the strategy graph. I.e. the source and the destination of the strategy graph are the "common" messages. All other messages traversed in the class graph are "adaptations". Therefore, we mark the classes that correspond with the source and the destination of the strategy graph in every copy of the class graph. The result is depicted in Figure 49. Here "marked" states can be recognized by their double border.

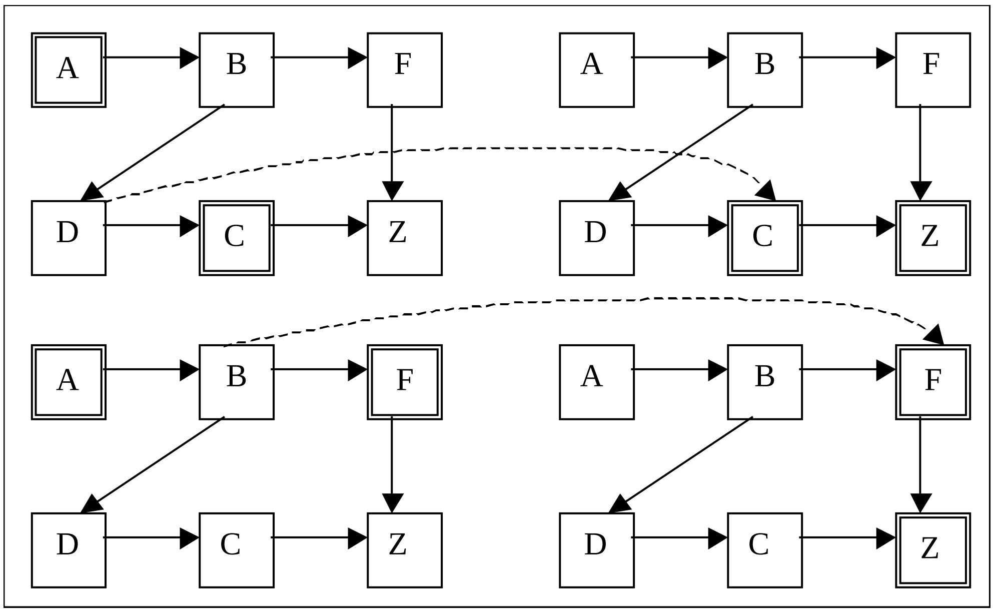


Figure 49: Marking the Traversal Graph for the example depicted Figure 47

5.3.4.3 Calculating the result

Proceeding with the algorithm as described in [Lieberherr, 1997] renders the result depicted in Figure 50.

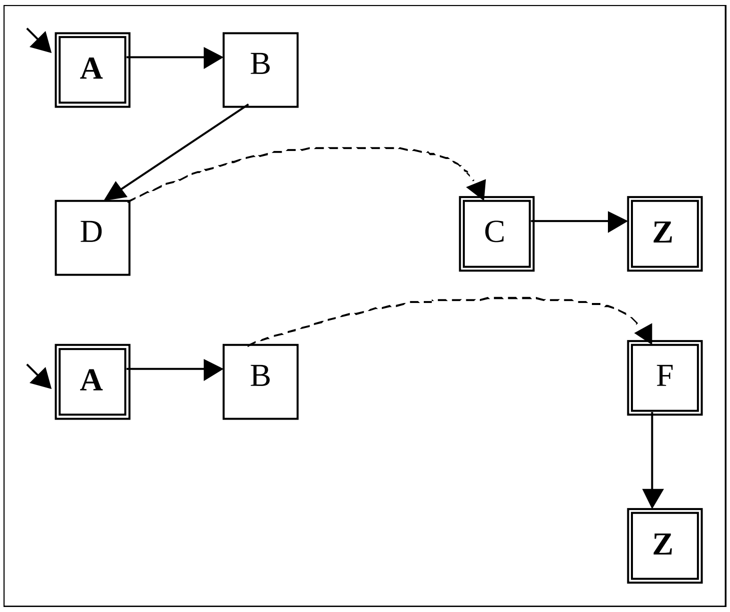


Figure 50: Resulting Traversal Graph for the example in Figure 47

Any traversal in this graph now gives a possible adaptation of the component to comply with the composition pattern. In this case we can add the message B and D between A and C in the component or we can add B between A and F.

5.3.4.4 Constraints as Adaptation Views

The traversal graph algorithm supports the concept of constraints. These constraints allow the specification of classes in the class graph that should be skipped or on the contrary classes that should be part of the traversal. This is specified with the keywords: "bypassing" and "via" in the traversal specification.

These constraints can be easily used to obtain a sub view of all adaptations by the specification of transitions that must be part of the solution or by specifying transitions that may not be used. Figure 51 shows a component and a composition pattern that are not compatible. If we want to adapt the component at the left hand side to role 1 specified by the composition pattern at the right hand side we have several possibilities.

The difference between the component scenario and the composition pattern are two notification messages: one after the "send" message and one after the "receive" message. To obtain a compatible component it suffices to provide only one of these notifications. Now suppose that the developer knows that he is going to use the component as a monitoring component and that this component never needs to send anything. Adding the constraint: " BYPASSING receive" (or in this case the equivalent phrase "VIA send" to the traversal graph algorithm, returns only the adaptations needed to use the component in "receive" mode.

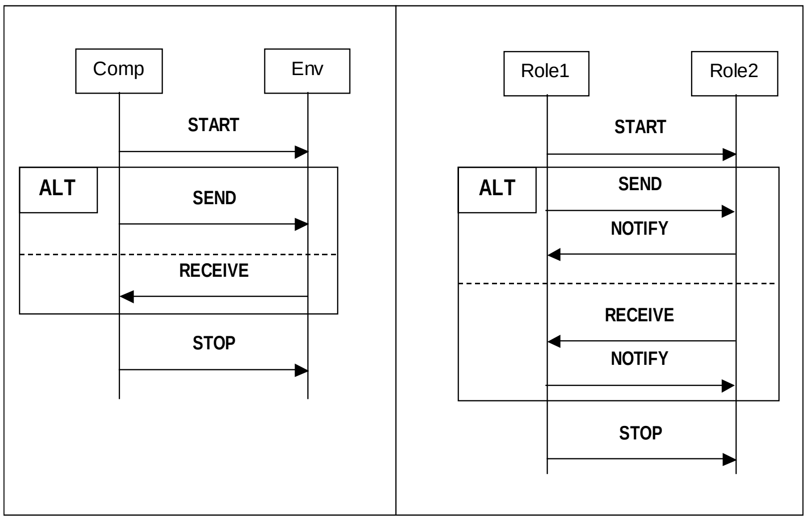


Figure 51: The usage scenario of the component at the left hand side is not compatible with role R1 of the composition pattern at the right hand side.

With the right kind of tool support, this can be done very naturally for the developer. One possibility is to run the adapter algorithm first without any constraints and present a list of messages that need to be added to the component to the developer. The developer then selects the messages he or she wants to add or vice versa the messages that he or she thinks are not needed for the application to work and we convert these in VIA and BYPASSING constraints. We then rerun the adapter algorithm to see if there exists a solution with these extra constraints.

To give an idea on how this works we look again at the correspondence with NDFA intersection. Remember that the component usage scenario is converted to an automaton where epsilon loops are added between every transition. During the intersection operation, these epsilon transitions are allowed to match with any transition specified by strategy graph (= the automaton corresponding with the composition pattern). Constraints are easily added by constraining this match. "BYPASSING" constraints are introduced by disallowing an epsilon transition to match with the specified transition. "VIA" constraints, on the other hand result in the specification of another node in the strategy graph. The interested reader is referred to [Lieberherr, 1997] for the details.

5.3.4.5 Handling Equal Named Messages Using Name Map

The traversal graph algorithm also offers a possibility to deal with one of the problems specified earlier. The problem is depicted again in Figure 52. Do we want the B message of the component to match with the first B in the composition pattern or with the second one? In this case, one could argue that it does not matter theoretically, but it often does matter when the implementation mapping is taken into account.

As we argued before this problem cannot be solved automatically. However the traversal graph algorithm allows the developer to specify which solution he prefers. This is done using the name map function.

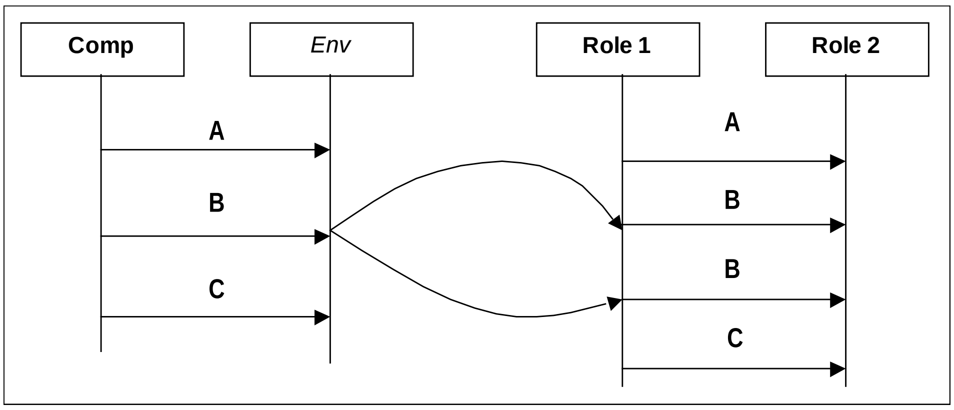


Figure 52: How to Adapt? Inserting the first B or the second?

The name map function maps names of the strategy graph (thus in our case the composition pattern) to names of the class graph (in our case the component usage scenario). The default value is identity. Of course, this assumes that every name in the class graph and the strategy graph is unique. Thus, in case a given transition occurs twice in the composition pattern or the usage scenario we give them a unique index. To specify the preferred solution we now specify a name map function that maps a transition to one of these indexed transitions. This way the traversal graph algorithm constructs the preferred adapters as it assumes that transitions with another index are totally different from the transition we are matching with. In terms of the traversal graph algorithm the name map function is used to construct the intercopy edges. Figure 53 shows the result of identifying message B with message B_1 or message B_2 on the intercopy edges.

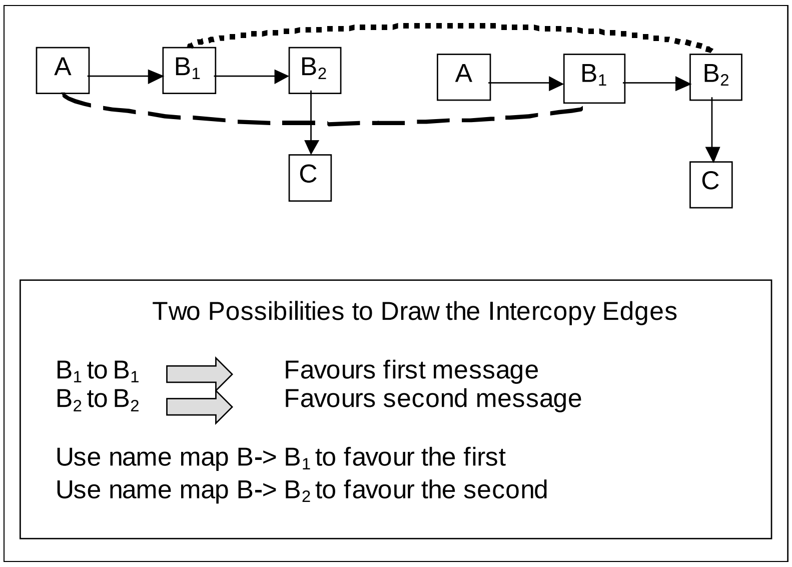

A
B_1
B_2
C
A
B_1
B_2
C
Two Possibilities to Draw the Intercopy Edges
B_1 to B_1
Favours first message
B_2 to B_2
Favours second message
Use name map B-> B_1 to favour the first
Use name map B-> B_2 to favour the second

Figure 53: Using the Name Map function to select the adapter.

For the developer this mapping is again very natural to do. The component usage scenario and the composition pattern are first analyzed to see if they contain several possibilities. The tool then presents a simple dialog box with a message of the composition pattern at one hand side and a set of component calls on the other side. The developer is then asked to identify the best possible match (for example based on the implementation mapping and the state of the component at that point). Once this identification is done, the traversal graph algorithm generates the adapter.

5.3.5 Conclusion

While the algorithm of Reussner returns one clear solution, it can be argued that this is not always a sensible solution. It has however the virtue of being easy to use. The algorithm using the adaptive programming library is as far as we know a new idea and has the advantage of generating *all* possible adaptations. However a lot more user intervention is needed, making this tool a bit harder to use.

6 Code Generation

"Automatic simply means that you can't repair it yourself."

- Mary H. Waldrip

6.1 Introduction

In section 2.2.2 we explained why we use the Java Bean component model throughout this text. Therefore, we need to take a closer look to this component model before we can start generating code.

The Java Bean component model defines two different external interfaces for a Java Bean [EJB, 2001]: API calls and events. This means that there exist two fundamentally different implementation mappings in a component usage scenario. Outgoing messages are mapped on (a set of) events, while incoming messages are mapped on a set of API calls. The most basic communication between Java Beans is that a method is called on one Java Bean in reaction to an event thrown by another component. This is what standard visual composition environments support. They allow you to connect an output event on one component with a method call on another component.

We improve on this model in several ways. First, we allow the reaction to be state dependant. I.e. the same event can cause different methods to be called based on state information. As the glue code we generate knows what events could be received in a given state, it also notices unexpected events. These events can then be ignored to avoid a disruption of the wanted behavior and/or a warning can be issued to the user. Because of our compatibility definition that allows components to offer more than what is asked for, it is possible that a component sends an event that is not supported by the other components. The glue code needs to recognize this. Finally, we allow the same event to cause a sequence of API calls on another component or several events to cause the same API call.

In short, our solution is characterized by the following properties:

- The composition pattern is an active part of the constitutive solution
- The composition pattern is simulated with a state machine
- Communication is via API / event translation from the source component to the right API on the destination component

The automaton resulting from the global checking process contains the information needed for the simulation of the composition pattern. This automaton contains compatible traces only, so it “knows” if a component sends an unexpected event. Therefore, we simulate this automaton to serve as the glue code between the components. It translates outgoing events of one component to incoming calls on another component based on the current state. However, before we can simulate this automaton we need to perform two pre-processing steps. In the first step, we remove all non-valid traces and in the second step, we combine transitions corresponding with sending a message with their subsequent transition corresponding with the reception of this message. These steps are now further explained.

6.2 Preprocessing the Glue Code Automaton.

6.2.1 Remove Non-Valid Traces

Remember that the resulting automaton of the global checking process contains separate transitions for sending and receiving messages. A valid trace needs to comply with the template as shown in Figure 54.

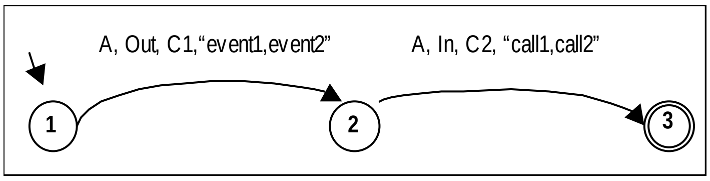


Figure 54: Template for messages with their implementation mapping in the resulting global check automaton.

This template indicates that a component first sends a message and this message is immediately accepted by another component. All traces that send a message out first and receive *another* message afterwards and all traces that receive a message first and send it afterwards are traces that have nothing to do with component interactions. It is trivial to remove all traces that do not comply with this template.

6.2.2 Collapsing Out/In pairs.

In the current automaton every transition is labeled with either a set of events, or a set of API calls. During the simulation, we need to have transitions that link a set of events with a set of API calls. More precisely: in the resulting automaton we want to have the following information on every transition:

1. The source component
2. The destination component
3. The (set of) outgoing event from the source component
4. The (set of) incoming API calls for the destination component

In the previous step, we removed all traces that do not comply with the template depicted in Figure 54. In this step, we construct a new automaton by "collapsing" these Out/In pairs.

During this "collapse" operation, we combine the full implementation mappings corresponding with the outgoing message with the implementation mapping corresponding with the incoming transition. Every transition now contains the wanted information. The process is depicted in Figure 55.

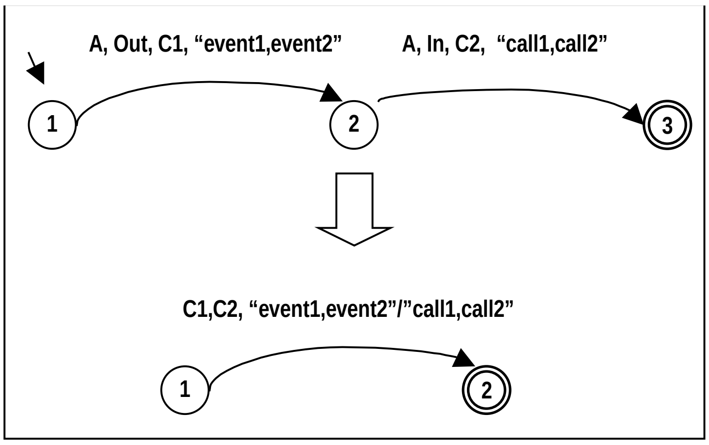
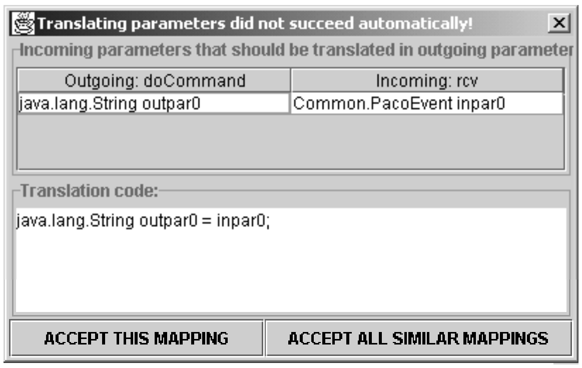

A, Out, C1, "event1,event2"
A, In, C2, "call1,call2"
1
2
3
C1,C2, "event1,event2"/"call1,call2"
1
2

Figure 55: "Collapsing" Out/In pairs.

Technically the collapse operation constructs a Mealy automaton [Hopcroft, 2001]. I.e. we produce an automaton that takes events as inputs and generate API calls as output. As the output depends on the transition rather than on the state, this is a Mealy machine. It is this machine that will be used as our glue code.

6.2.3 Parameter Mappings

Using the documentation, we know which events should be mapped on which methods. The documentation only defines the parameters that can be found in the event and the parameters that are needed for the methods. There is no indication how these parameters should be translated. Take for example a database component that throws an "addressRead" event every time it reads an address record. Take another component that wants to display this address. Using our documentation, we know that on the "addressRead" event the "displayAddress" method should be called on the other component. Now say that "addressRead" contains one string describing this address, while the "displayAddress" method needs a street parameter, a house number parameter, a city parameter and a zip code parameter. The utopian perfect code generator would then insert code that parses the string of the address into the four strings and integers expected by the target component. This is clearly impossible. The current tool therefore checks the type of the input parameter and the output parameter and passes the data along if these types are the same. If the number of parameters differs or if the type of the parameters differ it pops up a dialog box and asks the user to insert the needed conversion code. Figure 56 shows a screenshot of such a parameter-mapping dialog.


Translating parameters did not succeed automatically!
Incoming parameters that should be translated in outgoing parameter
Translation code:
ACCEPT THIS MAPPING
ACCEPT ALL SIMILAR MAPPINGS

Outgoing: doCommand	Incoming: rcv
java.lang.String outpar0	Common.PacoEvent inpar0

```
java.lang.String outpar0 = inpar0;
```

Figure 56: Parameter mapping dialog

It specifies the method that is going to be called (in this case "doCommand"), the event that is received (in this case "rcv") and it specifies the type of the parameters (in this case doCommand has one parameter with type String and the rcv event has one parameter with type PacoEvent). The user can then write a piece of Java code to translate the event parameter(s) in the method parameter(s). He can also accept the default mapping.

6.3 Generating Java Event Handling Code

Before we go into more detail on the actual glue code generation we need to explain the event model of Java Beans. The event model of Java Beans has had a major revision between version 1.0.2 and version 1.3 of the Java Development Kit (JDK). Instead of events percolating up to parents as they did in JDK 1.0.2, any object or component can register itself as a Listener, interested in hearing about a type of events originating in some other component. When the event arises, the source component processes the event by dispatching it to each of the registered Listeners. Dispatching is synchronous, i.e. the Listener handler routines are called directly while the calling dispatcher waits for the handler to complete. According to the specification, the Listeners may be informed in any order, but the usual implementation is a queue, with Listeners informed in the same order they were added.

There are many ways to implement this. We use a little example to show the possibilities and to indicate the problems. We present the code for two Java Beans: "Thrower" and "Receiver" and we describe how the Thrower bean sends an event to the Receiver bean. In a first iteration, we show the most straightforward implementation of the Java Bean event model.

We start with the code for the "Thrower" Java Bean.

```
public class Thrower {
  private Vector listeners = null;

  public Thrower() {
    listeners = new Vector();
  }

  public void addThrowerListener(ThrowerListener listener) {
    listeners.add(listener);
  }
  public void removeThrowerListener(ThrowerListener listener) {
    listeners.remove(listener);
  }
  public void doSomething(){
    System.out.println("The method doSomething is called.");
    notifyListeners();
  }
  private void notifyListeners() {
    for(Enumeration e=listeners.elements(); e.hasMoreElements();) {
      ThrowerListener listener = (ThrowerListener) e.nextElement();
      ActionEvent e = new ActionEvent(this,0,"Thrower Event");
      listener.handleThrowerEvent(e);
    }
  }
}
```

The constructor of this class allocates room for a list of objects. The methods addThrowerListener and removeThrowerListener simply add and remove ThrowerListener objects to this list of listeners. ThrowerListener objects are objects that implement the ThrowerListener interface. The ThrowerListener interface is defined as follows:

```
public interface ThrowerListener {
  public void handleThrowerEvent(ActionEvent e);
}
```

This means that any object implementing the ThrowerListener interface needs to have a method:

```
public void handleThrowerEvent(ActionEvent e);
```

This method contains the code that gets executed when the event is received. The notifyListeners method creates an event (we use a standard actionEvent in this example) and calls the handleThrowerEvent method on every object in its list of listeners.

We now turn to the implementation of the "Receiver" bean.

```
public class Receiver implements ThrowerListener {

  public Receiver () {
    listeners = new Vector();
  }

  public void handleThrowerEvent(ActionEvent e);
    System.out.println("Event received!");
  }
}
```

This bean is used as a listener for events of the Thrower bean. Therefore, this bean implements the ThrowerListener interface. We now build an application that instantiates one Thrower and one Receiver bean. We then add the Receiver as listener to the Thrower. The result is that any time the doSomething method in the Thrower is called; the Receiver's handleThrowerEvent is invoked. I.e. the Thrower instance sends an event to the receiver instance. The code to do this is typically something as:

```
public static void main(String args[]){
  Thrower t = new Thrower();
  Receiver r = new Receiver();
  t.addThrowerListener(r);
  t.doSomething();
}
```

The result of the previous method would be:

```
>The method doSomething is called. Informing listeners.
>Event received!
```

The problem with this naïve implementation is that it hard wires event senders and event receivers. I.e. only those components that implement the right event interface can be used as listeners. In our example, the Receiver bean needs to implement ThrowerListener to add it as a listener to the Thrower bean. To support a more flexible composition, visual component composition tools generate "a class in the middle". This class implements the interface needed to receive events and it calls any method on any other component as specified by the developer.

Suppose for example that we want to use the JTextArea class that is built in, in the Java Development Kit to react on events thrown by our Thrower class. More precisely we want to call the method setText("Event Received") on JTextArea any time the Thrower throws an event. To do this we need to generate the following code:

```
public class GlueCode implements ThrowerListener {
  JTextArea toInform;

  public GlueCode(JTextArea toInform) {
    listeners = new Vector();
    this.toInform = toInform;
  }

  public void handleThrowerEvent(ActionEvent e){
    toInform.setText("Event received!");
  }
}
```

The application now becomes:

```
public static void main(String args[]){
  Thrower t = new Thrower();
  JTextArea j = new JTextArea ();
  GlueCode c = new GlueCode(j);

  t.addThrowerListener(c);
  t.doSomething();
}
```

Thus, instead of adapting the JTextArea component to implement the ThrowerListener interface we generate a dummy component that listens to the events and calls the method we want on the receiving component. This solution also allows calling any method as a result of an event rather than only the method described by the listener interface. This is the standard solution found in all visual composition tools I know (including Visual Age for Java, Symantec Cafe, NetBeans, Forte for Java, Borland JBuilder, Visual J++,… see [Wydaeghe, 2001b] for a detailed overview)

It could be argued that the semantics of the previous solution is somewhat strange. In this implementation, the event handling is synchronous instead of asynchronous. I.e. the component sending the event is blocked while the event is handled. It is the responsibility of the receiving component to return as quickly as possible. Many Java tutorials suggest that components implement an event queue themselves and handle the events in their own thread. The following quote comes from the online version of the Java Tutorial in the chapter about threads [SUN, 2001].

> "Here is an example of using a "SwingWorker" to move a time-consuming task from an action event handler into a background thread, so that the GUI remains responsive.

```
//OLD CODE:
public void actionPerformed(ActionEvent e) {
    ...
    //...code that might take a while to execute is
here...
    ...
}

//BETTER CODE:
public void actionPerformed(ActionEvent e) {
    ...
    final SwingWorker worker = new SwingWorker() {
        public Object construct() {
            //...code that might take a while to
execute is here...
            return someValue;
        }
    };
    worker.start();  //required for SwingWorker 3
    ...
}
```

> The value that "construct" returns can be any object. If you need to get the value, you can do so by invoking the "get" method on your SwingWorker object. Be careful about using "get". Because it blocks, it can cause deadlock. If necessary, you can interrupt the thread (causing get to return) by invoking "interrupt" on the SwingWorker."

The author here suggests that the actionPerformed event is handled in a separate thread and provides a default class (called SwingWorker) that can be subclassed to do this. Note that this SwingWorker class is not part of the standard Java Development Kit. I.e. programmers are forced to program asynchronous event handling themselves.

Even in small experiments, this synchronous event handling leads to stack overflow problems. Stack overflow occurs whenever an API called in response to a certain event generates the same event, because the API call only finishes as the new event is handled an that one only finishes when its new event is handled and so on. Therefore, instead of an endless loop, we get endless recursion.

It is easy to generate glue code that handles events asynchronously. To do this we have two options: do the event dispatching asynchronously or handle every event asynchronously. The first solution implements an event queue that accepts events and uses its own thread to poll this queue and handle the event. The glue code of the example now becomes:

```
public class GlueCode extends Thread implements ThrowerListener {
  JTextArea toInform;
  Queue     eventQueue;

  public GlueCode(JTextArea toInform) {
    listeners     = new Vector();
    eventQueue    = new Queue();
    this.toInform = toInform;
    start();
  }

  public void handleThrowerEvent(ActionEvent e){
    eventQueue.push(e);
  }

  private void run(){
    while(true){
      ActionEvent e = eventQueue.pop();
      if (e != null) toInform.setText("Event received!");
    }
  }
}
```

This solution could be improved using the built in system event queue, but the idea is clear.

The second solution uses the solution as explained in the Java Tutorial quote. I.e. the glue code becomes:

```
public class GlueCode extends Thread implements ThrowerListener {
  JTextArea toInform;

  public GlueCode(JTextArea toInform) {
    listeners     = new Vector();
    this.toInform = toInform;
  }

  public void handleThrowerEvent(ActionEvent e);
    final SwingWorker worker = new SwingWorker() {
       public Object construct() {
           return toInform.setText("Event received!");
       }
   };
   worker.start();
  }
}
```

The second solution is very expensive as it creates a thread for every event. Tests with these possibilities revealed that even the first solution is very expensive (to use it we need to insert at least a delay in the event polling loop). This performance penalty explains why our prototype implements the synchronous solution rather than the more natural asynchronous solution.

6.4 Generating Code

We now turn to the problem of generating glue code that considers state information. More precisely, we want our glue code to implement the resulting Mealy automaton from the global compatibility check after the pre-processing. There exists a lot of literature on the implementation of automata (see for example [Aho, 1985]). The glue code to combine three components A, B and C using a state machine with 2 states now becomes:

```
public class GlueCode implements Alistener, Blistener, CListener {
  StateMachine stateMachine;
  A a;
  B b;
  C c;

  public GlueCode(StateMachine stateMachine, A a, B b, C c) {
     listeners  = new Vector();
     this.stateMachine = stateMachine;
     this.a     = a;
     this.b     = b;
     this.c     = c;
  }

  public void handleAEvent(ActionEvent e){
     processEvent(e, stateMachine.getCurrentState());
  }
  public void handleBEvent(ActionEvent e){
     processEvent(e, stateMachine.getCurrentState());
  }
  public void handleCEvent(ActionEvent e){
     processEvent(e, stateMachine.getCurrentState());
  }

  public void processEvent(ActionEvent e, PacoSuite currentState){
     String name = e.getMessage();
     PacoState nextState = stateMachine.doTransition(name);
     If (nextState == null) return;
     switch((int) currentState.getId()) {

        case STATE_1: {
           if (name.equals("actionPerformed") && (src == a)){
              b.Launch();
           }
           break;
        }
        case STATE_2: {
           if (name.equals("rcv") && (src == b)){
              c.signalReceived();
           }
           else if (name.equals("readyToChooseSession") && (src == c)){
              a.send();
           }
           break;
        }
        default: {
           System.out.println("Illegal event received: " + name);
           break;
        }
     }
```

```
    }
}
```

Thus, when an event is received, we first check if we find a transition labeled with this event that allows us to advance one step in the automata. Next, we execute depending on the state the right API call. We further generate a main method were the Mealy automata corresponding with the composition patterns in our application and all cooperating components are instantiated. This class also subscribes the Mealy automata implementations to receive the events of every component that is a member of its corresponding composition. All Mealy automata are started in their own thread.

Solving the odds and ends like handling multiple events in one state and user interaction for the parameter mapping is trivial. The full code of our prototype can be downloaded at [Wydaeghe, 2001a].

7 The Exam Construction Kit

7.1 Introduction

The application we illustrate here is a simple distributed exam service. The exam service provides a teacher with the possibility to set up an exam server that provides a set of multiple-choice questions and handles the interaction with the students during the exam. The exam client application provides a login to a student and connects to the exam server. After login, the student receives the first question from the server. The student selects an answer and sends it back to the exam server. The exam server stores the answer in a database and sends the next question. Once all questions are answered, the exam server produces a report for the teacher that gives an overview of the performance of the student. During the exam, the teacher can follow the progress of all examinees.

The next section introduces the usage scenarios and the composition pattern used in this example.

We show that there exist a spectrum of usage scenarios and composition patterns ranging from on the one hand: "very basic and simple, but very generic" and on the other hand: "very complex but application specific" and everything in between.

We then show how we use these components and composition patterns in our PacoSuite prototype to build the distributed exam service.

7.2 Documentation

In this section, we introduce the usage scenarios and the composition patterns needed to build the distributed exam service.

7.2.1 Components

7.2.1.1 Client side components

7.2.1.1.1 The Client User Interface

First, the components on the client side of the exam service are described. The first component we document is the user interface component. A typical usage scenario for such a component is that is launched first. After this launch, it loops forever waiting for new input and throwing events whenever the user answers a question. This is a very general usage scenario that can be used for a multiple choice exam user interface, an exam with or without pictures, sound, video and for a plain text exam. In this case, it is used to document the DrivingExamGUI component. This Java Bean has the following API and events.

// API
- public void Launch()
- public void submitAnswer(Object answer)
- public void doCommand(String signal)
- public void setProgressBarVisible(boolean b)
- public boolean isProgressBarVisible()
- public String getLookAndFeel()
- public void setLookAndFeel(String lf)
- public void setLanguage(String language)
- public void setFont(String font)

// listener management
- public void addUserExamListener(UserExamListener l)
- public void removeUserExamListener(UserExamListener l)

// events
- private void notifyReadyToChooseSession()
- private void notifyAnswer(String answer)
- private void notifyQuestion()
- private void notifySessionSelected(String sessionName)
- private void notifySessionJoined(String name)

The documentation for this component is depicted in Figure 57. The START primitive is mapped on the Launch API call. The generic super type DATA is used to indicate that any DATA handling code is expected to handle the answer event. All other events are not mapped on any primitive. This means that all other events thrown by this component are ignored as far as this usage scenario is used. The same goes for all property setting API calls. This indicates how the usage scenario captures typical uses. I.e. it shows that the properties setting API and many events can be ignored. It captures the knowledge of the developer of the component on how this component is used in practice.

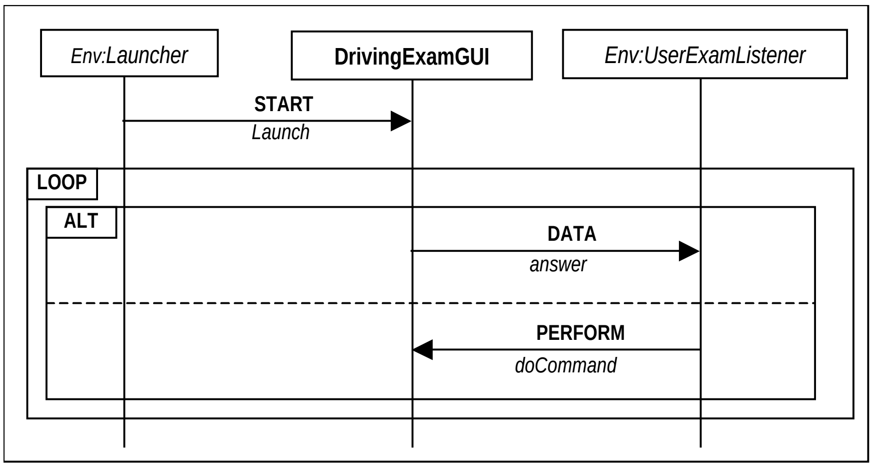


Figure 57: DrivingExamGUI usage scenario

7.2.1.1.2 *A Standard Java Button*

To launch the application a standard Java Button (JButton) is used. This is a good example to show the problems with plain API documentation. The following is a copy from the standard API documentation (JavaDoc) for the JButton component. Neither constructors, nor super class methods are mentioned.

- void configurePropertiesFromAction(Action a)
  Factory method which sets the AbstractButton's properties according to values from the Action instance.
- AccessibleContext getAccessibleContext()
  Gets the AccessibleContext associated with this JButton.
- String getUIClassID()
  Returns a string that specifies the name of the L&F class that renders this component.
- boolean isDefaultButton()
  Returns whether or not this button is the default button on the RootPane.
- boolean isDefaultCapable()
  Returns whether or not this button is capable of being the default button on the RootPane.
- protected String paramString()
  Returns a string representation of this JButton.
- void removeNotify()
  Overrides JComponent.removeNotify to check if this button is currently set as the default button on the RootPane, and if so, sets the RootPane's default button to null to ensure the RootPane doesn't hold onto an invalid button reference.

- void setDefaultCapable(boolean defaultCapable)
  Sets whether or not this button is capable of being the default button on the RootPane.
- void updateUI()
  Notification from the UIFactory that the L&F has changed

The most typical use of a JButton is to throw an event if it is clicked. However there is no indication in the previous list on how this can be achieved. Going up one level in the inheritance hierarchy of the JButton gives the following, rather impressive, list of methods:

> addActionListener, addChangeListener, addItemListener, checkHorizontalKey, checkVerticalKey, createActionListener, createActionPropertyChangeListener, createChangeListener, createItemListener, doClick, doClick, fireActionPerformed, fireItemStateChanged, fireStateChanged, getAction, getActionCommand, getDisabledIcon, getDisabledSelectedIcon, getHorizontalAlignment, getHorizontalTextPosition, getIcon, getLabel, getMargin, getMnemonic, getModel, getPressedIcon, getRolloverIcon, getRolloverSelectedIcon, getSelectedIcon, getSelectedObjects, getText, getUI, getVerticalAlignment, getVerticalTextPosition, imageUpdate, init, isBorderPainted, isContentAreaFilled, isFocusPainted, isFocusTraversable, isRolloverEnabled, isSelected, paintBorder, removeActionListener, removeChangeListener, removeItemListener, setAction, setActionCommand, setBorderPainted, setContentAreaFilled, setDisabledIcon, setDisabledSelectedIcon, setEnabled, setFocusPainted, setHorizontalAlignment, setHorizontalTextPosition, setIcon, setLabel, setMargin, setMnemonic, setMnemonic, setModel, setPressedIcon, setRolloverEnabled, setRolloverIcon, setRolloverSelectedIcon, setSelected, setSelectedIcon, setText, setUI, setVerticalAlignment, setVerticalTextPosition

This list contains a method: “addActionListener”. Reading the Swing tutorial (the documentation of the method self only mentions that an actionListener is added to the button) reveals that actionlisteners are notified whenever the button is pressed. Providing the usage scenario depicted in Figure 58 summarizes this information.

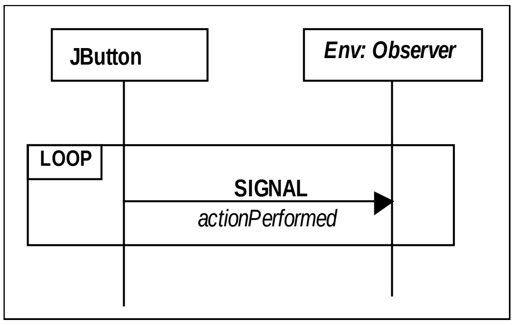


Figure 58: Typical use of a standard JButton Component

7.2.1.1.3 The Network Component

The last component we need on the client side is a network client component. This component sends strings over a TCP/IP connection and throws events when a string is received or when the connection is established or destroyed. It has a little user interface that allows the end user (thus not the developer) to specify a host name and a port number and a connect button. If the connect button is hit, the component tries to set up a connection. If it succeeds it throws the "rcvConnect" event, if it fails it throws the "rcvClose" event. The default behavior of this component is therefore to throw first the "rcvConnect" event, then to receive and send data until a disconnect is received, when it is ready to make a new connection and start all over. This is depicted in Figure 59.

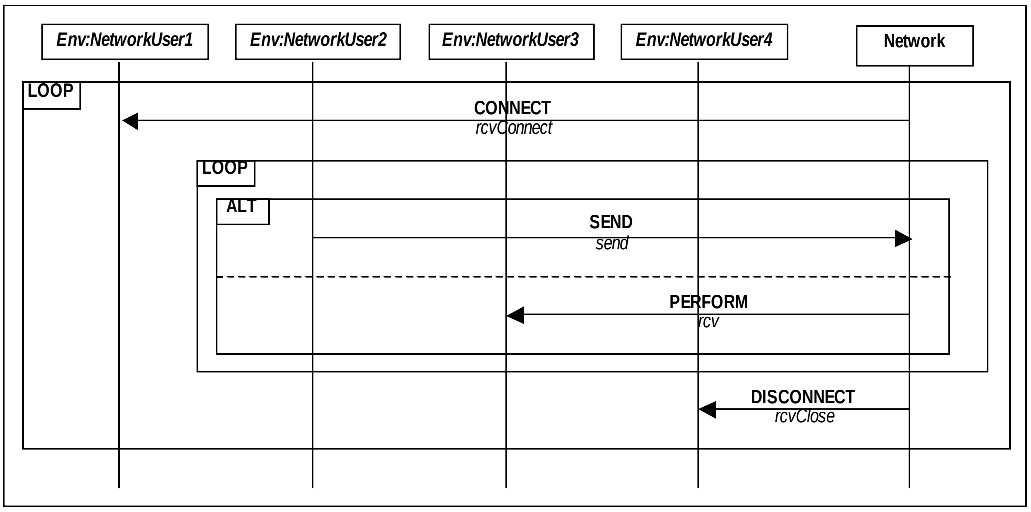


Figure 59: Usage scenario for the network client component

Note that four different environment participants are used. This allows us to use four different components to provide each one of the environments expected by the network component. In addition, because it is allowed to map one role on many environment participants (see section 4.9.1), this documentation also allows one component to provide all these environments.

7.2.1.2 *Server side*

At the server side, we also use a standard Java Button and a network client component. The components differing from the client side are the exam server component and a generic network server component.

7.2.1.2.1 *The ExamServer Component*

The ExamServer component supports a lot of different usage scenarios, but the most basic one is that it is launched and starts throwing events and listening for commands. At the level of primitives, it behaves exactly the same as the user interface component at the client side. The only difference is its implementation mapping. The usage scenario for the ExamServer component is described in Figure 60.

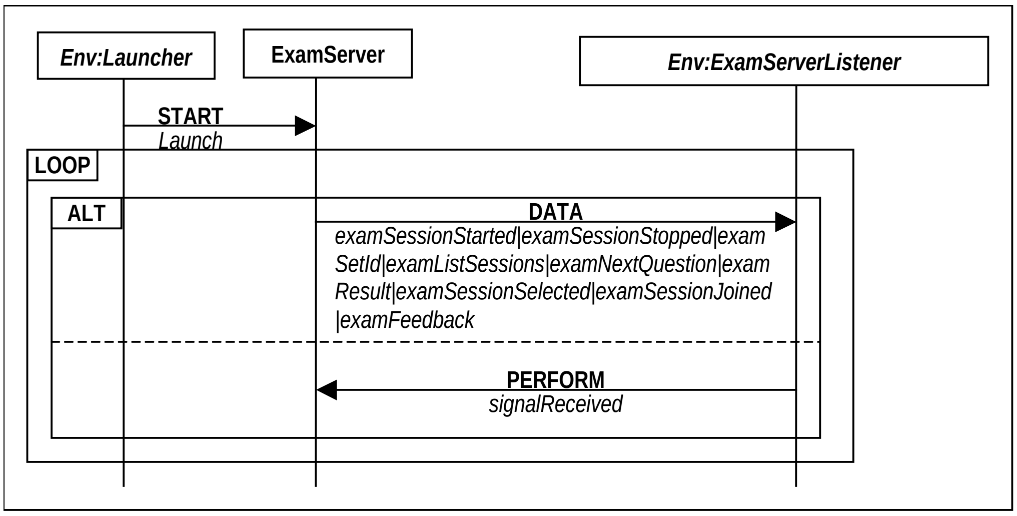

Env:Launcher
ExamServer
Env:ExamServerListener
START
Launch
LOOP
ALT
DATA
examSessionStarted|examSessionStopped|examSetId|examListSessions|examNextQuestion|examResult|examSessionSelected|examSessionJoined|examFeedback
PERFORM
signalReceived

Figure 60: Documentation of the driving exam server component

7.2.1.2.2 *The NetworkServer Component*

The NetworkServer component opens a port where it listens for connections. It support sending broadcast messages and point to point messages. Its use is very simple however. Or you simply launch it listening on the port set in its properties, or you launch it in GUI mode, where it pops up a window where you can specify the server port and hit a button to start the server. This is depicted in Figure 61.

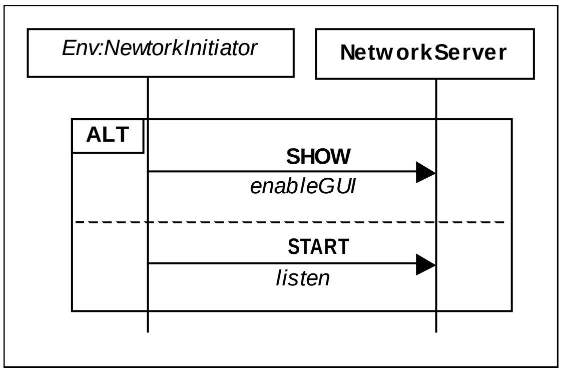


Figure 61: Description of a generic network server component

7.2.2 Composition Patterns

There are many possibilities for the composition patterns. We could have one composition pattern for the client side and one for the server side, but these composition patterns would be very specific. One of the ideas behind composition patterns is that they should be generic and reusable. The spectrum of composition patterns ranges from the simplest pattern describing one message between two roles, to composition patterns describing a full application. In practice, we use a mix of application specific (and in general more complex and less reusable) composition patterns together with a set of very basic (and in general simple and highly reusable) composition patterns. In the example application, only three distinct composition patterns are used: one application specific composition pattern and two basic patterns.

7.2.2.1.1 Application Specific Pattern

The first composition pattern describes the interaction between a launcher to start a network-based application, the network based application itself and a network role. This composition pattern is shown in Figure 62.

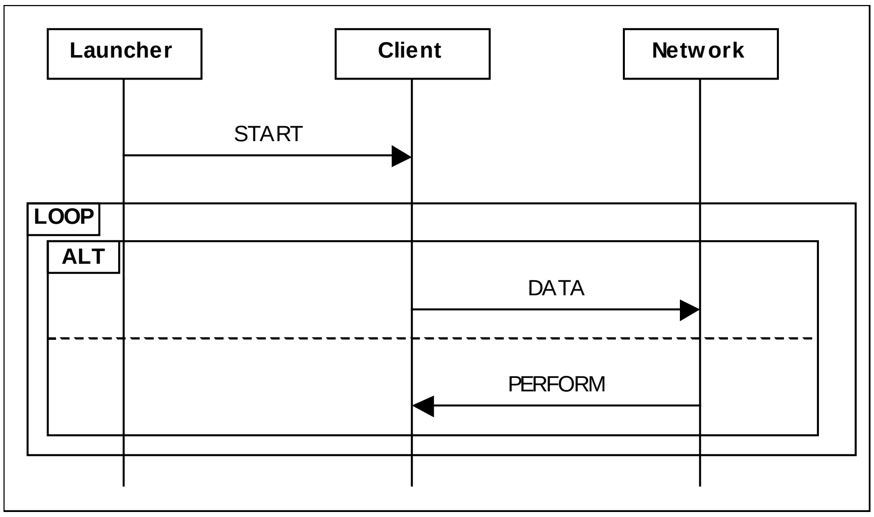


Figure 62: LaunchClientNetwork Composition Pattern describing the interactions between a launcher, a client and a network role

This composition pattern is in fact an example of a composition pattern between the two extremes of the spectrum. This composition pattern proved to be reusable for several other distributed applications, but it is not reusable outside the scope of network applications.

7.2.2.1.2 *Basic Composition Patterns*

We use two basic composition patterns: one to make something visible and one to start something. These are shown in Figure 63.

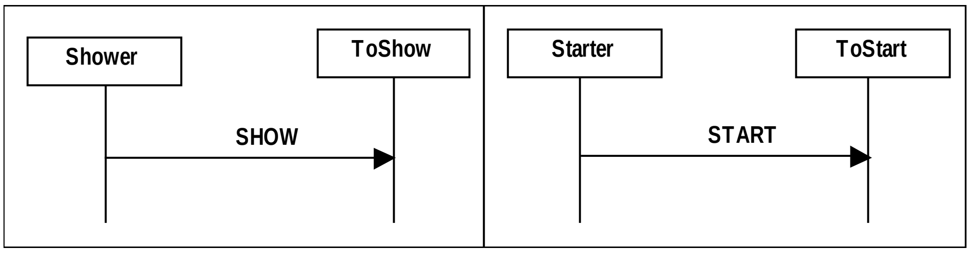


Figure 63: Two basic composition patterns.

7.3 The Composition Process

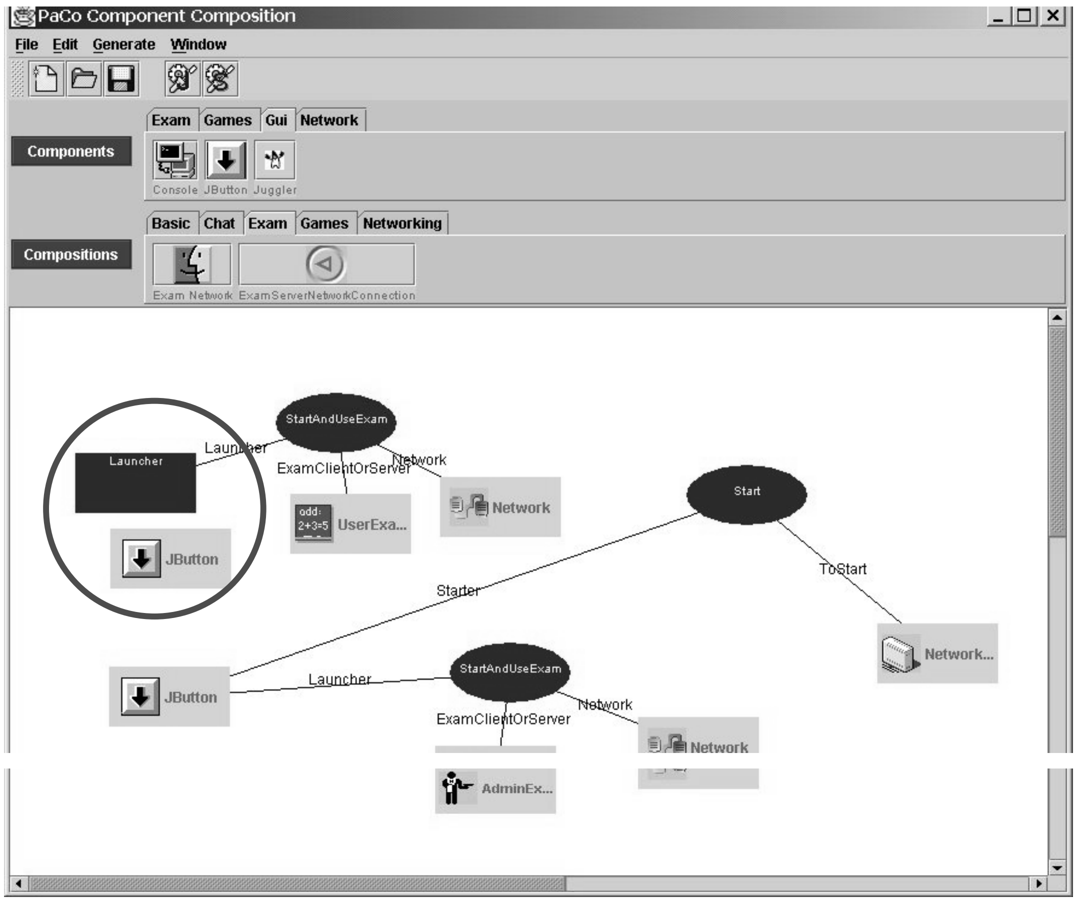
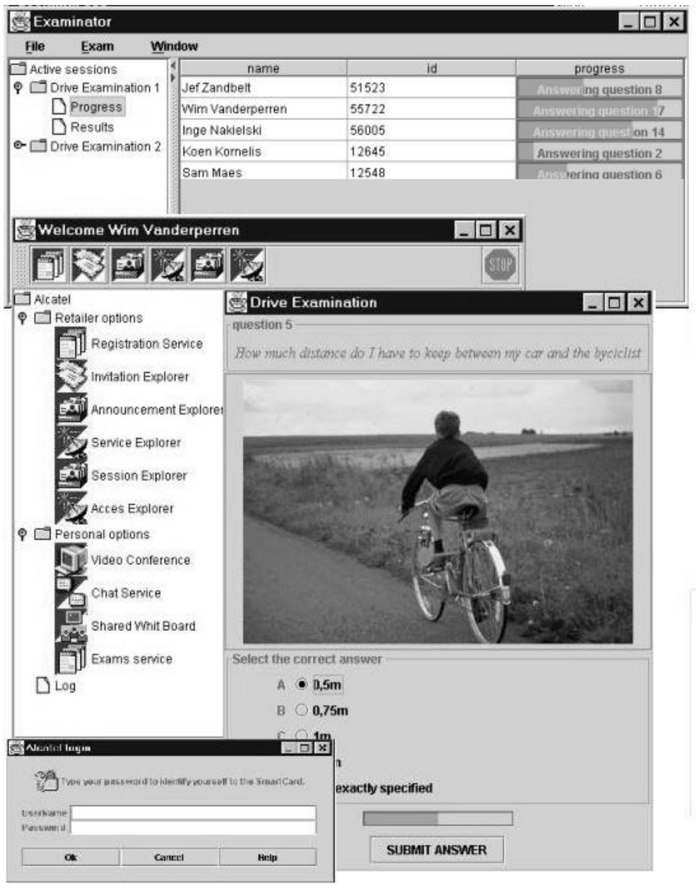


Figure 64: Snapshot of the development of an exam application using the exam construction toolkit.

Figure 64 show a snapshot taken during the development process of the exam application. The application uses three composition patterns. At the client side the LaunchClientNetwork (see Figure 62) composition pattern is used. The roles of this composition pattern are filled with a Network component, a standard JButton and an Exam User Interface component. The user already filled all the roles, except for the launcher role of the LaunchClientNetwork composition pattern. He is about to drag the JButton component onto this role (indicated by the circle). Using the local checking algorithm, we check whether the JButton component is compatible with the corresponding role. The drag is refused in case of a mismatch. In this case, the drag is accepted, but for example dragging a Network component onto Launcher role will be refused. The composition pattern at the top represents the client side of the exam and the composition pattern at the bottom is the administrator or teacher side of the application. Notice that the second button is used in two composition patterns, i.e. when the button is pressed it will start both the network server and the administrator interface. When the user initiates the glue-code generation, all filled composition patterns are checked as a whole using the global checking algorithm and glue-code to is generated. The resulting application is then launched. Figure 65 shows a compilation of screenshots of the resulting application.



name	id	progress
Jef Zandbelt	51523	Answering question 8
Wim Vanderperren	55722	Answering question 17
Inge Nakielski	56005	Answering question 14
Koen Kornelis	12645	Answering question 2
Sam Maes	12548	Answering question 6

question 5

How much distance do I have to keep between my car and the bycyclist

Select the correct answer

A [✓ 0,5m]
B 0,75m
C 1m
n
exactly specified

Type your password to identify yourself to the SmartCard.

UserName
Password

Figure 65: Compilation of screenshots of the exam service application.

To illustrate the power of our approach we show how chat functionality is added to the exam (see Figure 68). Therefore, two instances of the Console component are added. The usage scenario for this Console component is depicted in Figure 66.

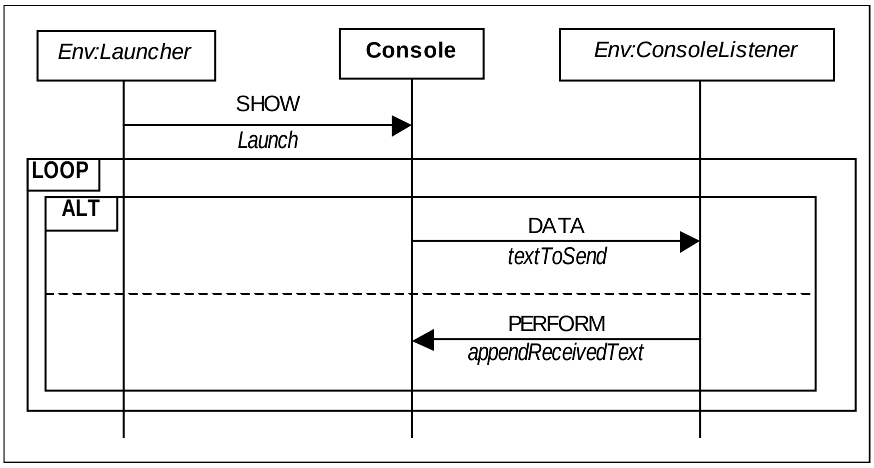


Figure 66: Usage scenario for a Console component

Both Console components are used as chat window. To use them one extra composition pattern is required. This composition pattern describes the interaction between a network and a network user. This pattern is depicted in Figure 67.

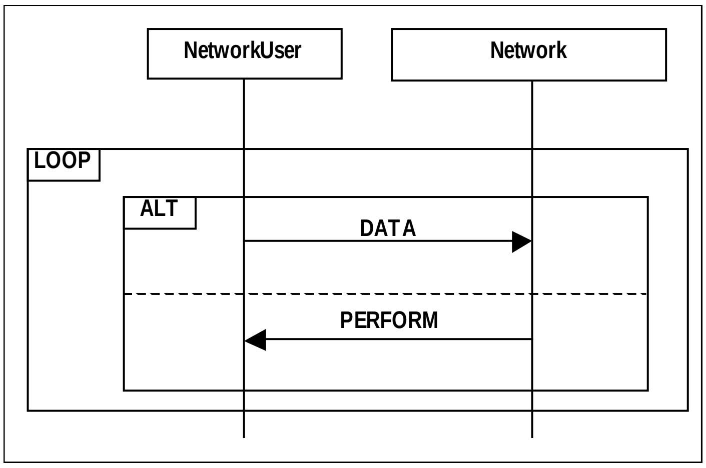


Figure 67: UseNetwork composition pattern

Chat functionality is added by filling the network user role of this composition pattern with the Console component and the network role with the Network component we already have in the LaunchClientNetwork composition pattern at the client side. We also use the Show composition pattern to make the Console window visible if the launch button at the client side is pressed. The same operation is done at the server side. The resulting composition is shown in Figure 68. Note that the same composition patterns are used both at the client side and at the server side indicating that these composition patterns are reusable.

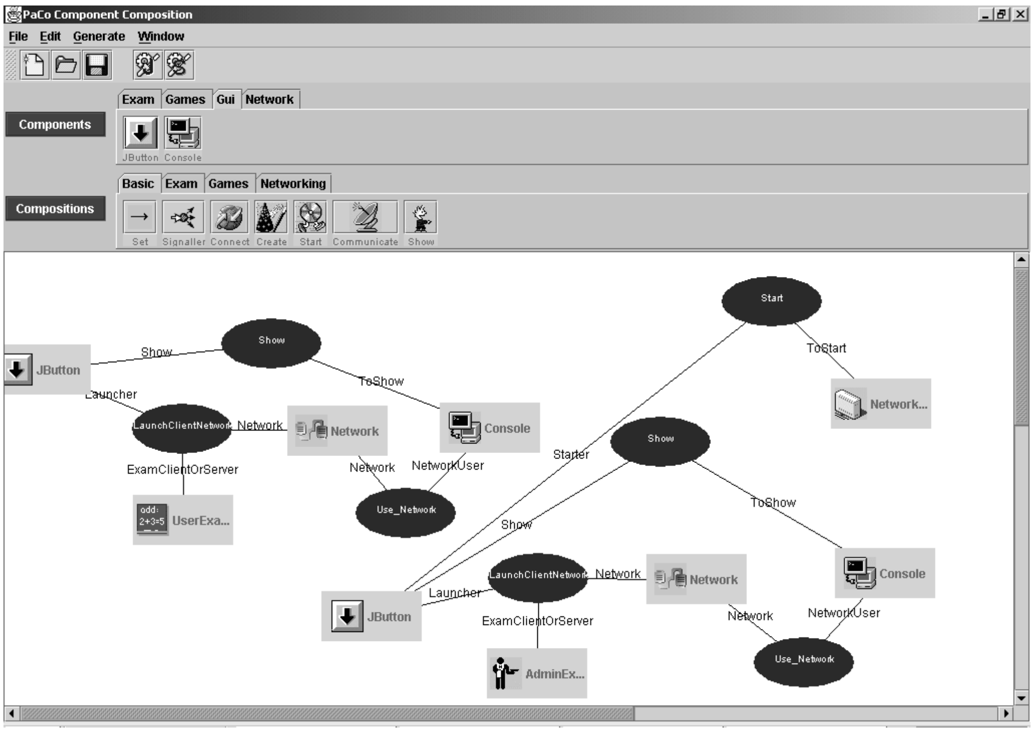


Figure 68: Extending the exam application with chat functionality.

7.4 Conclusions

This small example indicates that the concept of composition patterns allows us to improve on state of the art visual component composition tools by lifting the abstraction level of the wiring. It allows users to concentrate on the application rather than on technical details. This leads to a construction kit that is easy to use without sacrificing flexibility.

The exam construction kit was demonstrated both for our industrial partner Alcatel and during the final review of the Advanced Internet Access (AIA) project. The tool and the demonstration were very well received on both occasions.

8 Conclusions

“Enough research will tend to support your conclusions.”

- Arthur Bloch In "Quotable Business," ed. Louis E. Boone, 1992

-

"C'est le temps que tu a perdu pour ta rose qui fait ta rose si importante."

- Antoine de Saint-Exupéry, Le Petit Prince

8.1 Contributions

In section 1.3, we mention five main contributions of this thesis. Our first claim is that we improve current visual component composition environments using the concept of composition patterns. To support this claim we argue first why using composition pattern is at least as good as current component composition environments. Current state of the art component composition environment allow you to connect any event with any method. We obtain the same kind of wiring by documenting every component with the usage scenario (left hand side of the picture) and the composition pattern (right hand side of the picture) as shown in Figure 69.

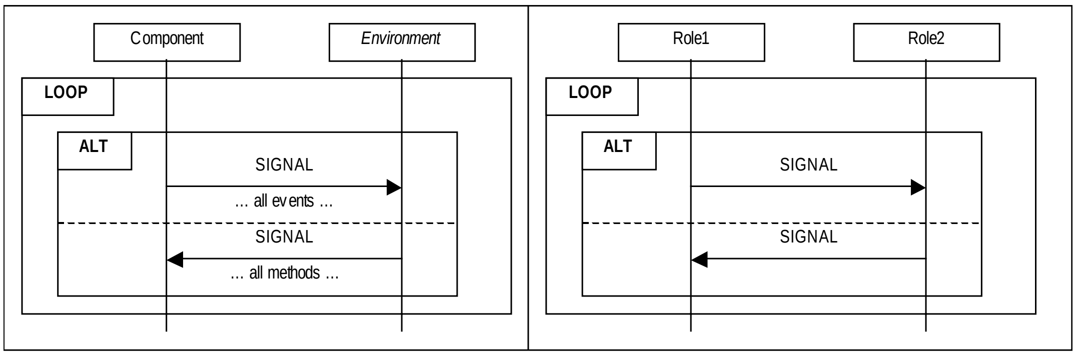


Figure 69: Generic usage scenario and composition pattern

This usage scenario specifies that a component can throw any event at any given moment. It also specifies that any method can be called at any given time. Documenting components with this kind of usage scenarios and subsequentially composing these components using this kind of composition pattern in our prototype tool launches a dialog box where links between events and their corresponding API call can be specified. This is the same as what current visual component composition environments provide.

Composition patterns improve on this scheme as they allow a developer to specify the order of events and method calls. They also support the specification of multi party composition patterns whereas current tools are limited to binary relationships. Composition patterns further support the reuse of wiring information (wiring in current commercial tools cannot be stored and reused independently of an application. It needs to be redone for every new application) and they make compatibility checking possible. This leads us to our second claim.

We claim that we can do an automatic compatibility check using a compatibility definition that allows components to offer more than what the composition pattern asks for and allows composition patterns to be more general than what the components offer. To this end, we developed checking algorithms based on finite automata theory using the intersection of automata rather than the difference. To avoid problems at runtime for calls outside this intersection, we generate a special kind of glue code. This leads us to our third claim.

We claim that glue code can be generated that both forces components to follow only compatible traces and restricts the composition pattern to what the components can offer. This allows us to use more generic and more reusable composition patterns than what is proposed by current architectural description languages. This is done by implementing the resulting state machine of our compatibility check. As this state machine only contains valid traces, it is able to blocks all other traces resulting in the wanted runtime behavior.

The fourth contribution is improved feedback during the component composition process. We not only add usage protocol checking to the composition process, but also a check if the specified composition pattern can be used with the selected set of components. None of these checks exists in current tools. In case a mismatch is detected, we provide a flexible adapter generator that can be used to search for possible adaptations of the components, composition patterns, or glue code, to cure the mismatch. We noticed the similarities between algorithms used in the adaptive programming field [Lieberherr, 1997] and the algorithm used by Reussner to generate adapters [Reussner, 1999]. Applying the adaptive programming algorithm to adapter generation leads to a more flexible and efficient adapter generator process.

The last claim we made is that we provide support for "composition based" construction of component-based applications. This means that instead of selecting a set of components and trying to hack them together, we can search for a set of composition patterns first and select the components based on their compatibility with these composition patterns. As composition patterns are documented using a kind of sequence diagram, they correspond very well with use cases built to specify the requirements.

With the small distributed exam application, we indicated how this work could be used to build very flexible construction kits. It allows the developers of such a kit to provide standard composition patterns together with their set of components without touching the ability of the users of these construction kits to build very complex applications that were not foreseen by the developers.

8.2 Applicability

8.2.1 Influence on Visual Component Composition

It would be very naive to think that the prototype tool we developed in this work will be the component composition tool of the future. The most we can hope is that some of the ideas will make it to future commercial component composition environments.

A tool that shows a lot of efforts in this direction is Visual Age for Java. Recently IBM has introduced the concept of "helpfull beans" in their Visual Age for Java product. A "helpfull bean" is a component with built in documentation describing typical wiring schemes. To a certain extent, these can be seen as usage scenarios. It specifies no order of events or method calls but it does indicate the events and method calls that are most often used to compose this component. Until now, this documentation is passive. I.e. it is not possible to instantiate a typical wiring and to fill in the blanks. However there are rumors that this will be possible in further versions.

The same tool also allows to "swap" components in existing wiring. I.e. it is possible to update a given component with another component. If the new component has another set of events and/or method calls the tool allows the user to reroute the wiring that is not "compatible" anymore.

Many tools also support a kind of ordering of events on their components. If an event needs to invoke several method calls on several components, it is possible to specify in what order these methods are called.

These are all indications that current tools search for ways to improve the wiring process and the usage documentation of their components. This work provides a set of ideas that are very useful in this context.

8.2.2 Scalability

An important issue in the application of these ideas is scalability. Even small experiments prove that the checking algorithms start to take to much time when composition patterns reach more than ten participants. There is no real limit to the use of multiple composition patterns though. Current visual composition tools only support binary composition patterns and large applications are built using many components and many composition patterns. We can take the same approach. I.e. we compose large applications using a large set of composition patterns and a large set of components. However, this means that we use the results of this work only locally. There is no possibility to define high-level synchronization between subsystems this way. The obvious approach to cure this problem is to make a composition of a set of components using one composition pattern and to pack the result into a new component. This new component needs a new set of usage scenarios. This can be done manually or semi automatic. Current visual composition environments all ask user input to provide the new interface of the composed component. One possibility to provide this set of usage scenarios automatically is to take the union of all usage scenarios of the constituent components, but this leads to a huge set of usage scenario for components that are made from a large collection of other components. We also experimented with composition patterns that are only partially filled where we consider the interaction between the empty roles and the filled roles as the usage scenario for the new component. However, this leads quickly into problems when using multiple composition patterns to build a new component. In general, we state that a new component needs to be documented manually, but we can support this documentation process with automatic tools.

As current visual composition environments proved to be scalable enough to be used in industrial strength applications, we do not foresee scalability problems with he improved approach presented in this work.

8.2.3 Influence on Other Research

This work mainly uses the results of finite state machines, architectural description languages, adapter generation and the adaptive programming library. We think that many of the results in this work could be used in these areas to improve the current results.

We use an algorithm based on dynamic programming to find contradicting role/component mappings in a state machine resulting from a global check between a set of components and a composition pattern. In general, this algorithm is a very efficient implementation to find all traces constrained by all its values along the trace. I.e. constraints like "find all traces where the label "a" occurs at most n times". These are very common constraints for regular languages.

In the area of architectural description languages, we think that the algorithms explained in this text could be used to improve the efficiency of the compatibility check as proposed by [Allen, 1994b]. At the moment they use a theorem prover to do this. A small adaptation to our global check algorithm would be a more efficient solution. The only adaptation we need to do is to take the difference rather than the intersection between the composition pattern and the components (and calculating the intersection and difference automaton is very similar in automata theory). In their work, they prove that a composition is deadlock free if, among others, their glue code is conservative (for a definition see [Allen, 1997]). Our glue code generation generates conservative glue code. It could be interesting to apply our ideas to their research to force glue code to be conservative instead of just assuming it.

We further explained how our adapter generation algorithm improves the adapter generation algorithms we found in literature.

We also indicated the mapping between the parallel composition operator ($\|$) as defined in CSP and the calculation of a traversal graph as defined by [Lieberherr, 1997]. This can only help in improving the understanding and the implementation of these algorithms.

8.3 Future Work

No work is ever finished and this one is no exception. On the first place, there are improvements to the efficiency of our algorithms. One idea that is worth looking at is to do the global check in a tree like fashion rather than incrementally. As the $\|$ operator is associative, we can place the brackets anywhere we like with the same result. Another interesting idea is to use state machine minimization. This should speed up our algorithms, but it is far from trivial to keep track of the implementation mappings.

Support for automatic documentation generation for a new component made from a composition of other components is another item on the wish list. We have done a set of experiments, but especially multiple composition patterns and the conversion from state machines to MSC's gave us troubles. There exists a well-known conversion algorithm [Hopcroft, 2001] to do the latter, but it results in very strange MSC's. The conversion can be done in many different ways and the standard algorithm does not necessarily result in the more intuitive one. It would be interesting to try to guide the conversion based on the original MSC's.

My colleague Wim Vanderperren takes a more ambitious direction. He tries to build on the aspectual component and AOP research to add aspect weaving to the component composition process. The idea is to have a special kind of composition patterns that adapt other composition patterns. Figure 70 gives an example of an adapter composition.

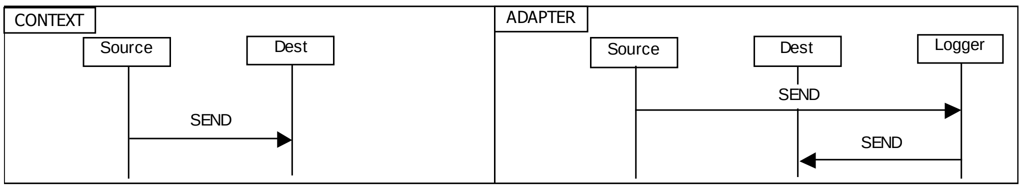


Figure 70: Composition Adapter: CONTEXT specifies where to apply and ADAPTER specifies what to do.

In this example, the adapter composition will re-route every occurrence of a SEND from role *Source* to role *Dest* through a *Logger* role. The composition pattern shown at the left hand side of Figure 71 shows the result of applying the composition adapter of Figure 70 to the composition pattern shown at the right hand side of the picture.

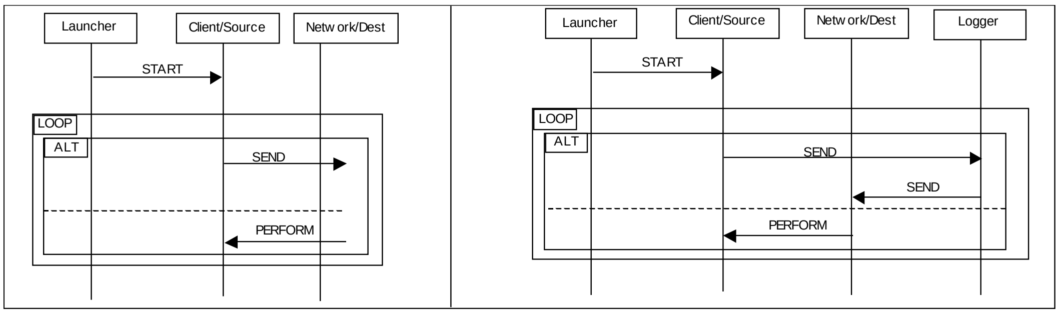


Figure 71: Applying the composition pattern adapter to the original composition pattern on the left hand side results in the composition pattern at the right hand side.

I.e. we construct a new composition pattern by first combining composition patterns with a set of adapting patterns. Benefits do not only arise in the development of the component-based application, but also in evolutionary changes to the application. New requirements often lead to concerns that crosscut the existing component composition. Instead of having to re-wire all the components, the existing composition patterns are altered using adapter compositions.

There are plenty of other possibilities left to build onto our current results and further improve the component based development process.

9 Products and Specifications

ADA, "ADA Reference Manual", http://www.ada-auth.org/arm.html

BeanBox, "Java BeanBox (SUN)", http://java.sun.com/products/javabeans/docs/spec.html

COM, "Component Object Model (COM)", http://www.microsoft.com/com/tech/com.asp

CORBA, "The Common Object Request Broker: Architecture and Specification- Revision 1.2", http://www.omg.org/technology/documents/formal/corbaiiop.htm

DCOM, "Distributed Component Object Model (DCOM) - Downloads, Specifications, Samples, Papers, and Resources for Microsoft DCOM", http://www.microsoft.com/com/tech/DCOM.asp

Delphi, "Borland Delphi", http://www.inprise.com/delphi/

EJB, "Enterprise JavaBeans Specification", http://www.javasoft.com/products/ejb/docs.html

IDL, "CORBA 2.4.2 OMG IDL Syntax and Semantics chapter", http://www.omg.org/cgi-bin/doc?formal/01-02-07

MSC. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). 1993. Geneva.

NetBeans, "Net Beans", http://www.netbeans.org/index.html

UML, "UML 1.3 Documentation", http://www.rational.com/uml/resources/documentation/index.jsp

VisualAge, "IBM Software : Application Development : VisualAge Developer Domain", http://www7.software.ibm.com/vad.nsf/

VisualBasic, "Microsoft Visual Basic Home Page", http://msdn.microsoft.com/vbasic/

VisualJava, "Microsoft Visual J++ Home Page", http://msdn.microsoft.com/visualj/

ActiveX, "ActiveX Controls - Microsoft Papers, Presentations, Web Sites, and Books, for ActiveX Controls", http://www.microsoft.com/com/tech/activex.asp

10 References

Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers Principles, Techniques and Tools.* 1985.

Allen, R. and Garlan, D. Formal Connectors. CMU-CS-94-115. 1994a.

Allen, R. and Garlan, D. Formalizing Architectural Connection. 71-80. 1994b.

Allen, R. and Garlan, D., "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213-249, 1997.

Alpert, S. R., Brown, K., and Woolf, B., *The Design Patterns Smalltalk Companion* Addison-Wesley, 1998.

Bach, M. J., "The Design of the UNIX Operating System," *Software Series* Prentice-Hall, 1986, pp. 111-119.

BeanActiveXBridge, "http://java.sun.com/products/javabeans/software/bridge/index.html,".

Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D., "Making Components Contract Aware," *IEEE Computer*, pp. 38-44, 1999.

Booch, G., *Object-Oriented Analysis and Design with Applications* Santa Clara, California: The Benjamin/Cummings Publishing Company, Inc., 2001.

Brand, D. and Zafiropulo, P., "On Communicating Finite-State Machines," *Journal of the ACM*, vol. 30, no. 2, pp. 323-342, 1983.

Campbell, R. and Habermann, A. The specification of process synchronisation by path expressions. 89-102. 1974. Springer-Verlag.

Christopher Alexander, *The Timeless Way of Building* 1979.

Clarke, S. and Walker, R. Composition Patterns: An Approach to Designing Reusable Aspects. 2001. ACM Press.

Compare, D., Inverardi, P., and Wolf, A. L., "Uncovering architectural mismatch in component behavior," *Science of Computer Programming*, vol. 33, no. 2, pp. 101-131, 1999.

Coplien, Jim, "Why Patterns Are Different", http://www.linuxcare.com.au/mirrors/c2wiki/WikiPagesAboutWhatArePatterns/WhyPatternsAreDifferent.html

DeRemer, F. and Kron, H., "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, vol. 2 pp. 321-327, 1976.

Ducasse, S. and Tamar, R. Executable Connectors: Towards Reusable Design Elements. Jazayeri, M. and Schauer, H. 483-499. 1997. Springer-Verlag. Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)".

Ellsberger, J., Hogrefe, D., and Saram, A., *SDL, Formal Object-Oriented Language for Communicating Systems* London: Prentice Hall, 1997.

Esparza, J. and Nielsen, M., "Decidability Issues for Petri Nets - A Survey," *J.Inform.Process.Cybernet.*, vol. 3, no. 30, pp. 143-160, 1994.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley, 1995.

Garlan, D., Allen, R., and Ockerbloom, J. Architectural Mismatch or Why it's hard to build systems out of existing parts. 1995.

Garlan, D. and Shaw, M., "An Introduction to Software Architecture," in Ambriola, V. and Tortora, G. (eds.) *Advances in Software Engineering and Knowledge Engineering* World Scientific Publishing Company, 1993, pp. 1-40.

Genesereth, M. R. and Fikes, F. E. Knowledge Interchange Format, Version 3.0 Reference Manual. Logic-92-1. 1992. Stanford University.

Harel, D. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8[3], 231-274. 1987.

Harrison, W. and Ossher, H. Subject-Oriented Programming (A Critique of Pure Objects). 411-428. 1993. ACM Press.

Helm, R., Holland, I. M., and D., G. Contracts: Specifying behavioural compositions in object-oriented systems. 169-180. 1990.

Hoare, C. A. R., *Communicating Sequential Processes* Prentice Hall, 1985.

Hopcroft, J. E., Motwani, R., and Ullman, J. D., *Introduction to Automata Theory, Languages and Computation*, Second ed. 2001.

INMOS, *Occam 2 Reference Manual* Prentice Hall, 1988.

Jacobson, Ivar, Christerson, Magnus, Jonsson, Patrik, and Övergaard, Gunnar, "Object-Oriented Software Engineering. A Use Case Driven Approach." 1992.

KIF, "Knowledge Sharing Web Site", http://logic.stanford.edu/sharing/knowledge.html

Kramer, R. iContract—The Java Design by Contract Tool. 295-307. 1998. Los Alamitos, California, USA, IEEE CS Press.

Krasner, E. G. and Pope, T. S. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1[3], 26-49. 1988.

Lajoie, R. and Keller, R. Design and Reuse in Object Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. 1994. Montreal, QC, Canada.

Lamping, J. Typing the Specialization Interface. 201-214. 1993. ACM Press.

Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns* PWS Publishing Company, 1996.

Lieberherr, K., "NFA algorithms and AP algorithms", http://www.ccs.neu.edu/research/demeter/papers/boaz-jacm/NFA/

Lieberherr, K. J. and Patt-Shamir, B. Traversals of Object Structures: Specification and Efficient Implementation. NU-CCS-97-15. 1997. College of Computer Science, Northeastern University, Boston, MA.

Lopez, Christina, "D - A language famework for distributed programming." 1997.

Luckham, D., Kenney, J., Augustin, L., Vera, D., Bryan, D., and Mann, W., "Specification and analysis of system architecture using RAPIDE," *IEEE Transactions on Software Engineering*, vol. 21 1995.

Mairson, H. On Axiomatic Characterizations of CSP. 02254. 1989. Department of Computer Science, Brandeis Univerity, Waltham, Massachusetts.

Marshall, A. D. Programming in C, UNIX System Calls and Subroutines using C. 1999.

Martin, Jeremy. Malcolm. Randolph, "The Design and Construction of Deadlock-Free Concurrent Systems." University of Buckingham, 1996.

Mclennan , S. G., Roesler, A. W., Tempest, J. T., and Spinuzzi, C. I., "Building More Usable APIs," *IEEE Software*, pp. 78-86, 1998.

Meyer, B., "Applying Design by Contract," *Computer*, pp. 40-52, 1992.

Mikkonen, T. Formalizing Design Patterns. 115-124. 1998. Kyoto, Japan.

Nierstrasz, O. and Dami, L., "Component-oriented software technology," *Object Oriented Software Composition* Prentice Hall, 1995, pp. 3-28.

Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

Petri, C. A., "Kommunikation mit Automaten," *Schriften des IIM*, vol. 2 1962.

Pintado, X. and Junod, B., "Gluons: Support for software component cooperation," *Object Frameworks*, pp. 311-346, 1992.

Reussner, R. Dynamic types for software components. 1999.

Reussner, R. An Enhanced Model for Component Interfaces to Support Dynamic Adaption. 2000. Cannes, France.

Riehle, D. A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose. 97-1-1. 1997. Zürich, Union Bank of Switzerland.

Rudolph, E., Graubmann, P., and Grabowski, J. Tutorial on Message Sequence Charts (MSC'96). 1996.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design* Prentice-Hall, 991.

Schank, R. C., *Identification of Conceptualization Underlying Natural Language* Freeman, San Francisco: 1973.

Schmidt, H. and Reussner, R. Automatic Component Adaptation by Concurrent State Machine Retrofitting. 25/2000. 2000. Karlsruhe, Universität Karlsruhe, Department of Informatics.

Shaw, A. C., "Software descriptions with flow expressions.," *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp. 242-254, 1978.

Short, K. Component Based Development and Object Modeling. 1997. Texas Instruments Software.

Shu, J. and Liu, M. A synchronization model for protocol conversion. 1989.

SUN, "Java Tutorial", http://java.sun.com/docs/books/tutorial/index.html

Szyperski, C., *Component Software; beyond Object-Oriented Programming* Addison-Wesley, 1997.

Taylor, R. N., Medvidovic, N., Anderson, M. K., Whitehead, E. J., and Robbins, E. J. A Component- and Message-Based Architectural Style for GUI Software. 295-304. 1995. Seattle WA.

Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, J. E., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L., "A Component- and Message-Based Architectural Style for GUI Software," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390-406, 1996.

Thatte, S. Automated synthesis of interface adapters for reusable classes. 174-487. 1994.

Tiberghien, J. Course: Personal Communication. VUB Press.

Warmer, B. J. and Kleppe, G. A., *The Object Constraint Language : Precise Modeling With UML* Adison-Wesley, 1999.

Weaver, L. and Robertson, L., *Java Studio by Example* 1998.

Weber Systems Inc., *dBASEIII Users' Handbook* New York: Ballantine Books, 1985.

Wells, L. K. and Travis, J., *LabVIEW for Everyone: Graphical Programming Made Even Easier* New Jersey: Upper Saddle River, NJ :Prentice Hall PTR, 1997.

Wydaeghe, B. and Vanderperren, W., "PacoSuite", http://ssel.vub.ac.be/Members/BartWydaeghe/member_pacosuite.htm

Wydaeghe, B., Verschaeve, K., Westerhuis, F., and De Moerloose, J. Multi-level Component Oriented Methodology for Service Creation. IS&N. 2000. Athens, Greece.

Wydaeghe, B. and Westerhuis, F. Evaluation of state-of-the-art component composition tools. AIA-WP3.3T1D1. 29-12-2001b.

Yellin, D. and Strom, R. Interfaces, protocols and the semiautomatic construction of software adapteors. 10, 176-190. 1994a.

Yellin, D. and Strom, R., "Protocol specifications and component adaptors," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, pp. 292-333, 1994b.

Zaremski, A. M. and Wing, J. M., "Specification Matching of Software Components," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 333-369, 1997.