INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600
HIERARCHICAL CONNECTORS
A CONTRIBUTION TO SOFTWARE ARCHITECTURE

by

Jelena Ivanisevic

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

© Copyright by Jelena Ivanisevic 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-40744-6
Abstract

Hierarchy has always been an important property of software architectural descriptions regardless of the level of formality those descriptions have had. Specifically large and complex software systems are easier to comprehend when described hierarchically. Emerging architecture description languages use hierarchical decomposition of components to represent large systems. Now that software connectivity is getting more attention in the area of software architecture and connectors are becoming explicit and first class entities in architectural languages, connectors, as well as components, deserve to be expressed at different levels of abstraction.

In this thesis we use Wright, an architecture description language specifically designed to describe architectural connections, to define hierarchical connectors. Based on the notation and theory used for composite components, we present notation and theory for composite connectors. Describing hierarchical connectors opens new avenues of specifying and reusing complex interactions.

Keywords: Software Architecture, Architectural Description Language, Wright, Software Connectors, Hierarchical Connectors.
Acknowledgments

First of all, I would like to thank my supervisor, David Wortman. Four years of part time studies and work would have never resulted in this thesis if it was not for his constant encouragement and support. I highly appreciate his ability to not only feel and understand the materials presented in this thesis but also feel and understand the student behind it. Thank you for being a true mentor.

I also thank my parents and my mother-in-law who were always with me through all ups and downs of my graduate studies and work. Although oceans away, they were always with me when I needed them.

Finally and the foremost I want to thank my Ivan. There is a piece of his support and understanding in every page that follows.
# Table of Contents

1. INTRODUCTION .................................................................................................................. 1
   1.1 SOFTWARE ARCHITECTURE ......................................................................................... 1
   1.2 COMPONENTS, CONNECTORS, SYSTEMS ................................................................. 2
   1.3 ARCHITECTURAL FORMALISM .................................................................................. 3
   1.4 ARCHITECTURAL DESCRIPTION LANGUAGES .......................................................... 4
   1.5 WRIGHT ...................................................................................................................... 6
      1.5.1 Architectural Description in Wright ..................................................................... 6
      1.5.2 Wright Notation .................................................................................................. 7
      1.5.3 Analysis and Checking ....................................................................................... 13
      1.5.4 Why Wright is Right ........................................................................................ 13
      1.5.5 Why Wright is Incomplete ................................................................................. 14
   1.6 ABOUT THIS THESIS .............................................................................................. 14

2. HIERARCHY IN ARCHITECTURAL DESCRIPTIONS ......................................................... 15
   2.1 WHY IS HIERARCHY IMPORTANT? ........................................................................... 15
   2.2 HIERARCHY IN OTHER ADLs .................................................................................. 16
      2.2.1 UniCon ............................................................................................................... 16
      2.2.2 Rapide ............................................................................................................... 18
      2.2.3 Darwin ............................................................................................................. 20
   2.3 WHAT NEEDS TO BE DONE? .................................................................................. 21
      2.3.1 Composite Components .................................................................................... 21
      2.3.2 Composite Connectors ...................................................................................... 26
      2.3.3 Composite Ports/Roles ...................................................................................... 30
      2.3.4 Analysis and Checking Support ........................................................................ 30
   2.4 CONCLUSION ........................................................................................................... 31

3. COMPOSITE COMPONENT CASE STUDY ........................................................................ 32
   3.1 HIERARCHICAL SYSTEM DESCRIPTION .................................................................. 32
      3.1.1 Master Station / Composite Component ............................................................ 37
      3.1.2 Master Station's Configuration Behavior ............................................................ 40
      3.1.3 Discussion .......................................................................................................... 42
      3.1.4 Master Station's Computation ........................................................................... 43
      3.1.5 Discussion .......................................................................................................... 44
   3.2 CONCLUSION ........................................................................................................... 44
4. COMPOSITE CONNECTORS ....................................................................................................................... 46

4.1 NOTATION ............................................................................................................................................ 46
4.2 CONNECTOR CONFIGURATION ........................................................................................................... 48
  4.2.1 Connector Configuration Behavior ............................................................................................... 49
  4.2.2 Correct Connector Compositions ................................................................................................. 50
4.3 BEHAVIOR OF COMPOSITE CONNECTORS ....................................................................................... 50
  4.3.1 Behavior Computation Example ................................................................................................ 52
4.4 WHAT MAKES A COMPONENT PART OF A CONNECTOR? ............................................................... 56
4.5 USING CONNECTOR COMPOSITIONS .................................................................................................. 57
  4.5.1 Parallel Composition .................................................................................................................... 57
  4.5.2 Bus Connectors ............................................................................................................................ 58
4.6 WHY COMPOSITE CONNECTORS? .................................................................................................... 62
4.7 CONCLUSION ....................................................................................................................................... 63

5. CONCLUSIONS ........................................................................................................................................ 65

  5.1 SUMMARY ........................................................................................................................................ 65
  5.2 FUTURE WORK ................................................................................................................................ 66

REFERENCES .................................................................................................................................................. 68

APPENDIX A .................................................................................................................................................. 71

  WRIGHT PROCESS NOTATION REFERENCE ....................................................................................... 71
  MISCELLANEOUS DEFINITIONS ............................................................................................................ 72

APPENDIX B .................................................................................................................................................. 73

  FILE ACCESS CONNECTOR ..................................................................................................................... 73

APPENDIX C .................................................................................................................................................. 74

  COMPOSITE COMPONENT EXAMPLE ................................................................................................ 74

APPENDIX D .................................................................................................................................................. 80

  REMOTE PROCEDURE CALL EXAMPLE ............................................................................................... 80

APPENDIX E .................................................................................................................................................. 81

  EVENT BUS 2 .......................................................................................................................................... 81
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 1</td>
<td>Simple example of a system in Wright</td>
<td>6</td>
</tr>
<tr>
<td>FIGURE 2</td>
<td>File specification in Wright</td>
<td>10</td>
</tr>
<tr>
<td>FIGURE 3</td>
<td>Example of component definition in Wright</td>
<td>11</td>
</tr>
<tr>
<td>FIGURE 4</td>
<td>A Message Passing Connector</td>
<td>11</td>
</tr>
<tr>
<td>FIGURE 5</td>
<td>Composite Component - Three stage pipeline</td>
<td>16</td>
</tr>
<tr>
<td>FIGURE 6</td>
<td>Component Specification in UniCon</td>
<td>17</td>
</tr>
<tr>
<td>FIGURE 7</td>
<td>Primitive Component in UniCon</td>
<td>17</td>
</tr>
<tr>
<td>FIGURE 8</td>
<td>Three stage pipeline specification in Unicon</td>
<td>18</td>
</tr>
<tr>
<td>FIGURE 9</td>
<td>Interface Specification in Rapide</td>
<td>19</td>
</tr>
<tr>
<td>FIGURE 10</td>
<td>Pipeline Specification in Rapide</td>
<td>19</td>
</tr>
<tr>
<td>FIGURE 11</td>
<td>Primitive Component in Darwin</td>
<td>20</td>
</tr>
<tr>
<td>FIGURE 12</td>
<td>Composite Component in Darwin (Pipeline)</td>
<td>20</td>
</tr>
<tr>
<td>FIGURE 13</td>
<td>Composite Component Notation in Wright</td>
<td>22</td>
</tr>
<tr>
<td>FIGURE 14</td>
<td>Composite Module Example</td>
<td>22</td>
</tr>
<tr>
<td>FIGURE 15</td>
<td>Composite Component Specification in Wright</td>
<td>23</td>
</tr>
<tr>
<td>FIGURE 16</td>
<td>Parallel Compositions of Connectors</td>
<td>26</td>
</tr>
<tr>
<td>FIGURE 17</td>
<td>C-Fortran Procedure Call Connector</td>
<td>27</td>
</tr>
<tr>
<td>FIGURE 18</td>
<td>Remote Procedure Call Connector</td>
<td>28</td>
</tr>
<tr>
<td>FIGURE 19</td>
<td>Composite Interfaces</td>
<td>30</td>
</tr>
<tr>
<td>FIGURE 20</td>
<td>System Configuration</td>
<td>33</td>
</tr>
<tr>
<td>FIGURE 21</td>
<td>Master Station Protocol</td>
<td>34</td>
</tr>
<tr>
<td>FIGURE 22</td>
<td>Slave Station Protocol</td>
<td>35</td>
</tr>
<tr>
<td>FIGURE 23</td>
<td>Multidrop Connector</td>
<td>35</td>
</tr>
<tr>
<td>FIGURE 24</td>
<td>Dial Connector Specification</td>
<td>36</td>
</tr>
<tr>
<td>FIGURE 25</td>
<td>Master Station Configuration</td>
<td>37</td>
</tr>
<tr>
<td>FIGURE 26</td>
<td>The Mailbox Connector Specification</td>
<td>38</td>
</tr>
<tr>
<td>FIGURE 27</td>
<td>The Poll Controller Specification</td>
<td>38</td>
</tr>
<tr>
<td>FIGURE 28</td>
<td>Dial Controller</td>
<td>39</td>
</tr>
<tr>
<td>FIGURE 29</td>
<td>Master Station Composite Component</td>
<td>39</td>
</tr>
<tr>
<td>FIGURE 30</td>
<td>The PollerComputation Process</td>
<td>41</td>
</tr>
<tr>
<td>FIGURE 31</td>
<td>The DialerComputation Process</td>
<td>41</td>
</tr>
<tr>
<td>FIGURE 32</td>
<td>The PollerMboxGlue and DailerMboxGlue Processes</td>
<td>42</td>
</tr>
<tr>
<td>FIGURE 33</td>
<td>Master Station's Configuration - Projected Behavior</td>
<td>43</td>
</tr>
<tr>
<td>FIGURE 34</td>
<td>Master Station's Computation - Calculated Behavior</td>
<td>44</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Software Architecture

"Software systems have always had architectures but it is now that architectures became obvious and explicit." [SG94,95]

In the development of large and complex software systems, first steps in the software design process are the most important ones. Any omissions in initial design decisions become huge obstacles in later phases of software development and maintenance. Well conceived overall system structure provides a framework for satisfying initial and changing software requirements as well as firm basis for software design.

In the past, modularization provided by programming languages was the first step towards handling large and complex software systems by decomposing them into manageable pieces called modules. Over time, beyond the definition-use relationship\(^1\) of programming modules, many rich interaction mechanisms and patterns of organization have been recognized and reused in development of new software systems. Systems may be classified as client-server, layered, pipe and filter, blackboard, interpreter, etc [SG94]. These patterns of system organization, usually referred to as architectural styles, have served as a common vocabulary in the software engineering community when the architecture of a system is described.

Software architecture is a high level of software design concerned with global system properties such as system structure, performance, maintainability and evolution. It provides a firm basis for all phases of the software development life-cycle by being a framework for satisfying requirements, reasoning about alternative solutions, starting level of software design, reusing software organizations, reusing components of already developed systems, understanding changes and their consequences.

\(^1\) Modules DEFINE data and procedures that other modules USE. This relationship between modules is called definition-use relationship.
1.2 Components, Connectors, Systems

Software architecture describes an overall system structure as a collection of interacting components. In a client-server architecture, client and server components are interacting using remote procedure calls; in a pipe and filter organization filter components communicate via pipes. Concrete architectural elements such as clients, servers, filters, modules, files, databases, and processes are referred to as components. Components interact using interaction mechanisms such as procedure calls, database access, pipes, shared memory or queues. Component interaction mechanisms are called connectors.

In most system descriptions, components are the dominant architectural elements that carry out computation and hold state [SDK95]. Connectors are less obvious and therefore frequently overlooked architectural entities. Connectors are hidden in components as sequences of procedure calls (database access), system calls (queues, semaphores), communication controllers and their client libraries (event bus, database access through server), etc. By introducing connectors as first class entities, components carry out their computation without having to bother with the details of connectivity. Components focus on the functional properties of the system while connectors carry non functional, connective part of the system [SDK95].

Components can be primitive or composite. Primitive components are directly identifiable as objects in the system that need not be further decomposed or elaborated. Primitive components are often represented as source code, object code or data files. Composite components are composed out of other primitive or composite elements (component and connectors) of the system.

Connectors can be primitive or composite, symmetric or asymmetric. Examples of asymmetric connectors are procedure call connector that connects definer and user of the procedure, file access connector that connects file to its user, pipes that pass data from the pipe writer to the pipe reader. Examples of symmetric connectors are software bus connectors or event bus connectors; any component attached to the bus has the same interface and role in the bus communication.

Primitive connectors are connectors supported by programming languages or underlying system. For example, procedure call or data sharing are examples of the programming language supported connectors. Message queues, semaphores, pipes are some of the system provided connection services. Composite connectors are complex connection mechanisms specifically designed to facilitate communication between components in the system. Software bus connectors, event bus connectors,
database access connectors are examples of composite connectors. Composite connectors are composed out of other connectors and components.

1.3 Architectural Formalism

Highly informal "box and line" diagrams are still used by software designers to visualize the structure of the system and to analyze it [SG94,95]. In a box and line diagram, components are depicted as boxes or common graphical symbols that represent database storage, tables, processes, layers. Most critical and highly intuitive are lines that interconnect components in the diagram. The only distinctions between lines in the diagram are direction and number of interconnected components. Depending on what the components are, interconnections are interpreted as memory or database access, procedure call, pipes, etc.

Informal diagrams show a need for a formalism that will cover architecture specific needs. Although more formal, neither programming languages nor module interconnection languages have been able to fully express the rich architectural abstractions already used in practice [SG94,95]. For example, packages in Ada or rendez-vous in Ada are rich architectural abstractions provided by a programming language. However, the description of a software system has to be established and analyzed regardless of the programming language in use and not be limited to abstractions provided by a programming language.

Module Interconnection Languages (MILs) have been developed to isolate module connectivity from the modules themselves. However, the only components covered by MILs are modules and the only interconnections procedure calls and data use. That is too restrictive.

A rich vocabulary of components, connections and their patterns of organization needs to be provided by an architectural description notation. Components have to be represented in such a way that they do not encapsulate any interconnection specific knowledge. This allows components to be reused regardless of the context. Specific analysis needs to be supported on the architectural level of design such as performance analysis, deadlock freedom, type checking and the like. Both dynamic and static software configurations need to be described.

It became evident that none of the existing methods was able to address specific requirements of architectural descriptions. As the result of recognized need for a formal and highly specialized notations
to describe architectures of software systems, a number of architectural description languages has been developed.

1.4 Architectural Description Languages

Architectural Description Languages (ADL) are formal notations developed to specify and analyze architectural descriptions. Shaw and Garlan [SG95,94] list the following desirable properties of ADLs:

- **Composition** of architectural elements into a composite element; hierarchical system description
- **Ability to express** architectural abstractions
- **Promote reuse** of individual elements and their patterns of compositions
- **Support heterogeneity** of patterns in the same description, ability to decompose elements using different styles, support implementation in different languages and running on different machines
- **Facilitate analysis:** type checking, performance, throughput, deadlock, etc.

As result of the recognized need for more formal languages and notations for software architecture, a number of the ADLs have emerged: UniCon [SDK95], Wright [AG94a,b], Aesop [GAO94], Rapide [LK95], Darwin [MDK94], Olan [Bel95,96] etc. Table 1 shows comparison between the languages using some of the criteria specified by [SG95,94].

*UniCon's approach to system descriptions is highly pragmatic. UniCon supports explicit connectors but all connectors are built-in and new connections can not be easily added. Built in types of ports, roles and connector protocols constrain styles that can be used in system descriptions as well as style conformity analysis. UniCon can build executables out of architecture specifications provided some form of implementation for its primitive components. It also allows for 'compatible' (not only identical) component and connector interfaces to be connected.*

*Darwin's semantic base is Pi-calculus which permits support of dynamic configurations. Although the type of services or ports can be specified as stream or event, the interpretation is left to the tool that builds the executable. Darwin has no explicit support for connectors. Olan extends Darwin by supporting explicit connectors.*

*Rapide supports simulation of architectures of interacting components. The behavior of components is modelled using partially ordered set of events (posets). In simulating architectures, Rapid can use either components implementation or its behavior specification.*
More detailed description of UniCon, Darwin and Rapid specifically in respect to hierarchical architectural descriptions will be presented in Section 2.2.

<table>
<thead>
<tr>
<th>Requirement/ADL</th>
<th>UniCon</th>
<th>Wright</th>
<th>Rapide</th>
<th>Darwin</th>
<th>Olan</th>
<th>Aesop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composability/Hierarchy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Explicit Connectors</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Composite Connectors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inheritance</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Implementation/Code</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>Configuration(Dynamic/Static)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Different languages/machines</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multiple Views</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Simulation</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reuse Patterns/libraries</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Type Checking</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>✓</td>
</tr>
<tr>
<td>Extra Functional Properties</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Style</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Comparison of ADLs

In this thesis we have chosen to work with Wright because it recognizes connectors as first class architectural entities and provides a formal notation for specifying architectural connection. Wright specifies architectural interactions using protocol descriptions rather than predefined set of interaction types and interfaces. Being focused on architectural connectivity, Wright presents an excellent underlying formalism for what will become the central point of this thesis – hierarchical connectors.

We start with a brief description of Wright in paragraphs that follow.

---

2 The table of comparison uses the following two symbols:
✓ - supported by the language
? - questionable support or not enough research done by the author of this thesis.
1.5 Wright

Wright [RA97] is an architecture description language specifically designed to formalize architectural connection. Like other languages, Wright views architecture of the system as a set of interacting components. Unlike other languages, interactions in Wright are explicitly modeled using a connector abstraction.

Although initially designed to express architectural connection, Wright evolved into a language that allows for formal description of both architectural styles and architectural instances. Further, Wright defines tests of the internal consistency and completeness of those descriptions.

1.5.1 Architectural Description in Wright

In Wright, the software architecture of a system, called a configuration, is described using component and connector types, instances and attachments.

```
Configuration Coprocesses
  Component Process
  Port in <port protocol specification>
  Port out <port protocol specification>
  Computation <component computation specification>

Connector Pipe
  Role reader <role protocol specification>
  Role writer <role protocol specification>
  Glue <connector glue specification>

Instances
  P1, P2: Process
  p12, p21: Pipe

Attachments
  P1.out as p12.writer
  P2.in as p12.reader
  P2.out as p21.writer
  P2.in as p21.reader

End Coprocesses
```

Figure 1 Simple example of a system in Wright

Component types are defined as a set of ports and a component computation. Ports of components represent component's interface, "expectations and promises of the component into the points of interaction" [AG94b]. Ports are partial specifications of a component because they contain no knowledge about other interactions of the component with its environment nor how those interactions are combined to form a complete computation. A computation is a full specification of the component
behavior. It defines all component's "expectations and promises" to its environment by combining all interactions described by the ports.

Connector types are defined as a set of roles and a connector glue. The roles of a connector describe the behavior of each component as a participant in the communication and the glue describes how these are combined to form a communication. For example, the pipe connector described in Figure 1, describes an interaction between two components represented by the roles reader and writer of the pipe. The role of the reader component is to read data from the pipe until end of file is reached or reader closes the pipe. The role of the writer is to supply pipe with the data until it closes the pipe. The heart of the connector description is the glue. Its task is to coordinate reader and writer activities: reader can only read data supplied by the writer before the writer closes the pipe; supplied data are read in FIFO order; when all data are consumed, the reader is expected to close the pipe; both reader and writer can close the pipe at any point of communication.

Instances define instances of connectors and components that will compose the system. The system Coprocesses in Figure 1, comprises of two components P1 and P2, and two connectors p12 and p21.

Attachments define how components' and connectors' instances are connected together into a system. Ports of components are attached as roles of connectors to form a system. In the example above, all out ports of filter components are connected to writer roles of pipe connectors and all in ports of filters are connected to reader roles of pipes. This configuration of filters and pipe is well known in software architecture as a pipeline.

### 1.5.2 Wright Notation

Wright uses CSP process description to describe the glue, roles, computation and ports. Wright specifications are based on the idea that interactions between components are directly specifiable as protocols. Use of protocols in Wright is specific for two reasons [AG94a, AG94b]. First, protocols are not strictly used to describe an algorithm that needs to be followed by components. Second, communication between components, described by explicit entities, connectors, is modeled using separate role processes and a glue process, rather than just one connector process.
1.5.2.1 CSP Process Description / Wright Extensions of CSP

Processes are communication entities that can engage in events. Expression e->P specifies a process that engages in event e and then becomes process P. For example,

    Sleep = wakeup -> Awake

specifies that the process Sleep, when wake-up call occurs, becomes Awake.

When engaging in events, processes can make choices. Choices can be internal (non-deterministic) and external (deterministic). Internal choices are determined by the process based on its internal decision. For example,

    Sleep = wakeup -> (Awake ∪ Sleep)

means that the process Sleep, when wakeup occurs, can decide either to wake up or continue to sleep. External choices depend on the environment and not on the process itself. For example, a student can be described as

    Student = pass -> Driver ⊕ fail -> Student,

where the decision if a student passes driving exam and becomes driver comes from the environment not student itself.

Events can be assigned input or output data. For example, wakeup!time denotes that the event wakeup outputs time data; order?toppings means that the order event requires input data called toppings (for pizza). The event √ represents the success event. The process STOP is the simplest process that engages in no events. The sequence of the two, called successful termination, √ -> STOP in this thesis is presented as √ only.

---

3 This section outlines only basic notation of Wright and CSP. Wright process notation is summarized in Appendix A. Details of the Wright notation can be found in [RA97]. CSP details can be found in [Hoa85].

4 "becomes P" in this context is equivalent to "behaves like P".
Events can be observed or initiated by a process. Initiated events have an overbar (in this thesis underline) to be distinguished from observed events. For example, in the process that describes a student,

\[ \text{Student} = \text{pass} \rightarrow \text{Driver} \sqcap \text{fail} \rightarrow \text{Student}, \]

both pass and fail are observed events. However, in the process

\[ \text{Instructor} = \text{pass} \rightarrow \text{Instructor} \sqcap \text{fail} \rightarrow \text{Instructor}, \]

an instructor initiates both pass and fail event. This example also shows the typical use of internal choice with initiated events and external choice with observed events. This distinction between observed and initiated events does not exist in CSP.

More complex patterns of behavior are achieved in CSP by combining multiple processes using parallel composition operator \( (P \mid \mid Q) \). In parallel composition, all processes that are influenced by an event \( e \) have to be ready to engage in \( e \) before it occurs. That is, all processes have to agree on engagement into common events. Individual processes still remain in charge of events that are unique to them. For example, let us consider parallel composition \( P \mid \mid Q \) where \( P \) and \( Q \) be the following processes:

\[
\begin{align*}
P &= e \rightarrow f \rightarrow g \rightarrow P^5 \\
Q &= f \rightarrow h \rightarrow g \rightarrow Q
\end{align*}
\]

When \( P \) and \( Q \) are combined using parallel composition, both processes have to agree on common events \( f \) and \( g \) before they occur. Process \( Q \) could accept \( f \) first but \( P \) is not ready for it. Therefore, the first event to occur has to be \( e \). Further, both processes agree on \( f \). Since \( P \) can not proceed with \( g \) until \( Q \) is also ready for \( g \), \( Q \) accepts \( h \) and then both \( P \) and \( Q \) engage in \( g \). This sequence of events, named \( R \), occurs indefinitely:

\[ R = P \mid \mid Q \rightarrow e \rightarrow f \rightarrow h \rightarrow g \rightarrow R. \]

It is worth noting that processes that have no common events are not coordinated using parallel composition.

\[ ^5 e \rightarrow f \rightarrow g \rightarrow P \text{ is equivalent to } e \rightarrow (f \rightarrow (g \rightarrow P)). \text{ See Appendix A.} \]
Parallel composition is used in Wright to express semantics of connectors (Section 1.5.2.3), configurations (Section 2.3.1.1) and to express complex behaviors of both connectors and components (RTI connector in [AG97] or MixedComponent component in [AG96]).

1.5.2.2 Component

Component ports and computation are specified by CSP processes. Component computation is a process that describes the behavior of the component. For example, computation of the file component is a passive process (initiates no events) that expects the user of the file to open file first before any read/write activities start; it promises to provide any number of read/write activities before the user closes the file. As per file description in Figure 2, this cycle of events starting with open and ending with close event can be repeated any number of times.

```
Component File
    Port FileAccess = open -> Use []
                     where Use = read -> Use [] write -> Use [] close -> FileAccess

    Computation = FileAccess.open -> Use []
                  where Use = FileAccess.read -> Use
                     FileAccess.write -> Use
                     FileAccess.close -> Computation
```

Figure 2: File specification in Wright

The port processes are independent processes that define the expected local behavior of a component at the point of interaction. Port is actually a projection of the full computation into the events that are related to that specific interaction [RA97]. For example, the file component example has one port only. Therefore, the entire computation of the component is actually represented in the single point of interaction. That is, port and computation of the file component specify the same behavior.

If a component has more than one port, then its computation process describes how events of the ports are combined together. For example, a module specified in Figure 3, uses two files: FileA and FileB. The computation describes that the component first opens FileA and then FileB, reads from FileA and then writes to FileB some number of times and finally decides to close first FileA and then FileB. Note how ports see sequence of events in respect to two points of interaction: FileA and FileB. This is a projection of the computation behavior to the events related to the file interaction.
1.5.2.3 Connector

A connector is described as a set of role processes and a glue process. Roles are processes that describe the behavior of each component, participant, in respect to the communication. The glue process coordinates the behavior of the roles by coordinating and interleaving the events of the roles. For example, the MessagePassing connector has two roles: Sender and Receiver. Sender component constantly initiates send events, while the Reader component constantly observes receive events. Sender and receiver processes are constrained by the glue which specifies that each send event triggers receive event. Therefore, all messages sent by the sender will be received by the receiver.

The semantics of the connector in Wright is the parallel composition of all role processes and a glue process. All events in roles are re-labeled with the role name for the purpose of composition. For a connector with n roles \( R_i \) and role processes \( RP_i, i = 1..n \), the behavior of the connector is obtained by parallel composition of the glue and all the role processes as in the following CSP process:

\[
\text{Glue} \parallel (\parallel (\ldots (\parallel \text{Glue} \parallel \text{Role}_1 \parallel \ldots \parallel \text{Role}_n) \ldots))
\]
Re-labeling of role processes gives unique names to all events of connector roles. Also, all event names in the glue process contain role names to provide for unique naming in the glue description. When the glue and all roles of a connector are composed using a parallel composition, the only direct interaction is between the glue and each of the roles. For example, in Figure 4 the behavior of the connector is defined as

\[
\text{Glue} \parallel \text{Sender:Sender} \parallel \text{Receiver:Receiver}
\]

where

\[
\begin{align*}
\text{Sender:Sender} &= \text{Sender}.\text{send}!x \rightarrow \text{Sender:Sender} \\
\text{Receiver:Receiver} &= \text{Receiver}.\text{receiver}?x \rightarrow \text{Receiver:Receiver} \\
\text{Glue} &= \text{Sender}.\text{send}?x \rightarrow \text{Receiver}.\text{receive}!x \rightarrow \text{Glue}
\end{align*}
\]

According to rules of parallel composition, when \(\text{Sender}.\text{send}\) occurs, both \(\text{Glue}\) and \(\text{Sender:Sender}\) processes are engaged in the event. Further, the only event that can happen is the \(\text{Receive}.\text{receive}\) event expected by the \(\text{Glue}\) and \(\text{Receiver:Receiver}\) processes. This sequence of events can occur indefinitely. This simple example shows the function of the glue as coordinator between roles.

**Port-Role Attachment**

The port protocol takes the place of the role protocol in the actual system. An attached connector is defined by the processes of the attached ports instead of roles. Therefore, the behavior of an attached connector with roles \(R_i\) and attached port processes \(P_i\) where \(i = 1..n\) is the following parallel composition:

\[
\text{Glue} \parallel ( \parallel \ldots \parallel R_i : P_i )
\]

The beauty of Wright is that it does not require port protocols to be identical to the role protocols that they replace. Ports have to be able to "fulfill the interaction obligations" rather then behave identically [AG94a]. For example, port \(\text{FileA}\) can be attached to the following role:

\[
\text{Role FileUser} = \text{open} \rightarrow \text{Use where Use} = \text{read} \rightarrow \text{Use} \cap \text{write} \rightarrow \text{Use} \cap \text{close} \rightarrow \text{\top}
\]

The port \(\text{FileA}\) does not exercise the ability of the role to write to the file, but it otherwise fulfills the obligations of the role in the interaction.
1.5.3 Analysis and Checking

Architectural entities such as ports, roles, components, connectors, configurations all can be defined as CSP processes. This allows Wright to use CSP as a semantic base for analysis and checking of architectural descriptions. Many properties already defined in CSP such as process refinement or deadlock freedom are utilized in defining architecture checking in Wright. Some of the valuable checks that Wright performs are the following:

- Connector and role deadlock freedom
- Port-role compatibility
- Port-computation consistency.

Wright also benefits from adopting CSP notation for automating compatibility checking. Wright creates input for the commercial specification checking tool FDR [RA97] that is capable of handling all of the compatibility checks mentioned above.

1.5.4 Why Wright is Right

Wright is an architecture language that makes architectural interaction explicit, visible, and reusable by introducing connectors as separate semantic entities. Unlike UniCon that also supports explicit connectors and facilitates only a fixed set of connector types, any connector can be specified by Wright. Wright's unique treatment of interaction protocols separated in roles and glue facilitates checking of port-role compatibility (connector-component attachment compatibility). Formal connector theory facilitates reasoning about architectural interaction in isolation. By checking deadlock freedom of the connector, Wright guarantees that components attached to the connector will never deadlock in interaction carried out by the connector.

Static checking in Wright is a powerful tool in hands of the designer. Note that although Rapide also allows for component behaviors to be specified, no static checks regarding behavior are available. Finally, Wright architectures are checked using the automated checking tool FDR.

Although not discussed in more detail in this thesis, Wright also allows for the formal description of architectural styles.
1.5.5 Why Wright is Incomplete

Although Wright has evolved over time and satisfied many architecture language specific requirements, the following are at the moment major, yet not fully addressed issues:

- Configurations described in Wright are all static. Some of the work has been recently done to address dynamic configuration in Wright ([ADG98]).
- Hierarchy has been recently added to Wright but only for components. Connectors can not be composite.
- Wright is a specification language that can not be used to build executable systems. Behaviors of components are specified using process notation, not using their implementation.
- Most of the work in Wright is focused on interactions while the data aspect of architectural descriptions is completely neglected. In [RA97], the specification language Z is used in parallel to CSP to address this issue.

1.6 About This Thesis

In this thesis we have chosen to address some of the shortcomings of Wright and connector formalism in general to support hierarchical descriptions for both components and connectors.

In Chapter 2, "Hierarchy in Architectural Descriptions", we first describe how hierarchy is supported in architectural description languages such as UniCon, Rapide and Darwin. Further we discuss what is missing in Wright and other architectural languages with respect to hierarchy and how that can be addressed through Wright.

In Chapter 3, "Composite Component Case Study", we verify Wright's notation and theory for hierarchical descriptions of components through a detailed case study. This chapter bridges the gap between the bare theory and notation provided in [RA97] for composite components, and the complex theory and notation for hierarchical connectors presented in the chapter that follows.

In Chapter 4, "Composite Connectors", we introduce a notation and theory for composite connectors based on the formalism already available for components. We provide parallel definitions for connectors as well as examples to support them. We also argue why composite connectors are important in architectural descriptions.

Appendix A is a reference guide to Wright and CSP notation. Other appendices provide detailed examples referenced throughout the thesis.
2. Hierarchy in Architectural Descriptions

This chapter presents hierarchy as an important property of architectural descriptions. Most of the architectural languages already support hierarchy. In this chapter UniCon, Darwin and Rapide are used to demonstrate how systems are presently described in a hierarchical manner. Further, Wright is used to demonstrate what is missing in hierarchical architectural descriptions today and how that can be addressed using Wright as a language that supports both components and connectors as first class entities.

2.1 Why is Hierarchy Important?

Hierarchical decomposition has always been present in architectural descriptions regardless of the level of formality those descriptions supported. A system has always consisted of parts, which in turn have their own parts, etc.

The design of large and complex systems requires that their description be presented at various levels of abstraction. Each of the parts presented in a high level description of the system can be gradually refined and decomposed into manageable pieces in lower level descriptions. For example, for a client-server system consisting of a number of servers and clients, a high level description of the system has to depict the architecture as a set of interacting clients and servers and completely hide their implementation details. Each client and server can be further decomposed as a subsystem of the initial complex system.

Hierarchical system descriptions increase comprehension of a complex large-scale system by suppressing unnecessary details at the higher levels and revealing details at the lower levels of description. It also provides for reasoning about the system at different levels of abstraction. When a system description supports different levels of abstraction, both lower, primitive, as well as high level patterns of organization can be reused; refinement methods can be reused as well. Different levels of architectural
descriptions may be implemented using different architectural styles. Correct refinement is one of the main issues to be resolved when decomposing system into subsystems.

2.2 Hierarchy in Other ADLs

Hierarchy has been accepted by most of the architecture description languages such as UniCon, Rapide, Darwin, etc. Since the explicit connector theory is not supported by most of the languages, hierarchical decomposition is predominantly (only) supported for components. Composite connectors have not been yet described.

The following sections describe support for hierarchical descriptions as implemented by UniCon, Rapide and Darwin. Along with general description of the features, a common example shown in Figure 5 will be used to illustrate the similarities and differences in the languages. The intention is to gather experience built in other languages before the hierarchy in Wright is presented, analyzed and enhanced.

![Figure 5: Composite Component - Three stage pipeline](image)

2.2.1 UniCon

A system in UniCon [SDK95] is composed of component and connector instances glued together. The highest level component is the actual system description.

Components and connectors in UniCon are symmetric constructs. Both constructs have the specification and implementation (Figure 6). Components are specified by interfaces and connectors are specified by
protocols. A component interface has a type\(^8\) and a list of players. Players in UniCon are analogous to ports in Wright and represent component interaction points with the outer world (other components). Each player has a list of properties that further describe the component’s interface.

```
Component SpaceFilter
  Interface is type Filter
    Player in = StreamIn
    Player out = StreamOut
  End interface

  Implementation is
    /* specify file that implements filter binary, source or object */
  End Implementation
End SpaceFilter
```

**Figure 6:** Component Specification in UniCon

Each connector has a protocol specification. Like Wright, UniCon connectors specify roles to be satisfied. The connector protocol defines interaction properties including the rules for players that can match roles of the connector.

```
PrimitiveFilter
```

**Figure 7:** Primitive Component in UniCon

Implementation of components and connectors can be primitive or composite. Primitive components are directly specifiable as files that implement them: binary files, source code files, object files (Figure 7), etc. Composite components are constructed out of components and connectors. To illustrate a composite component specification in Unicon, the three stage pipeline example is specified in Figure 8.

```
```

**Figure 7:** Primitive Component in UniCon

Primitive connectors are built-in constructs and can not easily be specified by the user. Composite connectors are not supported by UniCon.

UniCon performs various component/connector compatibility checks such as: whether the player satisfies the role, type checking of connectors and components, etc. However, the major weakness of UniCon is that all of the connectors as well as the role/player types are built into the language.

---

\(^8\) The component interface type is used for type checking.
**Component Pipeline**

**Interface** is type Filter

- **Player in** = StreamIn
- **Player out** = StreamOut

**End interface**

**Implementation is**

/* Part Instances *************/

- Uses F1 interface Filter1
- Uses F2 interface Filter2
- Uses F3 interface Filter3
- Uses p12 protocol Stream
- Uses p23 protocol Stream

/* Bind Abstractions *************/

- Bind in to F1.in
- Bind out to F3.in

/* Connect Instances *************/

- Connect F1.out to p12.input
- Connect F2.in to p12.output
- Connect F2.out to p23.input
- Connect F3.in to p23.output

**End Implementation**

**End Pipeline**

---

**Figure 8:** Three stage pipeline specification in Unicon

---

**Summary of UniCon features**

- Composite components are comprised of an interface and an implementation; the implementation is described through instances of components and connectors glued together into a system or yet another component.
- Composite component players are defined using the binding abstractions to constituent component players.
- UniCon explicitly supports connectors but connectors are only primitive and built-in.

---

### 2.2.2 Rapide

Rapide [LK95] is an event based, concurrent object-oriented language specifically designed to prototype architectures of distributed systems. In Rapide, a system component consists of two parts: an interface and a module that is either an executable prototype or another system represented as an architecture. At a minimum, a module satisfies an interface if it defines all items declared in the interface.
An interface defines features provided to other modules (public and private) and required from the other modules (external). Complex interfaces are structured into sets of related constituents called services. Interface has usually three sections: declarations, constraints and behavior (Figure 9). The constraints specify the visible behavior of modules. The behavior of a module is described as a set of processes that observe events and react to them by generating new events. The behavior specification is used when no module is defined to construct a module and when the module is defined to check against its behavior.

```plaintext
interface <interface type> is interface
  public |private |extern
  <interface point declaration >
  constraint
    <algebraic and pattern constraints >
  behavior
    <behavior of the module in execution language >
end <interface type>;
```

Figure 9: Interface Specification in Rapide

An architecture is a set of components and set of connections between the components' interfaces. Components are connected by static and dynamic connections. Connections define event flow between interfaces. The three stage pipeline specification shown in Figure 10 gives an example of an architecture specification in Rapide.

```
architecture Pipeline return Filter is
  F1, F2, F3: Filter;
connect
  F1.out to F2.in;
  F2.out to F3.in;
end Pipeline;
```

Figure 10: Pipeline Specification in Rapide

Mapping provides for the definition of relationships between pairs of architectures or between systems (how events in one system depend on events in another). This facilitates conformance checks to reference architectures and consistency for different levels of architecture.

Summary of Rapide features
- Complex interfaces are grouped into composite interfaces
- Interface is separate from implementation
- Interface can replace a module in simulation or be used against the module for checking whether it satisfies it
- Mappings are used to map events from one system to events in another reference system
Note that interfaces in Rapide resemble Wright component interfaces. They are comprised of observed and generated events and described as a set of concurrent processes that react to events.

2.2.3 Darwin

Darwin [MDK94] is a language designed to describe architectures of distributed systems. Darwin describes an architecture of a distributed system as a collection of interacting components. Component interface contains a set of component interaction points, called services. Services can be required and provided depending on whether the component implements or acquires service from other components in the system.

```
class Filter {
    provides out < stream char >;
    requires in < stream char >;
}
```

**Figure 11:** Primitive Component in Darwin

Primitive components are computational units described using an interface. Composite components are described by an interface and a collection of component instances and their bindings. Figure 11 shows a filter as a primitive component while the three stage pipeline specification in Figure 12 shows how a composite component is specified in Darwin.

By allowing components themselves to be constructed of interconnected components, Darwin supports hierarchical system structures. In terms of the description of composite components, UniCon and Darwin have very similar approaches. The only major difference is the lack of explicit connector theory in Darwin.

```
class Pipeline {
    provides out < stream char >;
    requires in < stream char >;

    inst F1, F2, F3:Filter;

    bind
        F1.in -- in;
        F2.in -- F1.out;
        F2.out -- F3.in;
        F3.out -- out;
}
```

**Figure 12:** Composite Component in Darwin (Pipeline)
Summary of Darwin features

- Hierarchical component descriptions (similar to UniCon)
- Dynamic structures can be described

2.3 What Needs to be Done?

As presented in the previous section, most of the existing architecture languages support hierarchical system structure i.e., most allow for components to be composed out of other components and connectors where connectors are explicit. However, connectors, even when explicitly supported, are considered primitive and built-in. For example, UniCon's approach to connectors is mainly based on the built-in types of roles, ports and primitive connectors. The only language that gives us more freedom in defining architectural connection is Wright.

Wright has support for hierarchical system descriptions but yet not hierarchical connector descriptions. In the sections that follow, we describe Wright's support for hierarchy and identify what is missing in the descriptions of composite components, composite connectors, and composite ports/roles.

2.3.1 Composite Components

Initially, systems or configurations in Wright were flat structures comprised of component and connector instances. Recently, in [RA97], the Wright authors introduced hierarchical descriptions of systems by allowing components themselves to be described as configurations or subsystems. The description adopted by Wright is analogous to the one already supported by UniCon and Darwin.

The notation used to specify composite components in Wright is presented in Figure 13. The composite component is specified using a configuration specification. The configuration specifies instances of components and connectors that comprise the composite component. Attachments of the configuration describe how components and connectors are glued together. The ports of the constituent components are mapped to the composite component ports using bind statements.

In Wright, the computation of the composite component is either a configuration or protocol specification but not both. The configuration fully describes a composite component; all ports of the composite component are implemented as ports of its constituent components; the computation of the composite component is defined by the behavior of the configuration [RA97].
Let us consider a simple example that demonstrates hierarchical description of components in Wright. The composite component shown in Figure 14 is a module that comprises of three other modules. Since each of the constituent modules requires file access, the interface of the composite module comprises of three FileAccess ports. File access is described using the connector FileAccessConnector in Appendix B.

The composite module is specified using a configuration specification. The configuration has three modules: M1, M2, and M3, each defining one File port (Figure 15). File ports of the constituent modules are bound to the FileAccess ports of the outer component using bind statements. The role of the bind statements is similar to the attachment statements because both type of statements perform event renaming. Bind renaming facilitates renaming of events by replacing the constituent component instance name with the name of the outer component port. For example, "M1.File to FileAccess(1)" denotes that all event names of the form M1.File.e are renamed to FileAccess(1).e. This is important because component type specifications require events to be of the form PortName.e.
Component CompositeModule
Port FileAccess(1..3) = FileUserProtocol
Computation
Configuration
Component Module1
Port File = FileUserProtocol
Component Module2
Port File = FileUserProtocol
Component Module3
Port File = FileUserProtocol
Instances
M1:Module1; M2:Module2; M3:Module3
End
Bind
M1.File to FileAccess(1)
M2.File to FileAccess(2)
M3.File to FileAccess(3)
End Bind

Figure 15: Composite Component Specification in Wright

Note that UniCon or Darwin do not provide an alternative way to describe elements that are not primitive - primitive elements directly correspond to source codes, object files or executables. Abstract elements can only be described through compositions of primitive elements. It is important to notice that the hierarchical descriptions in Wright are not due to inability of Wright to describe composite elements on higher levels of abstraction.

Wright's notation is capable of describing a component or connector at any level of abstraction using their behavior specification. For example, in the AEGIS case study [AG96], MixedComponent is described as a parallel composition of client and server processes. As an alternative, MixedComponent could be composed of multithreaded client and server components. Or, our pipeline computation could have been described as a process that shows how composite's in and out ports are related instead of breaking it into multiple filters (that finally express the same relation between in and out ports). Therefore, hierarchical system descriptions in Wright are an additional, alternative way of describing large and complex structures rather than the only way to describe them.

2.3.1.1 Configuration Behavior
Along with the composite component notation briefly presented in the previous section, Allen in [RA97] provides more theory related to composite components and their behavior. This theory is used for analysis and checking of Wright descriptions. Since the notation describes composite components as configurations, Allen first defines configuration behavior as a function of behaviors of its constituent
component and connector instances. Further the theory describes how composite component’s computation can be derived from the configuration behavior.

The behavior of the configuration depends on the glue processes of the connectors and the computations of components. Informally, the behavior of the configuration is the parallel composition of component computations and all connector glue processes renamed according to attachment functions. Configuration behavior is formally defined in Wright [RA97] in the following way:

Definition (Configuration behavior). The behavior of the configuration is CSP process

\[
( \big| \big| i : (1..n), \ Cpi : Cpi \big| \big| ( \big| \big| j : (1..m), \ \forall i \ ( Cni : Cni ) ))
\]

where configuration declares

- component instances \( Cpi : Cpi \ldots Cpi : Cpi \), where each component type \( Cpi \) has computation process \( Cpi \),
- connector instances \( Cni : Cni \ldots Cni : Cni \), where each connector type \( Cni \) has glue process \( Cni \),
- attachment declarations with attachment functions \( R1 \ldots Rk \) (\( R = R1 \circ \ldots \circ Rk \))

Note that the behavior of roles, glue, ports and component computations are processes related to component and connector types, not instances. In order to provide for unique event names within the configuration, all component events have to be labeled using instance names. Also, connector glue events are first renamed using connector instances and further renamed using attachment renaming. This is because attachment renaming refers to roles of connector instances and ports of component instances.

In the configuration definition, \( Cpi : Cpi \) denotes instance renaming\(^9\) of component computation processes. Similarly, \( Cni : Cni \) denotes instance renaming of connector glue processes. Further, according to attachment renaming\(^{10}\), events in glue processes are renamed using ports of component instances. The later renaming facilitates the actual interaction and coordination of components using glues of connectors - events generated by components are observed by the glue processes of connectors.

\(^9\) Instance renaming - formal definition provided in Appendix A
\(^{10}\) Attachment renaming - formal definition provided in Appendix A
and cause action on other components attached to the same connector. Connector glue becomes the true moderator of component interaction.

2.3.1.2 Composite Component Computation

Wright authors ([RA97]) use a CSP function projection to hide all events of the process not relevant to the computation. The projection of the process $P$ to the event set $S$, $P |^* S$, is a process that hides all events of the process $P$ that do not appear in $S$. Refer to [RA97] or [Hoa85] for a formal definition of projection and hiding.

For example, a process $P = e \rightarrow f \rightarrow g \rightarrow P$ projected onto $S = \{e, g\}$ is a process $P' = e \rightarrow g \rightarrow P'$. More complex situation is would be $P = e \rightarrow f \rightarrow P \triangleright g \rightarrow h \rightarrow P$ projected onto $S = \{f, h\}$ is $P' = f \rightarrow P' \triangleright h \rightarrow P'$. Once $e$ and $g$ are hidden, the choice between $f$ and $h$ becomes internal. An important feature of projection is that it maintains the relationship of the events and the behavior of the process with respect to these events.

Bindings in the composite component definition are mapping ports of constituent components to the ports of the composite component (bind renaming). Each binding defines a keeping event set as a set of events exposed at the port of the composite component. Events of ports not specifically bound to the ports of the composite component are not relevant to the composite component computation and need not appear in the computation.

The computation of a composite component can be formally expressed in the following way ([RA97]):

\[ \mathcal{R}(\text{Conf} |^* S) \]

where composite component declares

- a set of ports $P_i$, $i=1..n$
- a computation as a configuration with the behavior Conf,
- bind declarations with bind renaming functions $\mathcal{R}_1...\mathcal{R}_k$ ($\mathcal{R} = \mathcal{R}_1 \circ ... \circ \mathcal{R}_k$), and
- keeping event set $S = S_1 \cup ... \cup S_n$ where $S_i$ are keeping event sets of the component ports $P_i$. 

---

25
Composite Components: What needs to be done?

The complex theory presented by Allen is lacking a detailed hierarchy case study. We present a study in Chapter 3 with the following tasks in mind: first, a system needs to be hierarchically described using component decomposition; second, a component's configuration could be chosen to specify configuration behavior; third, based on the configuration behavior, the composite component's computation could be calculated. This scenario would exercise all the theory defined for composite components in [RA97].

2.3.2 Composite Connectors

Connectors as first class entities are relatively new to the field of software architecture. Composite connectors are therefore an issue that has had little chance to be developed in the existing architectural languages. All connectors in UniCon are primitive and hard-coded into the language. Wright connectors allow for any kind of connector to be described using roles and glue specification. However, none of the languages allows for simple composition of connectors.

For a long time, interaction properties were hidden in the computation of components. Connectors are used to capture the interaction between the components, to make it explicit and visible. Also, hierarchy is used for components to ease the understanding of a system by allowing high level components to be viewed as compositions of components and connectors. Interfaces of composite components are implemented as interfaces of constituent components. Computation of a composite component is derived from the computation and interaction of its sub-components.

![Figure 16: Parallel Compositions of Connectors](image-url)
Similarly to components, we can build a connector out of components and connectors. Why define composite protocols of interaction or, vice versa, decompose interaction into manageable pieces? The first step towards answering this question is getting a better feeling for what a composite connector might be.

An easily justifiable case is a composite connector composed of a set of connectors only. Roles of the constituent connectors are mapped to the roles of the composite connector. For example, a math library is accessed by a module (Figure 16). There are many independent procedure calls from the module to the library. At the high level of abstraction, it is sufficient to state that the module and the library interact through the composite procedure call connector.

This method of composition is equivalent to a number of non-interacting components grouped into a composite component. For example, mathematical libraries are grouped because they provide similar services and have the same users. However, it is not always the case that components can be connected using independent set of connectors. Similarly, the composite connector interactions can not be split into independent sets of interactions i.e. represented as a parallel composition of connectors.

As software systems are getting large and complex, heterogeneity in underlying platforms that system runs on and languages that implement system component are inevitable. Connectors that bridge the differences in underlying platforms and programming languages are an important class of complex interaction mechanisms. For example, Figure 17 shows a procedure call connector that supports a caller in C and a callee in Fortran.

![Figure 17: C-Fortran Procedure Call Connector](image)
Let us consider a class of connectors related to maintaining properties of 'locally' available connectors when interacting with remote machines. For example, remote procedure calls, 'remote' pipes (Windows NT), remote database access, etc. These are sometimes built in the operating system and can be therefore considered 'primitive'. However, there are cases where some these connectors have to be added explicitly to the system when they are not supported. The idea is to allow the designer to use locally available primitive connectors and their well known properties for interactions with remote components. Typically, this type of connectors can be built out of local connectors on both the local and remote end along with some communication link between the local and remote machine.

For example, a remote procedure call is usually implemented as a local procedure call to a client stub that provides data transformation and communication link handling to the server stub that finally invokes a procedure implemented on the server end; the remote procedure delivers results which are communicated back to the client. This pattern of interaction is typical for remote procedure call implementation and is worth capturing, analyzing and reusing. For most of the applications, an RPC call connector can be treated as primitive and not further decomposed. The remote procedure call connector is illustrated in Figure 18.

Figure 18: Remote Procedure Call Connector
When is the decomposition necessary? Why composite connectors?

As for components, it is up to the designer to introduce levels of granularity to the description of the system. If a component or a connector is a well-known or primitive element, then it need not be further decomposed. Examples of primitive elements are procedure calls, data accesses, remote procedure calls for connectors; or a component that is built in the system as a whole and need not be described in details other than its interfaces and calculation in respect to the interfaces. If a designer feels that the audience needs to understand an element or build an element out of its parts then that element is further decomposed. So the language needs to facilitate composition of new elements out of existing ones and it is to designer's discretion when and how these will be used.

Composite Connectors: What needs to be done?

In order to introduce composite connectors to Wright, the following needs to be done:
- Define a notation that allows for composite connectors to be specified
- Define a method to specify composite connector behavior
- Provide examples of composite connectors using the outlined theory

We will take up these tasks in Chapter 4.
2.3.3 Composite Ports/Roles

Since all elements of the architecture, both connectors and components, are composable, it is worth considering whether the actual interfaces of components and connectors, ports and roles, need to be composable as well. Interface points, if grouped into a small group of interfaces (Figure 19), can result in a crisp description of the architecture. Grouping of interfaces is also already present in Rapide.

![Composite Interfaces Diagram]

Figure 19: Composite Interfaces

2.3.4 Analysis and Checking Support

The Wright authors have explicitly defined the behavior of a configuration and a computation of a composite component. The behavior of the configuration is the process composed out of computations of constituent components and the glues of constituent connectors. Further, Wright specifies the computation of the composite component as a behavior of the configuration projected onto the set of events used by the composite component's ports. Wright authors also define when a role or a port can be unattached in a configuration and when a port is consistent with the computation of the component.

Similar definitions can be derived for composite connectors assuming that they also represent a variant of configuration. What is the glue of a composite connector? When can a role be left unattached? Answers to these questions will be found in chapters that follow.
2.4 Conclusion

Hierarchy is an important property of architectural descriptions. Hierarchy in architectural description languages such as UniCon, Darwin and Rapide is based on the hierarchical decomposition of components. Connectors have recently become first class entities in architectural descriptions (Wright, UniCon) and, just like components, deserve to be presented at different levels of abstraction.

Wright supports connectors as first class entities but does not provide means for describing them hierarchically. The introduction of composite connectors to Wright, a notation, a composite connector behavior specification, and examples of composite connectors using the theory outlined are required. Hierarchical system descriptions using composite components and a theory related to their behavior is already described by Allen in [RA97], but it has never been validated through examples and case studies. Specifically in the context of this thesis, the validation of Allen's theory through a case study provides a firm ground for understanding the notation and theory required for composite connectors.

Detailed study of existing support for hierarchical descriptions in Wright will be described in Chapter 3. A solution to hierarchical connector descriptions in presented in Chapter 4.
Hierarchical system descriptions are a new but expected extension of Wright. Allen in [RA97] explains how systems can be described by hierarchically decomposing components. The decomposition of components is implemented using Wright configurations. Along with the notation that specifies composite components, Allen defines the behavior of the configuration and describes how a composite component's computation can be derived from the configuration's behavior. However, the bare theory outlined in [RA97] is lacking a detailed hierarchy case study to support its applicability and verify it.

The intention of this chapter is to provide a hierarchical system description using Wright and exercise the following theory outlined in [RA97]:

- Hierarchical System Description using Composite Components
- Composite Component Configuration Behavior
- Composite Component Computation

The results of this study will prepare the stage for what is coming - hierarchical description of connectors outlined in the chapter that follows.

We first describe a system as a configuration of components and connectors. Further, we select one of the components to be further decomposed. The decomposed component's configuration is used to demonstrate the configuration behavior. Finally, the configuration behavior is used to calculate the computation of the composite component.

### 3.1 Hierarchical System Description

The system described in this study is a simplified version of a large class of systems that are comprised of a number of stations being polled by the master station; the master station that is responsible for collecting messages from stations and transmitting messages to the remote server; and the server that performs processing of station requests. Stations can be terminals or sensors that occasionally have messages that need to be processed by the server. Local communication between the master station and slave stations is implemented in this example as a variation of the poll/select protocol. Communication with the remote server is through the dial line.
Multidrop Connector. The master station polls slave stations one by one. When a slave station has a message to send to the server, the slave station responds with the message to be transmitted. In this case, the master station continues to poll when the response is obtained from the server. Otherwise, the slave station rejects the poll and the master station continues to poll. Note that slave stations have completely passive role in the communication; they never transmit before being explicitly "asked" to do so.

The role of the master station in the communication is elaborated through the MasterProtocol protocol\textsuperscript{11} in Figure 21. The role of a slave station in the multidrop communication is specified\textsuperscript{12} as the StationProtocol protocol in Figure 22.

\textsuperscript{11} Note that underlined events in the notation represent initiated events; observed events are not underlined. For example, poll event is initiated by the Master station while reject and ack events are observed events.

\textsuperscript{12} The complete Wright specifications along with inline comments are included in Appendix C.
MasterProtocol
Master protocol describes communication between the master station and slave stations. The master station polls \(1..n\) stations one by one. For simplicity stations are considered to be numbered from \(1..n\) and assumed to be consecutive numbers with no gaps. If a station has a message to be sent, it sends a message; otherwise it rejects the poll. The master station awaits for either the message or reject from the slave station.

```c
/*
MasterProtocol
Master protocol describes communication between the master station and slave stations. The master station polls \(1..n\) stations one by one. For simplicity stations are considered to be numbered from \(1..n\) and assumed to be consecutive numbers with no gaps. If a station has a message to be sent, it sends a message; otherwise it rejects the poll. The master station awaits for either the message or reject from the slave station.
*/

Protocol MasterProtocol(nStations:1..) = Poll(1) /* Start polling station 1 */
where
/* Poll i-th station */
Poll(i) = poll!i -> WaitForRequest(i)
    /* Initiate poll of the i-th station and wait for request message from the station (if any) */
Poll(nStations+1) = Poll(1) /* When polling reaches nStations +1 it becomes 1, the first station is polled again */
/* Wait for polled station to respond with request or reject the poll */
WaitForRequest(i) = sendMessage -> Response(i)
    /* Observe message from the station; if received, stop polling and wait for the response to come from the server */
[] reject -> Poll(i+1)
    /* Observe reject in case polled station has no message to send to the server */

[]√
    /* Observe successful termination in case station performs termination (this helps master not to deadlock when station terminates while being selected/polled) */
/* Deliver server's response to the i-th station */
Response(i) = select!i -> WaitForReply(i)
    /* Initiate select for the polled station and wait for the station to be ready to accept the response message from the server */
WaitForReply(i) = ack -> receivesendMessage -> Poll(i+1)
    /* Observe ack from the selected station and if ack received send message to the station */
[] reject -> Poll(i+1)
    /* Observe reject from the station, drop the response and continue to poll. */
[]√
    /* Observe successful termination in case station performs termination (this helps master not to deadlock when station terminates while being selected/polled) */
*/
```

Figure 21: Master Station Protocol
The glue of the Multidrop connector (Figure 23) has a simple function. Since it operates in parallel to the roles, it observes events on the master end and triggers events on the station end and vice versa. The sequencing of the events is fully driven by the protocols of the roles. There is no additional constraint introduced by the glue rather than simple event "mapping". This is not always the case. For example, in the connector Mbox, described later, the glue of the connector has crucial role in coordinating and constraining the behavior of the roles.
Dial Protocol. *DialConnector* (Figure 24) describes dial up communication between the master station and the server. When a message is collected from a station, the master dials the server, connects, sends a message or a number of messages and ends communication by disconnecting.

Note that *DialConnector* allows for multiple pairs or request-response messages to be communicated from the caller to the server before ending the call. It will be shown later that the master component in this example does not fully exploit the capability of the connector to transmit multiple messages to the server. It sends one message, collects the response and disconnects.

```
/*
   Dial Protocol
   Initiate connect and communicate or decide to stop
   Communicate sends a message x and observes response y; any number of
   request/response messages can be sent before the process decides to disconnect.
*/
Interface Type DialProtocol = connect -> Communicate \[ \Pi \] where Communicate = send!x -> receive?y -> Communicate
   \[ \Pi \] disconnect -> DialProtocol
   \[ \Pi \] \]

/*
   Host Protocol
   Observes connect event and enters communication with the caller.
   Communicate awaits request message from the caller and then sends response.
   Any number of request/response messages can be received/sent before
   the process accepts disconnect event generated by the caller.
*/
Interface Type HostProtocol = connect -> Communicate \[ \] \[ \] where Communicate = send?x -> receive!y -> Communicate
   [ ] disconnect -> HostProtocol
   [ ] \]

/*
   Dial Connector
   Roles are Caller and Host specified by the protocols DialProtocol and HostProtocol
   respectively. The glue process coordinates the communication of the roles. The
   glue in this case only observes connect, send, and disconnect of the Caller and
   initiates corresponding events for the Host. It also observes receive event of
   the host and triggers receive event of the Caller.
*/
Connector DialConnector
   Role Caller = DialProtocol
   Role Host = HostProtocol
   Glue
       Caller.connect -> Host.connect -> Glue
       [ ! Caller.disconnect -> Host.disconnect -> Glue
       [ ] Caller.send?msg -> Host.send!msg -> glue
       [ ] Host.receive?msg -> Caller.receive!msg -> .Glue
       [ ] \]
```

Figure 24: Dial Connector Specification
3.1.1 Master Station/Composite Component

The master station component can be divided into the following three components: the Poller, responsible for communication with stations; the Dialer, responsible for dialing the server and communicating messages from stations to the server and collecting responses for stations from the server; and theLogFile, a passive component that logs all incoming and outgoing request and response messages. For simplicity, the LogFile component will be initially omitted from the configuration.

![Diagram](image)

**Figure 25: Master Station Configuration**

Dialer and Poller communicate through a set of mailboxes, namely PollerMbox and DialerMbox. Since only one request message and one response message reside inside the master station at any one time, these two mailboxes are sufficient to handle the communication. Note that in a more realistic situation, where true polling is conducted and multiple request and response messages reside in the master station, either a pool of mailboxes or a queue is required to handle the communication.

The mailbox connector has two rules: In for the mailbox depositor and Out for the mailbox owner. The owner can check if the mailbox is full or empty and receive a message from the mailbox in the former case. The depositor can only deposit messages. In this implementation, the deposit overrides any previous "unread" messages. Wright specification of the message mailbox is shown in Figure 26.
The PollController polls stations one by one. If a station has a message to send, it deposits the message to the dialer mailbox and waits for the response message. The response message gets deposited to the PollerMbox. The response is passed to the currently polled station by selecting the station, waiting for the acknowledgment and finally passing the response to the selected station. Following the successful delivery of the message, the Poller continues to poll.
The DialController constantly checks its mailbox. If the dialer's mailbox has a message, the dialer connects to the server, sends the message and waits for the response. After successfully receiving the response from the server, the dialer disconnects and deposits the response message into the Poller's mailbox. The CallProtocol used to describe the port of the DialController is a variation of the DialProtocol with only one pair of request-response messages being exchanged with the server. Also, MboxDepositor and MboxOwner are the protocols identical to the protocols of the Mbox's roles in and out respectively.

```
Component DialController(nStations:1..)
  Port Host = CallProtocol
  Port MboxPoller = MboxDepositor
  Port MboxDialer = MboxOwner
  Computation = MboxDialer_check ->
    ( MboxDialer.nomsg -> Computation
      [ ] MboxDialer.receive?x -> Host.connect ->
        Host.send!x -> Host.receive?y ->
        Host.disconnect -> MboxPoller.deposit!y -> Computation)
```

**Figure 28: Dial Controller**

Now that all types of components and connectors required to construct the composite component have been defined, the configuration of the master station can be specified as in Figure 29.

```
Component MasterStation(nStations:1..)
  Port Stations = MasterProtocol(nStations)
  Port Host = CallProtocol
  Computation
  Configuration
    Component PollController...
    Component DialController...
    Connector Mbox ...
  Instances
    Poller: PollController(nStations)
    Dialer: DialController(nStations)
    PollerMbox: Mbox
    DialerMbox: Mbox
  Attachments
    Dialer.PollerMbox as PollerMbox.in
    Poller.PollerMbox as PollerMbox.out
    Poller.DialerMbox as PollerMbox.in
    Dialer.DialerMbox as PollerMbox.out
  End Configuration
  Bind
    Poller.Stations to Stations
    Dialer.Host to Host
  End Bind
End MasterStation
```

**Figure 29: Master Station Composite Component**

As specified in the "Instances" section of the specification, the configuration is comprised of instances of the PollController component, the DialController component and two instances of the Mbox connector.
The "Attachment" section of the specification shows how component and connector instances are glued together in this configuration. The attachments are replacing roles of connectors with ports of components. After attachment, each connector is represented in a configuration by the glue only; instead of roles, actual components are taking part in connector interactions.

Note that not all ports of the constituent components are attached. The remaining ports, namely Poller.Stations and Dialer.Host, are bound to composite component ports Stations and Host respectively. The configuration is complete for there are no unattached ports of components nor roles of connectors.

In this section, the master station component is defined as a configuration, a collection of component instances and attached connectors. While this section concentrates more on the behavior of individual components and connectors of the master station component, the section that follows concentrates more on the behavior of the configuration as a whole. Using Allen's definition [RA97], we define the behavior of this configuration based on the individual behaviors of attached components and connectors. It then becomes more obvious how components interact through connectors and how the total behavior of the configuration depends on both the behavior of the constituent components and glues of the connectors.

### 3.1.2 Master Station's Configuration Behavior

The behavior of the master station's configuration can be calculated according to the configuration behavior definition defined in Section 2.3.1.1. First, all configuration instances need to be identified. Second, all instance renaming of events is performed to provide for unique event names within the configuration specification. Third, attachment renaming is performed according to the configuration attachment definitions. Finally, configuration behavior is expressed as a parallel composition of component computations and connector glue processes after instance and attachment renaming.

Configuration component and connector instances are the following:

- **Poller:PollerController** (component instance)
- **Dialer:DialController** (component instance)
- **PollerMbox:Mbox** (connector instance)
- **DialerMbox:Mbox** (connector instance)
Instance computations of components \((C_{pi}:C_{pP})\) and glues of connectors \((C_{nj}:C_{nP})\):
- PollerComputation
- DialerComputation
- PollerMboxGlue'
- DialerMboxGlue'

Further, let \(PollerMboxGlue\) and \(PollerMboxGlue\)' be \(PollerMboxGlue\)' and \(DialerMboxGlue\)' after attachment renaming. Finally, the configuration behavior is the following parallel composition:

\[
Conf = \text{PollerComputation} || \text{DialerComputation} || \text{PollerMboxGlue} || \text{DialerMboxGlue}
\]

The process \(\text{PollerComputation}\), shown in Figure 30, is obtained by adding the prefix \(\text{Poller}\) to all events of the \(\text{PollerController}\) computation (instance renaming).

```
Process PollerComputation = Poll(1)
where
  Poll(i) = Poller.Stations.poll!i -> WaitForRequest(i)
  WaitForRequest(i) =
    Poller.Stations.send?msg -> Poller.MboxDialer.deposit!msg -> Response(i)
    [ ] Poller.Stations.reject -> Poll(i+1)
    [ ] \n  Response(i) =
    Poller.MboxPoller.check -> (Poller.MboxPoller.nomsg -> Response(i)
      [ ] Poller.MboxPoller.receive?msg ->
        Poller.Stations.select!i -> WaitForReply(i,msg)
    )
  WaitForReply(i,msg) =
    Poller.Stations.ack -> Poller.Stations.receive!msg -> Poll(i+1)
    [ ] Poller.Stations.reject -> Poll(i+1)
    [ ] \nFigure 30: The PollerComputation Process
```

The process \(\text{DialerComputation}\), shown in Figure 31, is obtained by adding the prefix \(\text{Dialer}\) to all events of the \(\text{DialController}\) computation (instance renaming).

```
Process DialerComputation = Dialer.MboxDialer.check -> (Dialer.MboxDialer.nomsg -> Computation
  [ ] Dialer.MboxDialer.receive?x -> Dialer.Host.disconnect
    [ ] Dialer.Host.receive?y ->
      Dialer.Host.send!x
    [ ] Dialer.MboxPoller.receive?y
    [ ] Computation)
Figure 31: The DialerComputation Process
```
The processes PollerMboxGlue and DialerMboxGlue, shown in Figure 32, suffered two rename stages:

1. instance renaming — adding the prefix DialerMbox (PollerMbox) to all events of the Mbox glue
2. attachment renaming — renaming all role names with the port names according to the attachment functions. For example, the prefix PollerMbox.in is replaced with PollerMbox.DialerMbox according to the attachment specification "Poller.DialerMbox as PollerMbox.in".

```
Process PollerMboxGlue = Empty
  where Empty = Poller.MboxPoller.check -> Poller.MboxPoller.nomsg
    [] Dialer.MboxPoller.deposit?x -> Full(x)
    Full(x) = Poller.MboxPoller.check -> Poller.MboxPoller.receive!x -> Empty
    [] Dialer.MboxPoller.deposit?x -> Full(x)

Process DialerMboxGlue = Empty
  where Empty = Dialer.MboxDialer.check -> Dialer.MboxDialer.nomsg
    [] Poller.MboxDialer.deposit?x -> Full(x)
    Full(x) = Dialer.MboxDialer.check -> Dialer.MboxDialer.receive!x -> Empty
    [] Poller.MboxDialer.deposit?x -> Full(x)
```

Figure 32: The PollerMboxGlue and DialerMboxGlue Processes

One important architectural property can be observed from the configuration behavior. Poller's and Dialer's interaction is mediated by the Mbox connectors. Poller and Dialer component instances have no common events. Both Poller and Dialer interact with the Mbox glue by engaging in common events with it\(^{13}\); while Dialer deposits message, Poller checks and receives message. Note also that the glue forces the sequence of events – message has to be deposited first before it can be consumed.

### 3.1.3 Discussion

When specifying composite component as a configuration, the contribution of individual components and connectors to the configuration behavior is well understood in isolation. Isolated pieces can be replaced with a minimal effect on the rest of the configuration. On the other hand, although pieces are well understood in isolation, composite component behavior as a whole might be difficult to understand. We need to express the behavior of the component in terms of events that appear on the component interfaces.

The component MasterStation has an interface comprised of the ports Host and Stations and a computation indirectly specified as a configuration. If we decided to specify the computation of the component directly, not through its configuration, it would not be specified as a parallel composition

\(^{13}\) Recall that in parallel composition processes synchronize on common events (Section 1.5.2.1).
Conf just calculated. This composite process has too many details not necessary for the computation of the component that has two ports with two sets of events that need to be coordinated in it. We need to somehow hide all events that are not relevant to the computation and not spoil the behavior with respect to the desired events.

3.1.4 Master Station's Computation

According to the hierarchy definition, the computation of the master station can be derived from the previously calculated behavior of the master station configuration. The "keeping event set" S of the master station configuration are all events of the form Dialer.Host.* and Poller.Stations.*. These events will be exposed to the outer ports of the composite component and all other events that appear in the configuration need to be hidden. The projected behavior onto the event set S is shown in Figure 33.

```
Process MasterStationComposite = Poll(1)
  where
  Poll(i) = Poller.Stations.poll!i -> WaitForRequest(i)
  WaitForRequest(i) = Poller.Stations.send?x -> Dialer.Host.connect ->
    Dialer.Host.send!x -> Dialer.Host.receive?y ->
    Dialer.Host.disconnect -> Response(i)
    [] Poller.Stations.reject -> Poll(i+1)
  [\]
  Response(i) = Poller.Stations.select!i -> WaitForReply(i,msg)
  WaitForReply(i,msg) =
    Poller.Stations.ack -> Poller.Stations.receive!msg -> Poll(i+1)
    [] Poller.Stations.reject -> Poll(i+1)
    [\]
```

Figure 33: Master Station's Configuration - Projected Behavior

According to bind renaming, Poller.Stations is replaced with Stations only and Dialer.Host to Host only and the resulting computation is as outlined in Figure 34.

The final calculation shows that the master station polls slave stations one by one. If a slave station has message to send, the master station dials the host, sends the message, awaits for the response and finally delivers the response to the slave station. The calculated behavior of the master station clearly outlines the behavior of the component with respect to its interfaces (ports) Stations and Host. All other details of the internal nature are hidden.
\[
\text{Process Composite} = \text{Poll}(1)
\]
\[
\begin{align*}
\text{where} & \\
\text{Poll}(i) &= \text{Stations.poll!i} \rightarrow \text{WaitForRequest}(i) \\
\text{WaitForRequest}(i) &= \text{Stations.send?x} \rightarrow \text{Host.connect} \rightarrow \\
& \quad \text{Host.send!x} \rightarrow \text{Host.receive?y} \rightarrow \\
& \quad \text{Host.disconnect} \rightarrow \text{Response}(i) \\
[] & \text{Stations.reject} \rightarrow \text{Poll}(i+1) \\
[\] & \text{Response}(i) = \text{Stations.select!i} \rightarrow \text{WaitForReply}(i, msg) \\
\text{WaitForReply}(i, msg) &= \text{Stations.ack} \rightarrow \text{Stations.receive!msg} \rightarrow \text{Poll}(i+1) \\
[] & \text{Stations.reject} \rightarrow \text{Poll}(i+1) \\
[\] & \checkmark
\end{align*}
\]

Figure 34: Master Station’s Computation - Calculated Behavior

3.1.5 Discussion

When calculating the behavior of the configuration that represents a composite component, all constituent components and their interactions are taken into consideration; the behavior of the configuration depends on individual behaviors of components and connectors and how their full internal specification contributes to the behavior of the composite component.

When specifying the computation of the composite component, we are interested only in the behavior that describes how interfaces of the component are satisfied. Only events that occur in component's ports need to appear in the specification of the component. When calculating the behavior of the master station, only those events that appear in the ports Stations and Host are used in the component specification. The way those events are combined to describe the component's behavior directly depends on the underlying configuration behavior. Although the intention of the decomposition is to pass the responsibilities of the composite component to its constituent components, putting elements together and verifying that the behavior of the composite component is as expected is also a power tool in hands of the designer.

3.2 Conclusion

The detailed case study presented in this chapter demonstrates the existing support for hierarchy in Wright. First, a software system is described as a configuration composed of three types of components (Station, Master Station, Host) and two connectors (Multipoint and Dialup Connector). One of the components (Master Station) has been further decomposed. The configuration used to decompose that
composite component is first used to demonstrate the configuration behavior specification and further to derive the composite component behavior (computation).

The contributions of this chapter can be summarized as follows:

- Demonstrates Wright's architectural description capabilities (previously demonstrated by authors only)
- Demonstrates existing support for hierarchy in Wright
- Exercises theoretical work presented in [RA97] related to the configuration behavior specification and the composite component computation calculation
- Guides reader through the complex theory related to the composite component behavior before it is used to define composite connectors and their behavior
4. Composite Connectors

The existing notation in Wright allows for components to be specified as either primitive or composite. When primitive, components are described using a computation behavior specified directly. When composite, component computation is described by a configuration specification. Connectors can be currently described only through a direct glue behavior specification. The notion of hierarchical decomposition is not provided for connectors. That is, components are specifiable using compositions of components and connectors but connectors are not.

The intention of this chapter is to show that connectors can be hierarchically described just using compositions of connectors or using both connectors and components. First, notation is introduced to allow connectors to be described either directly by a glue behavior specification or by a configuration specification. Second, a connector configuration is defined and used to derive the composite connector behavior. Finally, composite connector compositions are justified through examples and typical patterns of compositions.

4.1 Notation

A notation for composite connectors needs to allow for both roles and glue of connectors to be specified using compositions of other connectors and components. The roles of the composite connectors have to be implemented by roles of the constituent connectors. The glue specification has to be derived from the composite connector specification. The glue needs to be able to describe how the roles of the composite connectors are coordinated to form an interaction.

The notation used to describe composite components through configuration descriptions is used to describe composite connectors as well (Figure 35). Let us examine how this notation satisfies our requirements outlined in the previous paragraph. Analogous to component computation, composite connector glue is specified using a configuration specification. Instances of connector configuration can be connector instances only or both connector and component instances. When a connector includes both components and connectors, attachments are used to combine them into a connector configuration.
Bindings in composite connectors are concerned with mapping roles of constituent connectors to roles of the composite connector.

```
Connector <connector type>
  Role /* role of the composite connector */
  Role ...
Glue
  Configuration /* connector/component types
  Instances /* component and connector instances */
  Attachments /* port/role attachments */
End Configuration
Bind
  /* bind statements */
End Bind
```

**Figure 35: Composite Connector Type**

Composite Ports/ Roles. In addition to composite connector notation, in order to support concise and clear hierarchical descriptions, composite interfaces can be described using the following notation:

```
Interface Type <interface name>
  <list of interface specifications>
End Interface
```

Interface type can be used in both port and role specifications. When ports are attached to roles specified using the same composite interface, attachments for constituent interfaces need not be explicitly specified. When required, constituent interfaces can be accessed explicitly. For representative examples of composite interfaces refer to Figure 40 as well as the event bus example in Section 4.5.2.1 and Appendix E.

This additional complexity added to attachments and bindings requires additional static checks. First, if composite interfaces are used, name matching is a default attachment scheme. If different mapping is desired, it has to be explicitly specified. If an interface comprises of an array of interfaces, both port and interfaces have to have the same number of constituent interfaces in order to be attached.

Note also that the for loop is used in this thesis for specifying attachments and bindings of composite component and connector specifications. This is not a standard construct of Wright, but it is used in this thesis and can be considered a suggested extension of the language.
4.2 Connector Configuration

It is worth noting that configurations used to describe systems and components are slightly different than configurations that describe connectors. What makes a correct component configuration does not make a correct connector configuration and vice versa. For example, Figure 36 shows a sample component and a sample connector configuration. The connector configuration is newly introduced here to allow for composite connectors to be specified and needs to be more precisely defined.

A system configuration is a complete description that contains component and connector instances with all ports of components attached to roles of connectors using attachment statements. In a component configuration, all connectors are fully attached; components are either fully attached or partially attached. Partially attached components expose their unattached ports to the higher level of hierarchy using port bindings.

Connector configurations require some roles not to be attached. Those roles are bound to the outer connector roles using role bindings. Therefore, in a connector configuration, all components are fully attached; connectors can be either fully attached or partially attached allowing its unattached ports to be exposed to the higher level of hierarchy.

Figure 36 (a) Correct connector configuration  (b) Correct component configuration
4.2.1 Connector Configuration Behavior

The connector configuration behavior can be defined in a similar manner as the component configuration behavior (Section 2.3.1.1). The behavior of the component configuration includes only the glues of the connectors, not the roles because all connectors in composite components have to be fully. In addition to that, since all connectors are fully attached in system and component configuration, glues of partially attached connectors are not taken into account by the definition of the configuration.

In connector configurations, all components have to be fully attached, but connectors can be unattached, partially or fully attached. A connector configuration is a parallel composition of all component and connector instances after the process of attachment. Depending on the attachments defined, the configuration behavior includes the entire connector (all roles and the glue), unattached roles and the glue, or the glue only. Connectors that have at least one attached role are represented using renamed glue processes according to the attachment function. Unattached roles are represented using their role processes. Therefore, the behavior of the connector configuration, apart from the computation of components and glues of connectors, includes the behavior of all unattached roles.

Informally, a connector configuration behavior is $\text{Conf}' = \text{Conf} \mid | | \text{UnattachedRoles}$ where $\text{Conf}$ almost fully corresponds to the configuration definition from [RA97]. A more formal definition of the connector configuration behavior is formulated as follows.

**Definition (Connector Configuration Behavior).** The behavior of a connector configuration is a process

$$ \text{Conf}' = \text{Conf} \mid | ( | k: K, ( | l: L_k, \text{Cnk}; R_{k,l}; \text{RP}_{k,l}) ) $$

where

- $\text{Cpi}$ are the component calculation processes,
- $\text{CnP}_i$ are the glues of the connectors,
- $\mathcal{R}$ is the attachment renaming function,
- $\text{Conf} = ( ( | i: \text{Cpi}; \text{CnP}_i ) | | \mathcal{R} ( | j: \text{Cnj}; \text{CnP}_j ) )$,
- $K$ is a subset of $\{1..nConnectors\}$, is set of connectors that have unattached roles,
- $L_k$ is a range for unattached roles of the connector $\text{Cnk}$ and
- $\text{RP}_{k,l}$ are the role processes of the unattached roles.

In the connector configuration definition, $\text{Conf}$ is almost the same as the component configuration behavior definition except that it includes not only glues that belong to completely attached connectors.
but also glues of the connectors that are partially attached. The remaining part of the definition (| | k (| | l Cnk;Rkl;RPk,l)) simply takes care of all unattached roles of the connector configuration that will be bound to outer roles of the composite connector. The relabeling convention Cnk;Rkl;RPk,l specifies that event names to be of the form "Connector.Role.e" before those are bound to roles of the composite connector; then they become "Role.e". Cnk is a connector that has an unattached role Rk,l with the role process RPk,l.

Note that glues that belong to completely unattached connectors are included in Conf behavior but will not be renamed because attachments are not defined for them.

### 4.2.2 Correct Connector Compositions

Let us consider how composite connectors can be correctly composed as collections of either connector instances only or connector and component instances.

**Connectors composed of connectors only.** If a connector is composed out of connectors only, the behavior of the composite connector is equivalent to the parallel composition of the individual connectors after the bind renaming. That is, all roles of constituent connectors are either bound to composite connector roles or are equivalent to \( \forall \). All unattached ports have to be equivalent to \( \forall \).

**Connectors composed out of components and connectors.** If a connector contains both connectors and components, all unattached roles have to be bound to either composite connector roles or equivalent to \( \forall \). All unattached ports have to be equivalent to \( \forall \).

### 4.3 Behavior of Composite Connectors

The behavior of a connector depends both on the behavior defined by the roles and the constraints imposed by the glue. Role bindings define the behavior of the composite connector role as the behavior of the constituent connector role bound to it. The behavior of a composite connector glue can be derived from the connector configuration behavior in the same manner as the computation behavior is derived from the component configuration behavior.

---

14 The unattached role condition can be found in [RA97].
In Section 4.2.1, we have defined connector configuration behavior. The behavior of a connector configuration shows how individual component and connector contribute to the overall configuration behavior. As shown previously for components (Section 2.3.1.2), the behavior of composite element can be projected onto the set of events that are of some interest to us. In this case, we are interested only in those events of the glue that directly coordinate the events of the outer roles. That is, we only need those events that are observed by the outer roles to be initiated by the glue and vice versa. All other events of the connector configuration can be hidden. Using the terminology previously defined for composite components, this set of events is called the *keeping event set* of the roles. The keeping event set of the roles is a set of events exposed to the outer roles of the composite connector using role bindings.

The behavior of the composite glue is the behavior of the connector configuration projected onto the keeping event set of the roles.

---

**Definition (Connector Behavior).** The behavior of a composite connector is a process

\[ \mathcal{R}(\text{Conf} \mid \setminus S) \]

where

- \( \text{Conf} \) is the connector configuration behavior
- \( \mathcal{R}_i, i = 1..n \) are the roles of the connector
- \( S \) is a keeping event set of the roles \( \mathcal{R}_i \) defined by role bindings, \( S = S_1 \cup ... \cup S_n \).
- \( \mathcal{R} \) is a composite rename function based on the bind renaming functions \( \mathcal{R}_j, i = 1..m \) of the composite connector, \( \mathcal{R} = \mathcal{R}_1^\circ ... \circ \mathcal{R}_k \)

\( \text{Conf} \) in the connector behavior definition is the behavior of the connector configuration as specified in Section 4.2.1. The keeping event set \( S \) is actually a set of all events of the form Connector.Role'.e that will be exposed to the higher level of hierarchy as Role.e for all roles of the composite connector. Each bind renaming function \( \mathcal{R}_j, i = 1..m \) represents one role binding statement "Connector.Role' as Role" that renames events of the form Connector.Role'.e to Role.e.

Having defined the behavior of the connector as a whole, the connector behavior can be further expressed as a parallel composition of the glue and the behavior of all roles.
Definition (Composite Connector Roles and Glue). If \( R_{pi} \) are role processes of the internal roles bound to roles \( R_i \) of the outer connector, then the above connector behavior definition is equivalent to

\[
\mathcal{R}(\text{Conf} | \neg S) \ | | ( | | i: R_i; R_{pi})
\]

where
- \( \mathcal{R}(\text{Conf} | \neg S) \) is the behavior of the composite glue process
- \( | | i: R_i; R_{pi} \) is the behavior of all roles.

Note that the projection does not affect the role behavior since by definition, the keeping event set \( S \) is defined using all events that appear in roles that are bound to outer connector roles. There are no events in roles that need to be hidden.

The example that follows exercises the composite connector behavior definition in the simple and intuitive way.

4.3.1 Behavior Computation Example

This example shows a simplified version of the remote procedure call implementation. The remote procedure call connector is already discussed in Section 2.3.2. The intention now is to show that the protocol of interaction used for the remote procedure call and basically equivalent to the behavior of the procedure call is maintained by the underlying implementation.

Wright specification of the remote procedure call connector is shown in Figure 37. The complete specification of this connector is also included in Appendix D.
The behavior of the composition is calculated in two steps. First, the behavior of the composite component comprising the client stub, server stub and communication link (Figure 38) between them is calculated. Second, two procedure call attachments is made, followed by their binding to outer roles. These steps allow for simpler calculation of the overall behavior of the RPC connector.
4.3.1.1 STEP 1: Behavior of the composition

The behavior of the component composition is $C \parallel \text{Glue} \parallel S$ where $C$ is the client stub computation, $S$ is the server stub computation and Glue is a communication link glue after instance and attachment renaming:

$$C = \text{CS.Client.request} \rightarrow \text{CS.Com.send} \rightarrow \text{CS.Com.receive} \rightarrow \text{CS.Client.return} \rightarrow C$$

$$\text{Glue} = \text{CS.Com.send} \rightarrow \text{SS.Com.send} \rightarrow \text{SS.Com.receive} \rightarrow \text{CS.Com.receive} \rightarrow \text{Glue}$$

$$S = \text{SS.ComSend} \rightarrow \text{SS.Service.request} \rightarrow \text{SS.Service.return} \rightarrow \text{SS.Com.receive}$$

Using CSP parallel composition laws[Hoa85]:

- $A \parallel B = a \rightarrow (A' \parallel B)$ where $A = a\rightarrow A'$ and $a$ is not in the alphabet of $B$.
- $A \parallel B = a \rightarrow (A' \parallel B')$ if $A = a\rightarrow A'$ and $B = a\rightarrow B'$

the following can be derived

$$C \parallel \text{Glue} \parallel S = X = \text{CS.Client.request} \rightarrow \text{CS.Com.send} \rightarrow \text{CS.Com.receive} \rightarrow \text{CS.Client.return} \rightarrow X$$
Let us suppose that ports of the composite component are Client and Server. Let them be assigned to CS.Client and SS.Service ports respectively. When hiding events that do not appear in the keeping set of the ports Client and Server, the result is:

\[ X = \text{Client.request} \rightarrow \text{Service.request} \rightarrow \text{Service.return} \rightarrow \text{Client.return} \rightarrow X \]

This is the computation of the component that comprises out of Client Stub, Server Stub and Com Connector.

4.3.1.2 STEP 2: Behavior of the RPC connector

First, the role ClientPC.Definer is attached to the X.Client port and the role ServerPC.Caller to the X.Service port. Second, all events that do not appear in the ClientPC.Caller (bound to Caller) and ServerPC.Definer (bound to Definer) are hidden.

\[ \text{Glue}' = \text{ClientPC.Caller.request} \rightarrow \text{ServerPC.Definer.request} \rightarrow \text{ServerPC.Definer.return} \rightarrow \text{ClientPC.Caller.return} \rightarrow \text{Glue}' \]

Finally, renaming is done according to the bound roles. The resulting Glue becomes:

\[ \text{Glue} = \text{Caller.request} \rightarrow \text{Definer.request} \rightarrow \text{Definer.return} \rightarrow \text{Caller.return} \rightarrow \text{Glue} \]

This shows that the behavior of the remote procedure call implemented using the client and the server stubs with the communication link between them and client and server local procedure call, maintains the behavior of the local procedure call.

The remote procedure call composite implementation brings the following benefits:
- it increases the comprehension of the connector implementation
- it promotes the reuse of the connector design
- it allows a designer to verify the behavior of the composition against the well known or a target behavior.

Software engineers need to build complex connectors in terms of available primitive connectors and components. Further, they need to be able to verify how the configuration that is used to implement the connector meets the expected or the target behavior of the connector. Once the connector has been built, it can be reused by other systems or within the same system.
4.4 What Makes a Component Part of a Connector?

One might claim that there is some inconsistency and confusion now that components are part of connectors. What good is it to make components part of connectors? When is it meaningful to consider a component be part of a connector or an entire sub-configuration be a connector? Let us discuss some of the questions frequently raised on this topic.

Components are part of connectors only if they are required to facilitate the interaction. They are considered part of the communication infrastructure rather than relevant functional elements of the system. Communication controllers, routers, registrars are some typical infrastructural elements of the system; databases and files can be related to maintaining communication data or being a medium of communication; stubs can be automatically generated or manually implemented as part of establishing communication between components. All mentioned components can be used to facilitate interaction between components and can therefore be considered part of connectors. A customer database or any application related computation is not a meaningful part of a connector. It is not always appropriate to include components in connector compositions. Incorrect use of the freedom given to the designer can cause confusion instead of leading to better comprehension of a complex software system. Unfortunately, neither Wright nor other architecture description languages can check this for the designer.

Can a connector have memory (state)?

Although components are "the locus of the state and the computation", it does not imply that connectors can not have state or memory. Pipes, queues and mailboxes all have memory but the memory is used to facilitate communication between components.

Are connectors still required?

Now that connectors are described using components, a valid question if we still need connectors. Connectors are still required as an abstraction that depicts communication in architectures. It is long known that architectures can be described using just components, but with interactions hidden in them. We are not suggesting here that connectors be hidden in components. Our connectors are still explicit and describe interactions, but they are also able to hide entire architectures that are expressing infrastructural properties of the system.
4.5 Using Connector Compositions

4.5.1 Parallel Composition

Parallel Composition. One of the most frequently used compositions of connectors is definitely parallel composition. Examples include bundles of procedure calls or data use connectors, object use, file use, etc. Connectors are grouped together because they present a "connection pattern" [MDEK95] that can be reused. For example, if a two way communication is required, two one way connectors can be utilized. Figure 39 and Figure 40 show how a two way pipe can be constructed.

Parallel composition should be used with care. When parallel composition is used, decision should be made whether these interactions are truly independent. Sometimes, the connectors that we have "bundled" have dependencies hidden in components. For example, bundle of procedure calls has an exact calling sequence hidden in the component. In that case, the dependency should be pulled out of component and built into a connector (this is what connectors are for).

<table>
<thead>
<tr>
<th>Interface Type</th>
<th>PipeRWInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>WriterProtocol</td>
</tr>
<tr>
<td>Out</td>
<td>ReaderProtocol</td>
</tr>
</tbody>
</table>

Note that not only connectors of the same type can be composed in parallel. Any connection pattern can be captured as repetitive in the system or across multiple systems and represented as a composition. Note also that a good companion of the parallel connector composition is a composition of roles and ports.

Figure 39: Two way pipe

Figure 40: Two Way Pipe
4.5.2 Bus Connectors

A connector is called a bus if it allows multiple components called bus clients, tools, or participants, to communicate using the bus as communication media. The bus is a symmetric protocol and all participants have the same roles on the bus. The bus specifies the protocol of interaction of its participants. Most commonly used bus connectors are event bus and software bus connector. Figure 41 gives an example of a bus connector.

![Bus Connector](image)

**Figure 41: Bus Connector**

4.5.2.1 Event Bus

An event bus connector is used as a loose tool integration mechanism. Tools announce events that might be of interest to other tools. Other tools observe events that are of interest to them and perform actions in response to them. Tools are said to be loosely coupled because they interact without knowing the identity of one another. Many event based systems are available today such as FIELD [Reiss '90], HP's SoftBench [Ger89], Sun's ToolTalk [Sun92].

An event bus is a connector abstraction that allows for events to propagate between tools attached to the bus. Tools announce events that might be of interest to other tools. Other tools observe events that are of interest to them and perform actions in response to them. A tool that registers for, announces and receives events is usually referred to as a participant [GAO94].

The informer of an event is a participant that announces that event; the listener of an event can be any participant that observes that event [BCTW96]. The communication on the event bus connector adapted in this example requires both informer and listener to register their interest in events before they announce and observe them. Once registered, informer/listener can at any given point decide to deregister and no longer announce/observe events. Informer and listener roles in the event bus communication can be described as in Figure 42.
Event Channel. An event channel is a connector that carries only one event. An event can have multiple informers and listeners. The event channel presented in this example, called EventConnector, can accept dynamic registration and deregistration of event informers and listeners. This approach allows for flexible reconfiguration of the parties involved in communication.

The glue of the EventConnector is specified using parallel composition of two processes (Figure 43). Process AnnounceEvents is responsible for accepting events and for routing them to other tools registered as listeners. The HandleRegistration process is responsible to handle registration and deregistration of informers and listeners of the event. This process is also the locus of registration information for the event. It holds the current set of registered listeners (L) and informers (I). When AnnounceEvents process requires list of listeners for the event, the HandleRegistration process releases that information using the internal where!L event. Note that event where?L does not belong to any role, it is locally used inside the glue process to handle synchronization between the processes AnnounceEvents and HandleRegistration.

There are two techniques used to model event bus connector worth noting. First, registration handling and routing of events is split into two parallel processes. Second, one of the processes, namely HandleRegistration is used to hold connector state information. These two techniques are reused from...
HLA/RTI glue specification ([AG97]). HLA/RTI has a complex glue specification that is modeled using multiple parallel processes. Processes use a single "internal state process" to keep track of the federates (participants in the HLA/RTI communication) that joined/resigned execution. Similarly, in this example, we keep track of the event bus participants that registered/deregistered their interest in events.

The "internal state process" approach to glue specification, materialized in HandleRegistration, facilitated handling a deadlock situation in the connector specification. For example, let us consider a specification of the EventConnector glue that does not take advantage of the internal state process. Instead, the specification keeps track of the state using the process state variables I and L of the process AnnounceEvents. The EventConnector glue is specified in Figure 44.

\[
\text{Glue} = \text{AnnounceEvents}([],[]) \text{ where }
\]

\[
\text{AnnounceEvents}(I,L) = ([I]:I \text{ Inforer}(i).\text{annouce} \rightarrow (j:L \text{ Listener}(j).\text{listen} \rightarrow \top) ; \text{AnnounceEvents}(I,L))
\]

\[
[[[I]:I \text{ Inforer}(i).\text{deregister} \rightarrow \text{AnnounceEvents}(I-\{I\},L))
\]

\[
[[]][L] \text{ Listener}(i).\text{deregister} \rightarrow \text{AnnounceEvents}(I,L-\{i\})
\]

\[
\top
\]

**Figure 44: The EventConnector Glue Specification**

The process AnnounceEvents observes deregistration of both listeners and informers and observes announcement of events. Announced event is further routed to all registered listeners of the event in some order. Although the protocol looks fine, there is a small problem in this specification that can cause a deadlock. According to the listener's behavior specified by ListenerProtocol, a listener, once registered, can either observe an event or decide to deregister. If the listener \( \text{Listener}(j) \) decides to deregister, then the attempt to issue \( \text{Listener}(j).\text{listen} \) will result in a deadlock. \( \text{Listener}(j) \) will be blocked waiting to deregister while the glue will be blocked trying to synchronize on the listen event \( \text{Listener}(j).\text{listen} \) with the listener.

In order to resolve this problem, the glue has to be able to synchronize with the listener on both listen event and deregister event. Further, in the case the glue observes deregister event, the glue process has to update the set of registered listeners (L becomes L-\{j\}). Handling of connector state variables I (registered informers) and L (registered listeners) becomes cumbersome and difficult to understand. The easiest way to resolve the problem is to introduce the internal state process that will keep track of the registered listeners and informers. This justifies the existence of the HandleRegistration process in the EventConnector specification.
Event Bus Composite Connector. Multiple event channels are composed to form an event bus. All attachments to the bus are gathered in the composite Participant role. Note that although event channels themselves are asymmetric connectors, event bus composed out of event channels is a symmetric connector.

```
Interface ParticipantProtocol( nTools:1.. )
   Informer (1..nTools)= InformerProtocol
   Listener (1..nTools)= ListenerProtocol
End Interface

Connector EventBus(nTools:1.., nEvents:1.. )
   Role Participant = ParticipantProtocol
   Glue
      Configuration
      Instances
         ec(1..nEvents): EventConnector
      End Configuration
      Bind
         for e: 1..nEvents
            for i:1..nTools
               ec(e).Informer to Participant(i).Informer
               ec(e).Listener to Participant(i).Listener
            end for
         end for
End Bind
```

Figure 45: Composite event bus connector based on multiple event channels

This event bus shows how multiple event channels are composed to form an event bus. All attachments to the bus are gathered in the composite BusParticipant role. Although event channels themselves are asymmetric connectors, the event bus composed out of event channels is a symmetric connector (recall the pipe example from Chapter 3). Finally, an example of the event bus usage is given in Figure 46.

```
Configuration

   Instances
      Tools(1..nTools): Tool
      Ebus: EventBus(nTools, nEvents)
   Attachments
      for i:1..nTools
         Tool(i).EventBus as Ebus.Participant(i)
      end for
End Configuration
```

Figure 46: An Example of Event Bus Usage
This example demonstrates the following:

- an abstract connector event bus is composed out of event channel connectors
- event channel roles are composed into a single bus participant role
- asymmetric connectors are used to form a symmetric connector
- event channel abstractions ease the understanding of the event bus connector

Another model of an event bus, "Event Bus 2" (Appendix E) shows how an event bus can be implemented using a central component, often called Event Manager, to accept event messages and further forward them to the registered listeners.

4.5.2.2 Software Bus

The software bus abstraction is getting significant attention in the software development industry. SWBus [ML97], Q[MMH95], PDEBus[PE], Polyth [Pu91] are only some examples of software buses. Software buses as well as event buses are a method of loose tool coupling because tools on the bus are not aware of each other's identity. Software buses allow components to export their services which are invoked by components registered with the bus to import those services. The most important benefit for using the software bus abstraction is that it hides machine-dependent and language-dependent communication from communicating parties. All parties have to be linked to the bus library that provides a set of bus APIs that are used throughout the tool. Libraries are provided for different languages and underlying platforms.

All software buses provide certain interface and promise certain behavior to the participant tools. However, the interface and the behavior can be implemented using different underlying architectures. There is a need to describe connectors in terms of other connectors and components that each take over part of overall responsibilities of the complex connector.

4.6 Why Composite Connectors?

As shown through the above classes of connectors, connectors as well as components are abstractions that equally need support for compositions. Some of the connectors that were available to the software architect for a long time are procedure calls, mailboxes, semaphores, queues, shared memory, and many other programming and system services. However, as software systems are getting large and complex, new mechanisms are arising that are able to connect tools written in different languages, running on different machines. Some of them are various kinds of software buses, implicit invocation systems and
many others. New standards for tool integration such as CORBA are emerging. The time has come to develop not only complex components but more sophisticated methods for their connectivity.

Wright allows very complex standards such as RTI ([AG97]) to be specified and analyzed. Further, by introducing hierarchical connector description, complex connector descriptions can also be decomposed into manageable pieces. Some of the benefits gained from this approach are the following:

- easier comprehension of the complex behavior;
- higher abstraction hiding underlying details of implementation;
- reuse of complex connector implementations;
- the ability to create new connectors using the existing connectors and components;
- the ability to create and specify custom connectors.

4.7 Conclusion

We started this chapter by extending Wright's formal notation to support composite connectors. Although the notation is specific to Wright, the idea is to elevate composite connectors to a standard element of the architecture description languages.

We next defined a basis for analysis and checking of composite connector specifications. First, we defined the correct connector configurations allowing for checking of completeness of composite connectors. Second, we defined composite connector behavior as a function of its constituent connectors and components. This allows for analysis of the connector behavior. The behavior specification can be used by the designer to achieve the target behavior of the composition (as in the RPC example); it can also be used to better understand the effects of the composition; or to understand the implementation of the specified high level behavior. One of the important contributions of the behavior specification is that some of the checking and analysis initially defined for connectors in Wright (such as deadlock freedom analysis for roles and connectors) can be applied to composite connectors as well.

At the end of this chapter we advocated the hierarchical connectors by putting them in the context of the current trends in software development i.e. the software and event buses as well as by giving examples of connectors using the notation and theory presented in the chapter.
The most valuable contributions of the notation and theory presented in this chapter are the following:

- Specifying complex connectors at various levels of abstraction;
- Reusing connector designs;
- Increasing the understanding of the connectivity implementation details;
- Verifying the behavior of the connector against well known or target behavior;
- Performing static checks based on the definitions of correct configurations;
- Performing deadlock analysis for complex connectors.
5. Conclusions

5.1 Summary

By selecting Wright as an architectural description language specifically designed to describe architectural connection, the work presented in this thesis emphasizes the importance of connectivity in architectural descriptions. Although other languages also recognize interaction as an important part of architectural descriptions (Rapide), they do not provide for explicit connectors in the language.

We also argue that hierarchy, not initially present in Wright and introduced recently to express composite components, is an important feature of an architecture description language. Designers need to be able to describe systems at a high level of abstraction and to gradually present elements of the system in a hierarchical fashion. Existing architecture description languages support hierarchical descriptions but allow only components to be presented at various levels of abstraction. We recognize in Chapter 2 that both connectors and components need to be hierarchically decomposed. Further in Chapter 3 we present a detailed case study that exercises the theory behind composite components in Wright so that in Chapter 4 an analogous notation and theory can be developed for composite connectors.

Using Wright as underlying architectural formalism, we introduce composite connectors in Chapter 4. We propose a notation that can be used to specify composite connectors in Wright. Further we develop a theory around the composite connectors in a manner similar to Allen's [RA97] definition for components. This theory serves as firm basis for architecture analysis and checking related to composite connectors. First, we allow connector configurations to be checked for completion; second, the behavior of the composite connector is defined as the function of its constituents and that allows the theory previously defined for primitive connectors (deadlock freedom) to be extended for composite connectors. We also support our formalism with examples taken from the current software development practice (remote procedure call, event bus, software bus).

The composite connector notation and theory presented in this thesis can be used in a connector building tool (connector notation, connector configurations, analysis and checks such as deadlock freedom or other analysis of connector behavior), as part of an architecture specification tool, or in general to specify and reuse complex interactions.
5.2 Future Work

Connector Building Tool. The theory outlined here and original Wright related to connectors can be used in a connector building tool. Some of the analysis and checking that can be used includes: checking correct connector configuration, checking deadlock freedom based on the behavior of the composite connector, and checking of compatibility of ports and roles.

Other Languages. Composite connectors can be introduced to other languages that support connectors as first class entities.

Constraints on Components. It would be good to investigate and define how behavior of components in connectors differs from components in general. Are there any conditions that need to be met for a component that is part of connector? For example, if a component initiates an event, does that mean that the connector glue can initiate events as opposed to just coordinating events of the roles?

Refinement using different styles. The work in this thesis does not cover refinement - different styles being used to refine high level designs. Without saying, this is very important aspect of design and would be an interesting area of research. Consider how high level design constraints can be preserved on the lower level when different styles are used on the higher and lower levels of hierarchy.

Rapid-like Notation. The description of hierarchy in [RA97] is similar to what UniCon and Darwin support in terms of composite components. However, Rapid has slightly different approach that can be applied to Wright as well. That is, Rapid allows for both interface and behavior to be defined for both primitive and composite components. Further, if a component is elaborated through composition (architecture) previously defined behavior is used to check whether the behavior of the composite matches the behavior of the configuration. Being able to define both composite component computation and composite connector behavior, why not allow for both behaviors to be defined and compatibility checked.

This approach also allows connector behavior to be directly specified for both primitive and composite connectors. Further decomposition of composite elements can be deferred after checking and analyzing the high level specifications.
The Rapide-like notation could be the following:

a) for composite components:

```
Configuration <name of the composite element type>
is Component <name of the component type>
<Types of components and connectors>
Instances
<Instances of connectors and components >
Attachments
<Instances of connectors and components >
Bindings
<Bind internal configuration ports to external ports >
End Configuration
```

b) for composite connectors:

```
Configuration <name of the composite element type>
is Connector <name of the component type>
<Types of components and connectors>
Instances
<Instances of connectors and components >
Attachments
<Attach ports of internal components to roles of internal connectors >
Bindings
<Bind internal configuration roles to external roles >
End Configuration
```
References


[PDE] PDELab pdelab@cs.purdue.edu
http://www.cs.purdue.edu/research/cse/pdelab/pdebus/


Appendix A

Wright Process Notation Reference

e->P  Sequencing. Process engages in event e and becomes process P
Note that e->f -> g -> P is equivalent to e -> ( f -> ( g -> P ))

e!x, e?x  Events with data. Events can output data (e!x) or input data (e?x)
e, g  Observed and initiated events. Process e->P observes e and becomes P. Process e->P
observes e and becomes P. In Wright, overbar\textsuperscript{15} indicates initiated events.

Q= e->P  Naming. Behavior patterns (such as e->P) that frequently occur can be named
(Q=\overline{e->P}).

e->P where P=...Named processes can be used and then defined using the keyword where
P=e->P  Recursion. Recursion can be defined using naming.
e->P ∏ f->Q  Non-deterministic (external) choice. Process e->P ∏ f->Q may choose to engage in
either e or f depending on its internal processing. The behavior of the process is non-
deterministic if it solely depends on what the process internally decides to do. Usually
used with initiated events.
e->P □ f->Q  Deterministic (internal) choice. If e occurs first, the process e->P □ f->Q will engage in
e and become P; if f occurs, it will engage in f and become Q. The behavior of the process
is entirely determined by what environment does Typically used with observed events.
P;Q  Sequence of Processes. P;Q is a process that behaves like P until P successfully
terminates and then becomes process Q.
STOP  Stop process. Stop process engages in no events.
✓  Success event.
✓  Successful termination process. Equivalent to the process "✓->STOP"
P(i)  State. "Subscripting" is used to express state dependency. For example, P(0) where P(i)
= e!i->P(i+1) is a process that engages in event e and outputs total number of
engagements in the event e.
□x: S • P(x)  External choice between different P(x)
For example, □i: (1..n) • P(i) = P(1)□P(2) □...□P(n)

\textsuperscript{15} Overbar throughout this thesis represented as an underlined text. Underlined text is not originally used in
Wright.
\( \Pi: S \bullet P(x) \) Internal choice between different \( P(x) \)

For example
\[ \Pi: (1..n) \bullet P(i) = P(1) \Pi P(2) \Pi \ldots P(n) \]

\( \forall: S \bullet P(x) \) Execution of all \( P(x) \) in some (non-deterministic) order.

\( \forall: S \bullet P(x) = ( \Pi \forall: S \bullet ( P(x) \cup \{ y: S \setminus \{ x \} \bullet P(y) \}) \)

For example
\[ \forall: (1..3) \bullet P(i) = ( P(1); P(2); P(3) ) \Pi ( P(1); P(3); P(2) ) \]
\[ \Pi ( P(2); P(1); P(3) ) \Pi ( P(2); P(3); P(1) ) \]
\[ \Pi ( P(3); P(1); P(2) ) \Pi ( P(3); P(2); P(1) ) \]

Miscellaneous Definitions

Alphabet. Alphabet of a process \( P \) is set of all events, called \( \alpha P \), over which the process has influence.

Renaming. Types:

1. **Instance renaming** is re-labeling used to make multiple copies (instances) of behavior specifications. Re-labeling \( L: P \) denotes that all events in \( P \) are prefixed with \( L \). Event \( e \), controlled by \( P \), becomes \( L.e \).

2. **Attachment renaming** is used to rename events of the role with events of the port. Note that all events of the glue have role labels. Therefore, attachment renaming affects glue events as well. They become renamed after ports attached to corresponding roles.

   **Def.** For any names, \( N, N', M, M' \) not necessarily different,

   \[ \gamma_{(N,M), (N',M')}(e) = N'.M'.e', \quad \text{if } e = N.M.e' \]
   \[ e, \quad \text{otherwise} \]

2. **Bind renaming** is used to rename events of the port/role with events of the outer port/role in composite components/connectors. Note that all events of the glue have role labels. Therefore, bind renaming affects glue events as well.

   **Def.** For any names, \( N, N', M, M' \) not necessarily different,

   \[ \gamma_{(N,M), N}(e) = N'.e', \quad \text{if } e = N.M.e' \]
   \[ e, \quad \text{otherwise} \]
Appendix B

File Access Connector

Each of the file users promises to open file first before it uses it. After opening the file, user can read and write any number of times before it closes the file. After it closes the file, user successfully terminates.

\[
\text{Process FileUserProtocol} = \text{open} \rightarrow \text{Use} \sqcap
\]
\[\text{where} \quad \text{Use} = \text{read} \rightarrow \text{Use} \]
\[\quad \text{write} \rightarrow \text{Use} \]
\[\quad \text{close} \rightarrow \sqrt{\text{}},\]

Protocol \text{FileAccessProtocol} reflects the role of the file component in the file access interaction. File promises to observe open event first and then allow for multiple file reads and writes. Only after the file is closed, it is able to accept open event again. This is expressed using the recursive specification "close-FileAccessProtocol".

\[
\text{Process FileAccessProtocol} = \text{open} \rightarrow \text{Use} \sqcap
\]
\[\text{where} \quad \text{Use} = \text{read} \rightarrow \text{Use} \]
\[\quad \text{write} \rightarrow \text{Use} \]
\[\quad \text{close} \rightarrow \text{FileAccessProtocol} \]

File access connector is has one role for the file (File) and multiple file user roles (FileUser). File access protocol allows one user only at any time to access the file. The user has to open file first, then read and write any number of times and finally close the file.

\[
\text{Connector FileAccessConnector(numUsers: } 1..)\]
\[\text{Role FileUser...numUsers} = \text{FileUserProtocol} \]
\[\text{Role File} = \text{FileAccessProtocol} \]
\[\text{Glue } = \{[i:(1..\text{numUsers})\text{FileUser.open} \rightarrow \text{FileUse.open} \rightarrow \text{Use(i)} \} \sqcap \]
\[\text{where} \quad \text{Use(i)} = \text{FileUser.read} \rightarrow \text{FileUse.read} \rightarrow \text{Use(i)} \]
\[\quad \text{FileUser.write} \rightarrow \text{FileUse.write} \rightarrow \text{Use(i)} \]
\[\quad \text{FileUser.close} \rightarrow \text{FileUse.close} \rightarrow \text{Glue} \]
Appendix C

Composite Component Example

Dial Connection

/*
  Dial Protocol
  Initiate connect and communicate or decide to stop
  Communicate sends a message x and observes response y; any number of
  request/response messages can be sent before the process decides to disconnect.
*/

Interface Type DialProtocol = connect \rightarrow Communicate \Pi \\{\}
  where Communicate = send!x \rightarrow receive?y \rightarrow Communicate
  \Pi disconnect \rightarrow DialProtocol

/*
  Host Protocol
  Observes connect event and enters communication with the caller.
  Communicate awaits request message from the caller and then sends response.
  Any number of request/response messages can be received/sent before
  the process accepts disconnect event generated by the caller.
*/

Interface Type HostProtocol = connect \rightarrow Communicate \Pi \\{\}
  where Communicate = send?x \rightarrow receive!y \rightarrow Communicate
  \Pi disconnect \rightarrow HostProtocol

/*
  Dial Connector
  Roles are Caller and Host specified by the protocols DialProtocol and HostProtocol
  respectively. The glue process coordinates the communication of the roles. The
  glue in this case only observes connect, send, and disconnect of the Caller and
  initiates corresponding events for the Host. It also observes receive event of
  the host and triggers receive event of the Caller.
*/

Connector DialConnector
  Role Caller = DialProtocol
  Role Host = HostProtocol
  Glue
    Caller.connect \rightarrow Host.connect \rightarrow Glue
    \Pi Caller.disconnect \rightarrow Host.disconnect \rightarrow Glue
    \Pi Caller.send?msg \rightarrow Host.send!msg \rightarrow Glue
    \Pi Host.receive?msg \rightarrow Caller.receive!msg \rightarrow Glue
    \Pi \{\}

End Connector

The CallerProtocol is used by the port of the component Dialer and is compatible to the role Caller of the
DialConnector.

Interface Type CallerProtocol = connect \rightarrow Communicate \Pi \\{\}
  where Communicate = send?x \rightarrow receive!y \rightarrow disconnect \rightarrow CallerProtocol

74
Multidrop Connection

/*
MasterProtocol
Master protocol describes communication between the master station and slave stations. The master station polls 1..nStations one by one. For simplicity stations are considered to be numbered from 1..nStations and assumed to be consecutive numbers with no gaps. If a station has a message to be sent, it sends a message, otherwise it rejects the poll. The master station awaits for either the message or reject from the slave station.
*/

Protocol MasterProtocol(nStations:1..) = Poll(1) /* Start polling station 1 */
where
/* Poll i-th station */
Poll(i) = poll!i -> WaitForRequest(i)
  /* Initiate poll of the i-th station and wait for request message from the station (if any) */
Poll(nStations+1) = Poll(1) /* When polling reaches nStations +1 it becomes 1, the first station is polled again */

/* Wait for polled station to respond with request or reject the poll */
WaitForRequest(i) = send?msg -> Response(i)
  /* Observe message from the station; if received, stop polling and wait for the response to come from the server */
  [] reject -> Poll(i+1)
  /* Observe reject in case polled station has no message to send to the server */
  [] ✓
  /* Observe successful termination in case station performs termination (this helps master not to deadlock when station terminates while being selected/pollled) */

/* Deliver server's response to the i-th station */
Response(i) = select!i -> WaitForReply(i)
  /* Initiate select for the polled station and wait for the station to be ready to accept the response message from the server */
WaitForReply(i) = ack -> receive?msg -> Poll(i+1)
  /* Observe ack from the selected station and if ack received send message to the station */
  [] reject -> Poll(i+1)
  /* Observe reject from the station, drop the response and continue to poll. */
  [] ✓
  /* Observe successful termination in case station performs termination (this helps master not to deadlock when station terminates while being selected/pollled) */
/* Station Protocol */

Stations have passive role in the communication. Stations observe poll to send message to the server and select to receive response from the server.

Protocol StationProtocol = WaitPoll \( \Pi \) \( \sqrt{\cdot} \) /* Wait poll or decide to terminate */

where

/* Wait to be polled */

WaitPoll = poll \( \rightarrow \) AnswerPoll /* Observe poll and respond to it */

/* Answer poll by sending a message or reject if no message to be sent*/

AnswerPoll = send!x \( \rightarrow \) WaitResponse /* Send message */

\( \Pi reject \rightarrow \) StationProtocol /* Reject the poll */

/* If message sent, wait for response */

WaitResponse = select \( \rightarrow \) AnswerSelect /* Observe select */

/* Acknowledge select and get the response*/

AnswerSelect = ack \( \rightarrow \) response\(?^x\) \( \rightarrow \) StationProtocol /*one cycle finished, repeat this behavior again or terminate */

/* Multidrop Connector */

This connector specifies the behavior of the master/station connection using the protocols for the master and station roles described previously. The glue is responsible for coordinating events of roles in this communication i.e. all polled stations and the master station.

Connector Multidrop(\(nStations:1..\))

Role Master = MasterProtocol(\(nStations\))

Role Station \(1..nStations\) = StationProtocol

Glue = i:(1..nStations)

Master.poll!i \( \rightarrow \) Station(i).poll \( \rightarrow \) Glue

[] Station(i).send \( \rightarrow \) Master.send!msg \( \rightarrow \) Glue

[] Station(i).reject \( \rightarrow \) Master.reject \( \rightarrow \) Glue

[] Master.select(i) \( \rightarrow \) Station(i).select \( \rightarrow \) Glue

[] Station(i).ack \( \rightarrow \) Master.ack \( \rightarrow \) Glue

[] Master.receive?\(x\) \( \rightarrow \) Station(i).receive!\(x\) \( \rightarrow \) Glue

[] \( \sqrt{\cdot} \)

The Mailbox Connector

Connector Mbox

Role in = deposit!x \( \rightarrow \) in \(1\) \( \sqrt{\cdot} \) /* MBoxDepositor */

Role out = check \( \rightarrow \) (nomsg \( \rightarrow \) out \( \Pi \) receive?\(x\) \( \rightarrow \) out) /*MBoxOwner */

Glue Empty

where Empty = out.check \( \rightarrow \) nomsg

[] in.deposit?\(x\) \( \rightarrow \) Full(\(x\))

Full(\(x\)) = out.check \( \rightarrow \) receive!\(x\) \( \rightarrow \) Empty

[] in.deposit?\(x\) \( \rightarrow \) Full(\(x\)) /* Deposit overrides previous value*/
Composite Component MasterStation

/*
 * MasterStation Component
 */

Component MasterStation(nStations:1..)
    Port Stations = MasterProtocol
    Port Host = CallerProtocol
Computation
Configuration
  /*
   PollController Component
  */

Component PollController(nStations:1..)
    Port Stations = MasterProtocol
    Port MboxPoller = MboxOwner
    Port MboxDialer = MboxDepositor
Computation = Poll(1)
where
  /* Polling */
  Poll(nStations+1) = Poll(1)  /* Cyclic poll 1..nStations */
  Poll(1) = Stations.poll!i -> WaitForRequest(i)
  /* Initiate poll of i-th station, wait for request */

  /* Wait for slave station to send request */
  WaitForRequest(i) = Stations.send?msg -> MboxDialer.deposit!msg -> Response(i)
  /* Observe send from the slave station, deposite received message to the dialer's mailbox
   and wait for response from the server */
  [] reject -> Poll(i+1)
  /* Observe reject, continue to poll */

  [] /* Observe slave station termination
   to avoid deadlock */

  /* Wait for response from the server */
  Response(i) = MboxPoller.check -> GetMboxStatus(i)
  /* Check mailbox and get status or message */

  GetMboxStatus(i) = MboxPoller.nomsg ->Response(i)
  /* No message in the mailbox, check again later */
  [] MboxPoller.receive?msg -> Stations.select!i ->
    WaitForReply(i,msg)
  /* Receive message, send it to the polled station */
  WaitForReply(i,msg) = Stations.ack -> Stations.receive?msg -> Poll(i+1)
  /* Observe ack from the selected station,
   send message, continue to poll */
  [] Stations.reject -> Poll(i+1)
  /* Observe reject, drop the message,
   continue to poll */

  [] /* Observe slave station termination to avoid deadlock */

  /*
   DialController Component
  */

Component DialController(nStations:1..)
    Port Host = CallerProtocol
    Port MboxPoller = MboxDepositor
    Port MboxDialer = MboxOwner
Computation = MboxDialer.check ->
  ( MboxDialer.nomsg -> Computation )
  [] MboxDialer.receive?x ->Host.connect ->
    Host.send!x -> Host.receive?y ->
    Host.disconnect -> MboxPoller.deposit!y ->Computation )
Instances

Poller: PollController(nStations) /* Define Poller as a PollController instance */
Dialer: DialController(nStations) /* Define Dialer as a DialController instance */
PollerMbox: Mbox /* Define Poller's mailbox connector */
DialerMbox: Mbox /* Define Dialer's mailbox connector */

Attachments

/* Attach ports of component instances to roles of connector instances */
Dialer.PollerMbox as PollerMbox.in
Poller.PollerMbox as PollerMbox.out
Poller.DialerMbox as PollerMbox.in
Dialer.DialerMbox as PollerMbox.out

End Configuration

Bind

/* Bind constituent component's ports to ports of composite component */
Poller.Stations to Stations
Dialer.Host to Host

End Bind

End MasterStation

Calculating Behavior of the MasterStation Configuration. The behavior of the configuration is the following CSP process:

Conf = PollerComputation | | DialerComputation | | PollerMboxGlue | | DialerMboxGlue

Process PollerComputation = Poll(1)

where

Poll(nStations+1) = Poll(1)
Poll(i) = Poller.Stations.poll!i -> WaitForRequest(i)
WaitForRequest(i) = Poller.Stations.send?msg -> Poller.MboxDialer.deposit!msg - > Response(i)

[] Poller.Stations.reject -> Poll(i+1)
[]
Response(i) = Poller.MboxPoller.check -> GetMboxStatus(i)
GetMboxStatus(i) =
Poller.MboxPoller.nomsg -> Response(i)

[] Poller.MboxPoller.receive?msg -> Poller.Stations.select!i ->
WaitForReply(i,msg)
WaitForReply(i,msg) =
Poller.Stations.ack -> Poller.Stations.receive?msg -> Poll(i+1)
[] Poller.Stations.reject -> Poll(i+1)
[]

Process PollerMboxGlue = Empty

where

Empty = Poller.MboxPoller.check -> Poller.MboxPoller.nomsg
[] Dialer.MboxPoller.deposit?x -> Full(x)
Full(x) = Poller.MboxPoller.check -> Poller.MboxPoller.receive!x -> Empty
[] Dialer.MboxPoller.deposit?x -> Full(x)

Process DialerMboxGlue = Empty

where

Empty = Dialer.MboxDialer.check -> Dialer.MboxDialer.nomsg
[] Dialer.MboxDialer.deposit?x -> Full(x)
Full(x) = Dialer.MboxDialer.check -> Dialer.MboxDialer.receive!x -> Empty
[] Dialer.MboxDialer.deposit?x -> Full(x)
Process DialerComputation =
Dialer.MboxDialer.check -> {Dialer.MboxDialer.nomsg -> Computation
[]Dialer.MboxDialer.receive?x -> Dialer.Host.connect ->
Dialer.Host.send!x -> Dialer.Host.receive?y ->
Dialer.Host.disconnect -> Dialer.MboxPoller.deposit!y ->
Computation)

Calculation of the computation of the MasterStation. Computation of the composite component is CSP projection of the configuration process onto the keeping event set of the composite component ports. In this case, the keeping event set is comprised of the events of the form Dialer.Host.*, Poller.Stations.*.
Therefore, the projected behavior is:

Process MasterStationComposite = Poll(1)
where
Poll(nStations+1) = Poll(1)
Poll(i) = Poller.Stations.poll!i -> WaitForRequest(i)
WaitForRequest(i) = Poller.Stations.send?msg -> Dialer.Host.connect ->
Dialer.Host.send!x -> Dialer.Host.receive?y ->
Dialer.Host.disconnect -> Response(i)
[]Poller.Stations.reject -> Poll(i+1)
[]\√
Response(i) = Poller.Stations.select!i ->
WaitForReply(i, msg)
WaitForReply(i, msg) = Poller.Stations.ack -> Poller.Stations.receive!msg -> Poll(i+1)
[] Poller.Stations.reject -> Poll(i+1)
[]\√

After Bind renaming, the computation is:

Process MasterStationComposite = Poll(1)
where
Poll(nStations+1) = Poll(1)
Poll(i) = Stations.poll!i -> WaitForRequest(i)
WaitForRequest(i) = Stations.send?msg -> Host.connect ->
Host.send!x -> Host.receive?y ->
Host.disconnect -> Response(i)
[]Stations.reject -> Poll(i+1)
[]\√
Response(i) = Stations.select!i ->
WaitForReply(i, msg)
WaitForReply(i, msg) = Stations.ack -> Stations.receive!msg -> Poll(i+1)
[]Stations.reject -> Poll(i+1)
[]\√
Appendix D

Remote Procedure Call Example

The remote procedure call specification presented here is a complete specification of this connector. This example is used in Section 4.3.1 to demonstrate the calculation of the connector behavior.

```plaintext
Protocol DefinerProtocol = request?x->result!y -> DefinerProtocol /*Callee*/
Protocol CallerProtocol = request!x -> result?y -> CallerProtocol /*Caller*/

Connector RPCConnector
  Port Definer = DefinerProtocol
  Port Caller = CallerProtocol

Glue

Configuration
  /* ProcedureCall Connector Specification */
  Connector ProcedureCall
    Role Definer = DefinerProtocol
    Role Caller = CallerProtocol
    Glue Caller.request -> Definer.request ->
      Definer.result -> Caller.result -> Glue

  /* ClientStub Component Specification */
  Component ClientStub
    Port Client = DefinerProtocol
    Port Com = SenderProtocol
    Computation =
      Client.request?x->Com.send!x->Com.receive?y->Client.response!y -> Computation

  /* ComLink Connector Specification */
  Connector ComLink
    Role Sender = send!x -> Sender
    Role Receiver = receive?x -> Receiver
    Glue = Sender.send?x -> Receiver.receive!x -> Glue

  /* ServerStub Component Specification */
  Component ServerStub
    Port Service = CallerProtocol
    Port Com = ReceiverProtocol
    Computation =
      Com.send?x -> Service.request!x->Service.return?y -> Com.receive!y -> Computation

Instances
  CS:ClientStub
  SS:ServerStub
  Com:ComLink
  ClientPC: ProcedureCall
  ServerPC: ProcedureCall

Attachments
  ClientPC.Definer as CS.Client
  CS.Com as Com.Sender
  SS.Com as Com.Receiver
  SS.Service as ServerPC.Caller

End Configuration
Bind
  ClientPC.Caller to Caller
  ServerPC.Definer to Definer
End Bind
```
Appendix E

Event Bus 2

An event bus is a connector abstraction that allows for events to propagate between tools attached to the bus. We describe here an event bus implemented using a central component, often called Event Manager, to accept event messages and further forward them to the registered listeners. Note that the example "Event Bus" in Section 4.5.2.1 uses the composition of event channels to build an event bus abstraction. The underlying architecture of the two event buses is completely different.

A central component, called EventManager, is used to handle event registration and announcement of requests from the event bus participants. The event manager is responsible for collecting events from informers and routing events to registered listeners; it also handles registration of informers and listeners. The tools are indirectly connected via the central component; each tool uses a connector called ParticipantServer to connect to the EventManager component. All ParticipantServer connectors as well as the EventManager component are encapsulated in the composite event bus connector. The event bus connector specification is given in Figure 47.

```
Connector EventBus(nTools:1..)
  Role Participant(1..nTools)
Glue
Configuration
  ...
Instances
  EM: EventManager
  EC(1..nTools): EventConnector
Attachments
  for i:1..nTools
    EM.Participant(i) to EC(i).Server
  end for
End Configuration
Bind
  for i:1..nTools
    EC(i).Server to Participant(i)
  end for
End Bind
End Connector
```

Figure 47: The event bus implementation using the event manager component

---

16 For general description of the event bus abstraction, refer to Section 4.5.2.1.
Participant Interface. The protocol of registration has also changed comparing to the "Event Bus 1". Tools register first before they can announce or observe any events. Further, tools communicate by announcing and observing events. Any tool can at any point deregister as informer or as listener. After a tool has deregistered, it can not reregister again. Once all tools deregistered their interest in communication, communication successfully terminates.

```c
/*
ParticipantProtocol describes Participant's role
in Participant-Event Manager communication.
*/

Protocol ParticipantProtocol = RegisterAsInformer( ) where
    RegisterAsInformer( I ) = ([\text{I}:\text{E}=\text{I} \land \text{registerInformer}(I) \rightarrow RegisterAsInformer(I{\text{U}}(e))])
    \Pi \text{RegisterAsListener}(I,() =
    ([\text{I}:\text{E}=\text{LU},I \land \text{registerInformer}(I) \rightarrow RegisterAsInformer(LU{\text{U}}(e))))
    \Pi \text{Operate}(I,L)

Operate(I,L) = ([\text{I}:\text{I} \land \text{announce}(I,E) \rightarrow \text{Operate}(I,L))
    \Pi ([\text{I}:\text{L} \land \text{listen}(I,L) \rightarrow \text{Operate}(I,L))
    \Pi ([\text{I}:\text{I} \land \text{deregisterInformer}(I,L) \rightarrow \text{Operate}(I,L{\text{U}}(e))])
    \Pi ([\text{I}:\text{L} \land \text{deregisterListener}(I,L) \rightarrow \text{Operate}(I,L{\text{U}}(e))])

/*
Protocol that describes Event Manager's role in
Participant - Event Manager (Server) communication
*/

Protocol ParticipantServerProtocol = RegisterInformer( ) where
    RegisterInformer( I ) = ([\text{I}:E = \text{I} \land \text{registerInformer}(\text{I}) \rightarrow RegisterInformer(I))
    \Pi \text{RegisterListner}(I,()) =
    ([\text{E}:E=\text{LU} \land \text{registerListner}(\text{I},\text{L}) \rightarrow RegisterListner(LU{\text{U}}(e)))
    \Pi \text{Operate}(I,L)

Operate(I,L) = ([\text{I}:\text{I} \land \text{announce}(I,E) \rightarrow \text{Operate}(I,L))
    \Pi ([\text{I}:\text{L} \land \text{listen}(I,L) \rightarrow \text{Operate}(I,L))
    \Pi ([\text{I}:\text{I} \land \text{deregisterInformer}(I,L) \rightarrow \text{Operate}(I,L{\text{U}}(e))])
    \Pi ([\text{I}:\text{L} \land \text{deregisterListener}(I,L) \rightarrow \text{Operate}(I,L{\text{U}}(e))])

/*
ParticipantServer describes communication between
the Event Manager and each of the participants
*/

Connector ParticipantServer
  Role Participant = ParticipantProtocol
  Role Server = ParticipantServerProtocol
  Glue Participant.announce(e) -> Server.announce(e) -> Glue
                  \Pi ([\text{I}:\text{E} \land \text{registerInformer}(\text{I}) \rightarrow Server.registerInformer(I))
                  \Pi ([\text{I}:\text{E} \land \text{deregisterInformer}(\text{I}) \rightarrow Server.deregisterInformer(I))
                  \Pi ([\text{I}:\text{E} \land \text{registerListner}(\text{I}) \rightarrow Server.registerListner(I))
                  \Pi ([\text{I}:\text{E} \land \text{deregisterListner}(\text{I}) \rightarrow Server.deregisterListner(I))
                  \Pi ([\text{I}:\text{E} \land \text{listen}(\text{I}) \rightarrow Participant.listen(I))
                  \Pi ([\text{I}:\text{E} \land \text{listen}(\text{I}) \rightarrow Server.listen(I))
```

82
Event Manager. An event manager is a component that coordinates event communication between event bus participants. It collects event announcements from informers and routes them to the registered listeners. First, EventManager allows for any informer or listener to register. Following successful registration, EventManager starts to accept events and forward them to the registered listeners. During the operation of the system, EventManager does not accept any additional registration. Any participant can either deregister or announce/listen to events. Once all informers have deregistered, announcements can not be made, only deregistrations. Finally, once all components have issued deregistration for all their announced and observed events, EventManager process successfully terminates. The specification of the EventManager component is given in Figure 48.

Component EventManager(nTools:1..)
  Port Participant(1..nTools) = ParticipantProtocol
  Computation = Register([],[]); (Deregister([],[]) || Operate) where
  Register(I,L) =
  [[[i,e]:[1..nTools]xE-I • registerInformer!e -> Register(IU({i,e}),L)]
  [[[i,e]:[1..nTools]xE-L • registerInformer!e -> Register(LU({i,e}))]
  Deregister(I,L) =
  [[[i,e]:I • Participant(i).announce?e -> where?L]
  [[[i,e]:I • deregisterInformer!e -> Deregister(I-{{i,e}},L)]
  [[[i,e]:L • deregisterListener!e -> Deregister(I,L-{{i,e}})]
  Deregister([],[]) = ∅
  Operate = (Πe:E • announce!e -> where?L ->
  (:i:((i,e) ∈ L) • (Participant(i).deregister?e ->
  Π Participant(i).listen!e) -> ∨ ))

Figure 48: Event Manager Specification

The event manager is a component that is hidden inside the event bus abstraction. The tools are not directly aware of any other components that are built in the bus abstraction. The information about it is embedded in the connector. At a high level of abstraction, it appears as a symmetric connector with a number of participant roles. This gives designer more freedom in terms of event bus internal implementation. Although initially implemented using a component and connectors that allow tools to attach to the bus, at some point, instead of the centralized component only, the bus might change the underlying architecture. For example, two components, Event Registrar and Event Router could be used instead of just one - Event Manager; or the bus can be implemented using multiple communicating event managers. Using composite bus architecture, the architecture of the system that uses event bus does not change to follow the changes in the bus architecture. The change is hidden behind the curtains of the event bus interfaces.