APOC AND ADE: THEORY AND PRACTICE IN THE DESIGN OF
ARCHITECTURES FOR BEHAVIOR-BASED AGENTS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Virgil Andronache, B.S., M.S.C.S.E.

----------------------------------------

Matthias Scheutz, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
June 2004
© Copyright by
Virgil Andronache
2004
All Rights Reserved
APOC AND ADE: THEORY AND PRACTICE IN THE DESIGN OF
ARCHITECTURES FOR BEHAVIOR-BASED AGENTS

Abstract

by

Virgil Andronache

In this thesis we present an integrated theoretical and practical approach towards the
development of complex robotic agents. We describe APOC, an architecture framework
intended for the analysis and implementation of complex agent architectures. We then
show how APOC can be used to implement agent architectures in various architectural
design paradigms and how these designs can be analyzed in APOC. Next, we show how
APOC can be used to introduce new elements into architecture design, by creating ar-
chitectures which modify as the agent interacts with its environment. The practical side
of the research is presented next, with a description of the APOC Development Environ-
ment, ADE. We show how ADE functionality is based on the properties of the APOC
framework and present in detail the features ADE presents to the agent designer and ar-
chitecture developer. Examples of these features are shown for illustration. In the last part
of the thesis we focus on work done using APOC and ADE towards the development of
a robotic waiter. Here we discuss the main problems which need to be solved in the de-
velopment of a complex agent and present ADE solutions to these problems. We next
introduce the overall structure of the robowaiter architecture and show in detail some of
the main sub-systems of this architecture. Results of experiments which form the build-
ing blocks for the robowaiter architecture are then presented. The thesis concludes with
a brief discussion of future work using ADE, including the upcoming implementation of the robowaiter.
To everyone who has helped me get here. Δοξα τω Θεο.
CHAPTER 5: APOC AS A DESIGN TOOL .................................................. 59
   5.1 Architectural Mechanisms for Behavior Selection ....................... 60
      5.1.1 Cooperative versus Competitive Behavior Selection ............... 63
      5.1.2 Implicit versus Explicit Behavior Selection ...................... 64
      5.1.3 Non-Adaptive versus Adaptive Behavior Selection ............... 65
   5.2 The Argument for Dynamic Changes of Behavior Selection Strategies ..... 66
      5.2.1 Case 1: Selection of Sensory Information ....................... 67
      5.2.2 Case 2: Emergency Responses .................................. 67
      5.2.3 Case 3: Infeasible Behaviors .................................. 68
      5.2.4 Case 4: Extending the Behavioral Repertoire ................... 68
      5.2.5 Case 5: Mappings between Context and Behavior Selection Strategies ........................................................................... 69
      5.2.6 Case 6: Attentional Mechanisms .................................. 70
      5.2.7 Case 7: Learning Behaviors ..................................... 70
   5.3 Dynamic Behavior Selection Mechanisms .................................... 71
      5.3.1 Switching from Cooperative to Competitive ...................... 71
      5.3.2 Switching from Competitive to Cooperative ...................... 73
      5.3.3 Multiple Behavior-Selection Mechanisms in APOC ................ 74
   5.4 Expressing Other Concepts in the APOC Framework ........................ 75
      5.4.1 Developing Architectures ....................................... 75
      5.4.2 Environment Driven Changes in ART Networks .................. 77
      5.4.3 Incremental, Resource Constrained Planning ..................... 79
      5.4.4 Use Case Map Example ..................................... 83
      5.4.5 Cellular Automata Simulation .................................. 85
   5.5 Hierarchies in Non-Hierarchical Architectures .......................... 89
      5.5.1 Uses of Hierarchical Information ............................... 90

CHAPTER 6: THE ADE DEVELOPMENT ENVIRONMENT ............................. 93
   6.1 Background ............................................................................ 93
      6.1.1 The Basic Characteristics of ADE ................................ 93
      6.1.2 Other Agent Tools Compared to ADE ............................. 95
   6.2 ADE Building Blocks I: The APOC Framework ............................ 97
      6.2.1 ADE Components ............................................. 98
      6.2.2 ADE Links .................................................... 100
   6.3 ADE Building Blocks II: The User Interface ................................ 105
      6.3.1 Architecture View ........................................... 105
      6.3.2 Virtual Machine View ....................................... 109
      6.3.3 Other Functionality ......................................... 109
      6.3.4 Operating Modes ........................................... 110
   6.4 ADE Building Blocks III: The Supporting Environment ................ 111
9.2.4 Experiment 3 .................................................. 168
9.2.5 Setup in ADE .................................................. 170
9.3 Visual tracking .................................................. 172
  9.3.1 Background .................................................. 173
  9.3.2 The System Architecture .................................. 174
  9.3.3 The Attentional Subsystem ................................ 175
  9.3.4 The Motion Tracker ......................................... 176
  9.3.5 The Blob Selection and Camera Control ............... 177
  9.3.6 Experimental Set-up ....................................... 177
  9.3.7 Experiments ................................................ 178
  9.3.8 Implementation Details and Optimization Measures 181
  9.3.9 System Development ..................................... 183
  9.3.10 Summary ................................................... 185
9.4 Reference Resolution ........................................ 186

CHAPTER 10: CONCLUSION ....................................... 192

BIBLIOGRAPHY ..................................................... 195
FIGURES

3.1 The components of the RCS control paradigm: WM is the current world model, VJ is the value judgment system, BG is the behavior generation module, and SP is the sensory processing module. .......................... 15

3.2 A generic CAMPOUT architecture .................................. 18

3.3 The structure of an APOC component ................................. 22

4.1 Type diagram, initial instantiation, state after first request, and final state of sample APOC architecture ......................................................... 31

4.2 Type and instance of a fully connected, layered neural network in APOC 32

4.3 Sample APOC implementation of cooperative action selection: type specification and one of the potential architectural states ................. 36

4.4 An example of an APOC translation of a GRL structure ............ 38

4.5 The architectures modelled in APOC as discussed in Sections 4.2.1 through 4.2.9 - Part I ................................................................. 42

4.6 The architectures modelled in APOC as discussed in Sections 4.2.1 through 4.2.9 - Part II ................................................................. 43

4.7 The SOAR architectural structure ....................................... 49

4.8 The ACT-R architectural structure ..................................... 51

4.9 An example of an APOC translation of an ICARUS structure .... 54

4.10 The PRODIGY system with its component parts .................. 57

4.11 The structure of a PRODIGY plan ..................................... 58

5.1 Examples of an infeasible combination of feasible behaviors (go left and go down for agent A) and a feasible behavior from combination of infeasible behaviors (go right and go down for agent B) in order to get to the goal state represented by the black circle. White circles denote obstacles. ........................................ 68
5.2 Dynamic change from cooperative to competitive behavior selection: the type specification is shown on top, a standard cooperative system (A2) is on the lower left, the dynamic system (A1) in its default, cooperative state is on the lower middle, and the competitive state of the system on the lower right.  

5.3 Dynamic change from competitive to cooperative behavior selection: the type specification is shown on top, a standard competitive system (A4) is on the lower left, the dynamic system (A3) in its default, competitive state is on the lower middle, and the cooperative state of the system on the lower right.  

5.4 Dynamic change between any two behavior selection mechanisms: the type specification is shown on top, the competitive state of the architecture is on the lower left, and the cooperative state on the lower right.  

5.5 ART networks in APOC. See text for an explanation of the notation.  

5.6 Sensory processing example - final architectural state  

5.7 Sensory processing example showing the final architectural state with resource-constrained environmental inputs.  

5.8 Planning example: specification  

5.9 Planning example - sample instantiation  

5.10 Example of a Black Box UCM Representation in APOC  

5.11 White box UCM example  

5.12 Game of life example  

5.13 Game of life - sample APOC initial configuration  

5.14 Game of life: APOC configuration  

5.15 Game of life: Live-cell configuration  

6.1 ADE Interface showing a type-level description and a run-time view of the architecture  

6.2 APOC Component Specification Prompt. The three fields indicate the component class, the number of components present in the initial virtual machine, and the maximum number of components simultaneously present in the virtual machine  

6.3 ADE Link Specification Prompt, showing customization options for each link type.  

6.4 The relationship among ADE components in a generic set-up for a robotic agent.
7.1 Node information display for a robot representation. Buttons are available to select the display of various robot sensors ................................. 122

7.2 Camera panel displaying the original picture (left) and the post-processed image, the contour of the ball (right) ................................. 123

7.3 O-link information GUI (left) and data display (right) ....................... 124

7.4 Sample graph data in ADE ......................................................... 125

7.5 **Client request for connection to a server:** 1. The client contacts the AgeSRegistry (running on some host) and requests a connection to a server. 2. The AgeSRegistry locates the server and relays the request. 3. The server returns a remote “stub” object. 4. The AgeSRegistry relays the remote object to the client. 5. The client and the server maintain a direct connection. ......................................................... 128

8.1 The schematic of the “Greeting” behavior for a robot waiter. ............... 137

8.2 The schematic of the “Go to Patron” behavior for a robot waiter. ........... 138

8.3 A high-level view of the architecture for a robot waiter. .................... 145

8.4 The schematic of the “Take Order” behavior for a robot waiter. ............ 150

8.5 The schematic of the “Deal with Customer” behavior for a robot waiter. .. 150

8.6 The schematic of the “Clean Area” behavior for a robot waiter. ............ 151

8.7 The schematic of the “Plan and Go” behavior for a robot waiter. ............ 152

8.8 The schematic of the “Stock Table” behavior for a robot waiter. .......... 153

9.1 ADE Initial architecture for web search agents (top) and the final architecture of the system (bottom) ......................................................... 157

9.2 The robot on its way to the target location (orange ball). .................... 160

9.3 Architectural description for the robotic experiments. From left to right: the generic architectural set-up and the individual architectures for experiments 1, 2, and 3 ......................................................... 161

9.4 First robot test path ........................................................................ 163

9.5 Second robot test path ...................................................................... 163

9.6 The robot’s trajectory for different setups for experiment 2: with centered target behind the passage way (left) and the target off to one side, without (middle) and with supervisory control (right) ........................... 167

9.7 Robot environment for experiment 3 showing the robot path ............... 169
9.8 The system architecture for ball tracking in a robotic agent . . . . . . . 174
9.9 Camera images and tracker information for stationary camera tracking:
unprocessed image (top), tracker information (bottom) . . . . . . . . . 179
9.10 Camera images and tracker information for mobile camera tracking:
unprocessed image (top), tracker information (bottom). Two trackers are
present in some images; the main blob is moving to the left . . . . . . . 180
9.11 Control panel for dynamic color range adjustment . . . . . . . . . . . 184
9.12 Control panel for dynamic adjustment of PID controller parameters . . 184
9.13 Two-referent set-up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
9.14 One-referent set-up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
9.15 A high-level view of the components employed in the architecture used
in the reference resolution task. . . . . . . . . . . . . . . . . . . . . . . . . 187
9.16 The original state of the reference resolution system - virtual machine
view and robot perception . . . . . . . . . . . . . . . . . . . . . . . . . . 189
9.17 The state of the reference resolution system after “PUT” . . . . . . . 189
9.18 The state of the reference resolution system after “THE RED OBJECT” . 190
9.19 The state of the reference resolution system after “ON THE YELLOW
OBJECT”. The object to be moved is identified . . . . . . . . . . . . . . 190
9.20 The state of the reference resolution system after “ON THE BLUE OB-
JECT”. The target is identified . . . . . . . . . . . . . . . . . . . . . . . . 191
TABLES

5.1 EXAMPLES OF BEHAVIOR-SELECTION STRATEGIES CLASSIFIED

ALONG THE THREE PROPOSED DIMENSIONS: COMPETITIVE VS. COOPERATIVE, EXPLICIT VS. IMPLICIT, AND ADAPTIVE VS. NON-ADAPTIVE.

9.1 THE TIMES AND CUMULATIVE NUMBERS OF DELETED NODES FOR CONSECUTIVE RUNS IN TWO TYPICAL SETUPS OF EXPERIMENT 1.

9.2 THE TIMES AND NUMBERS OF DELETED SONAR NODES FOR RUNS IN THE FIRST SETUP OF EXPERIMENT 1.

9.3 THE TIMES AND NUMBERS OF DELETED SONAR NODES FOR RUNS IN THE SECOND SETUP OF EXPERIMENT 1.

9.4 THE AVERAGE TIMES TO TARGET AND CONFIDENCE INTERVALS FOR EACH OF THE 10 RUNS WITH 0, 1, AND 2 COMPONENTS DELETED FOR THE TWO TRAVERSING SCENARIOS OF ROBOT EXPERIMENT 1.
ACKNOWLEDGMENTS

I would like to thank everyone who has contributed towards the development of APOC, ADE, and the robowaiter system. First, Dr. Matthias Scheutz for his guidance, his work on the NNSIM project which formed the starting point for ADE, and for all the other things, too numerous to mention. Thanks to the other members, past and present, of the AIROLAB: Dan Dentinger for implementing several functions in the original NNSIM which found their way into ADE; Jim Kramer for his work on the Ages system on which ADE relies, his work on robot navigation which will be a part of the robowaiter and the fruitful discussions about the robowaiter; Paul Schermerhorn for his help with many little things over the years, as well as his suggestions about the robowaiter; Kyle Wheeler for getting the robots going, implementing camera code, and starting work on the registry.

I would also like to thank the undergraduate students who have contributed towards the robowaiter project: David Anderson, Holden Bonwit, Peter Bui, and Patrick Davis.
CHAPTER 1

PROLOGUE

This thesis is a description of APOC: a theoretical framework for the design, implementation, testing, and deployment of agent architectures, and ADE, an agent development environment which embodies the precepts outlined by the framework.

The creation of an agent usually requires the designer to answer several questions: whether higher-level cognitive functions such as planning or case-based reasoning should be part of the agent, whether the agent should employ a hierarchical architecture, e.g., subsumption [27], or a flat one, e.g., the agent network architecture [67]; for a behavior-based agent, whether its behavior-arbitration mechanism should be competitive, e.g., bayesian decision analysis [103], or cooperative, e.g., schema-based [12]; whether the architecture will run on a single computer, or be distributed over a network. The answer to each question depends, among other things, on the complexity of the agent, the available resources, and the preferences of the designer. Simple agents, for example, can be implemented equally efficiently in competitive and cooperative systems. However, as more complex agents are created and issues such as attentional mechanisms and modelling of mental processes are explored, a single arbitration mechanism may not be suitable.

This work is divided in three parts. In the first part, we present a theoretical architecture framework called APOC. We show the generality of the framework by modelling various behavior-based and cognitive architectures in the APOC formalism and we illustrate the use of APOC in analyzing and designing agent architectures. The second part of
the thesis presents the ADE software tool. ADE implements the theoretical concepts of APOC and allows the agent developer to create agents in both single- and multi-computer environments. The third part of the thesis presents theoretical considerations and experimental work towards the development of a complex robotic agent in the context of a robotic waiter.

The APOC framework provides the tools which allow an agent developer to navigate the first three choices presented above. For example, APOC allows the design of an agent with case-based reasoning, a hierarchical architecture, and a competitive behavior-arbitration mechanism. APOC also provides for the design of an agent with no higher-level cognitive functions, a flat architecture, and a cooperative behavior-arbitration scheme while still providing them with a basic structure on which to base their designs. APOC provides a common infrastructure within which all design methodologies and arbitration mechanisms can be expressed. By expressing mechanisms as diverse as neural networks, behavior-based architectures, and cognitive architectures in a unified manner, the combination of previously incompatible mechanisms becomes possible, opening the door for the creation of increasingly complex agent architectures. This allows the agent developer to use each mechanism in those areas where it is best suited. Thus, a symbolic reasoner can be provided with a behavior-based implementation of the behaviors it reasons about. The reactive nature of behavior-based systems can then provide quick response times to environmental changes while the reasoner provides the system with long term goal-directedness.

The ADE environment provides the tools which allow the user to create architectures in both a single- and multi-computer environment. Coupled with the options given by APOC, ADE allows the agent designer to implement the first agent described above in a multi-computer system, while the second agent could be implemented in a single-computer system. ADE implements the theoretical concepts of APOC and is there-
fore an illustration of the feasibility of the APOC approach. In addition to providing an implementation of APOC theory, ADE provides a flexible single-computer and multi-computer environment for agent development. ADE also provides basic graphical tools and support for user-defined extensions. These allow users to define graphical tools which are targeted towards specific components, allowing for easy inspection of those components, as well as the modification of their run-time parameters.

I would like to thank the following people for their help with various parts of the work presented here:

- Dr. Matthias Scheutz. Dr. Scheutz and I have worked together on the development of the APOC framework and his NNSIM tool provided the basis of the ADE GUI. He has also provided the blob and color-detection code for the experiments described in Chapter 9 and the figure describing the structure of an APOC component.

- Dan Dentinger. Dan has implemented several components of the interface, most notably the graph tool and the graphical buttons.

- James Kramer. Jim has worked on the AgeS system, which forms the basis for the distributed nature of ADE, the registry, and a software tool for robot navigation which interfaces higher-level route planning with low-level obstacle avoidance. The tool will be incorporated in the robowaiter architecture, although it was not used in the experiments presented here.

- Paul Schermerhorn. Paul, Jim, and I have had several fruitful discussions about the robowaiter project. Some of the ideas presented in Chapter 8 had their beginnings in those discussions.

- Kyle Wheeler. Kyle wrote code interfacing the robot cameras to an early version of ADE and started the work on the registry.

- David Anderson, Holden Bonwit, Peter Bui, Patrick Davis. The undergraduate students have worked or are currently working on robot navigation, GUI improvements, sound recognition, and speech production as part of the effort to complete a first version of the robowaiter by July 2004.

The structure of this thesis is as follows:

1. The APOC framework description. This section provides the theoretical definition for the constituents of the APOC framework and their interactions. The APOC framework was developed jointly with Dr. Matthias Scheutz. The set-theoretic description of an APOC component provided in Chapter 3 and the graph describing an APOC component were developed mainly by Dr. Scheutz following a series of
discussions in which we defined component functionality. Sections of Chapter 3 were published in the AAAI Spring Symposium 2003 [93], FLAIRS 2003 [94] and Agent Theory to Agent Implementation 2004 [10].

2. Uses of the APOC framework. Chapter 4 is dedicated to descriptions of behavior-based architectures, design methodologies, and cognitive architectures in terms of APOC components. Some of these translations have appeared in MAICS 2002 [9] and FLAIRS 2003 [94] or will appear in AAMAS 2004 [11]. Chapter 5 shows sample combinations of cooperative and competitive action-selection mechanisms, as well as applications of APOC to concepts on the fringes of agent design, such as cellular automata.

3. The ADE development environment. This section provides implementational details about ADE. The parallels between APOC theoretical constructs and ADE features are highlighted and additional features of ADE are presented. A detailed description of ADE including several sections of this chapter, will appear in the International Journal for Artificial Intelligence Tools [8].

4. Uses of ADE. Interfaces to various robots are described and graphical tools which facilitate the creation of robotic agents are presented in Chapter 7.

5. The requirements for the design and implementation of complex robotic agents. Complex agent design using APOC and ADE will be discussed in Chapter 8 in the context of a robotic waiter.

6. Experimental results. In this section, agent systems developed in ADE are described. The characteristics of APOC which aided in the development and testing of the systems are identified. ADE features used in the systems are described in the context of agent architecture development. The exception to the above is the experiment using simulated agents in a virtual world, which was developed by Dr. Scheutz in SimAgent, following a jointly developed architectural design. The reference resolution experiment was designed by Dr. Scheutz together with Dr. Eberhard from the psychology department. Experiments from Section 9.2 will appear in AAMAS 2004 [11]. The reference resolution experiment will be presented as as part of the AAAI Intelligent System demonstration at AAAI 2004.
CHAPTER 2

INTRODUCTION

Artificial intelligence (AI) efforts to design control systems for artificial agents are increasingly relying on research in the field of agent architectures. Various architecture schemes and design methodologies have been proposed, which focused on different aspects of agent control. For example, behavior-based architectures, e.g., subsumption and schema-based, [14,27], started out as control systems for embodied agents and contained no symbolic representation. At the other end of the spectrum, cognitive architectures such as SOAR and ACT-R [7,61] provided higher-level cognitive functions, but were not designed to work with embodied agents.

By nature, cognitive architecture are not targeted towards a particular agent or type of agent. However, that is not the case with behavior-based architectures, most of which were developed for particular kinds of agents or targeted at a particular class of tasks. For example, each Braitenberg vehicle had a specific architecture [26], Myrmix was developed specifically as a foraging agent [38] and Xavier as an office delivery robot [96].

Many of these architectures have proven successful in their application domain, as Myrmix and Xavier illustrate. It would be advantageous if components and principles which contributed to their success could be reused in other circumstances. An especially interesting question is whether these principles and components could be utilized in other

---

1In other related disciplines, e.g., cognitive psychology or philosophy, agent architectures have played a fundamental role for an even longer time, under the name “functional architecture,” in the analysis of the organization of control systems of living creatures.
architectures that do not use the same basic components or design methodology, e.g.,
integrating an efficient subsumption-based “go-to-goal” module into a DAMN-based ar-
chitecture [87]. Additionally, as behavior-based agents become increasingly complex and
cognitive architectures move towards use in embodied agents, e.g., ICARUS [63], using
both types of mechanisms in the same architecture becomes almost a necessity. However,
agent developers are faced with several problems in attempting to integrate code from
multiple architectures.

A first problem is credit assignment: it may be difficult to say what part of an architec-
ture or design accounts for its success. A second arises from the difficulty of comparing
two different architecture types directly, because their design assumptions and domain
restrictions may vary significantly, e.g., symbolic versus “sub”- or non-symbolic, high-
level versus low-level, serial vs. parallel, software versus robotic agents. A third problem
is that the characteristics which impede direct comparison also hinder the combination of
mechanisms into a unitary architecture. Consequently, it is difficult if not impossible not
only to assess the advantages and disadvantages of particular designs and methodologies
but also to utilize them in other designs without a common language or framework in
which architectures could be compared.

Another problem, specific to current behavior-based architectures, is that the mech-
anisms used for behavior selection are typically fixed. While it may be possible to
adjust some of the mechanisms’ parameters to make them more adaptive, they cannot
be changed altogether. A subsumption-based architecture [27], for example, cannot be
changed into a schema-based architecture [12] at the level of the architecture. Switching
among different behavior-selection strategies, however, may be desirable or even required
at times, either to increase the system’s performance or to enable the system to achieve
a given task in the first place. Looking at biological creatures, it seems that many ani-
mals are capable of modifying their behavior-selection strategies, typically as a result of
some learning process, which process then generally leads to better performance at the
given task [64]. Furthermore, they seem to be able to switch dynamically among different
behavior-selection strategies depending on which strategy leads to the best results.
It would seem natural to allow for a dynamic change of behavior-selection strategies in
behavior-based systems as well.

There are, however, three problems with dynamic changes of behavior selection strategies:
(1) current behavior-based architectures do not support multiple simultaneous behavior-
selection processes among which the system can switch, (2) it is not clear which behavior-
selection strategies should be included in the design of the system, among which it will
then be able to switch; and (3) there are no universally valid criteria for deciding under
what circumstances it is beneficial to switch between two conflicting strategies.

Thus, what is needed is an architecture framework which is general enough to al-
low researchers to evaluate and compare different kinds of architectures, but at the same
time is conceptually parsimonious enough to employ only a few intuitive, basic concepts
which can be used to define the concepts used in other architectures. Otherwise the frame-
work may end up as complex as any of the higher, universal programming languages in
which agent architectures are defined—obviously, such a framework would be of little
use in agent architecture research. One main goal for such a framework is to achieve
high expressiveness at different levels of abstraction. To our knowledge, no satisfactory
framework is available yet, although certain characteristics pertaining to the concept of an
“architecture framework” appear in various places in the literature. For example, some of
the ideas used in building our framework find a certain degree of parallelism in work by
Horswill [50] and Lyons and Arbib [66], e.g., GRL provides a framework for the imple-
mentation of a variety of behavior-based architectures, while the RS model provides an
environment of components with synchronous processes and links which connect com-
ponents via specially provided connection points called “ports.”
As a step towards the development of a general framework for agent architectures, we introduce the APOC architecture framework, which attempts to respect the requirements outlined in the previous paragraph. APOC is not only intended as a theoretical framework, which allows researchers to analyze, evaluate, and compare agent architectures, but also, complemented by ADE, the APOC development environment, it functions as a practical tool for the design of complex agents. In particular, APOC/ADE is tailored towards architectures which allow for specifications of architecture modifications and resource constraints at the architecture level.

As part of this thesis, we use APOC to model several behavior-based and cognitive architectures as a prelude to direct comparisons among architectures. We then propose a solution for the first problem of dynamic modification of behavior-selection strategies and show an approach for finding solutions to the other two. As it is not possible to investigate the utility of dynamically changing strategies if they cannot be implemented together in one architecture, we show how APOC can be used to define and study any combination of behavior-selection mechanisms, e.g., a subsumption-based architecture and a schema-based one. Specifically, we argue for the utility of such a framework, especially for the study of dynamically changing behavior-selection strategies, and demonstrate that such dynamic changes can be beneficial in different tasks using simulated and robotic behavior-based agents. We conclude the presentation of APOC with an illustration of its use in modelling various concepts related to agent-design, such as ART learning networks [32].

As we show in the first part of the dissertation, APOC is in itself a useful tool for agent developers. However, in order to be able to implement APOC-based agents, we need a software tool which puts into practice the flexibility of the APOC framework. Additionally, for generality, the tool should also support both single- and multi-computer systems and single- and multi-agent systems.
2.1 The ADE Software Tool

In recent years, several toolkits and frameworks have been proposed which are intended to support either the design of multi-agent systems, e.g., JADE [22], RETSINA [102], AGENTBASE [4], ZEUS [76], or the design of agent architectures for single agents, e.g., SimAgent [99], ARIA/Saphira/Colbert [56–58], Player/Stage [43, 44]. Currently, there are no systems available that combine and integrate these two realms. To bridge the gap between multi-agent system frameworks and agent architecture toolkits for single virtual and robotic agents, we propose the agent architecture development environment ADE, which provides a homogeneous, user-friendly environment for the development of architectures for virtual and robotic agents in single and multi-agent settings.

Multi-agent systems typically provide the distributed infrastructure which allows agents to reside on different hosting computers and move from host to host in a way that is hidden from the user. For example, JADE [22] provides a communication language, a graphical user interface for controlling and monitoring agents, and a directory facilitator which provides services needed to allow agents to contact one another and communicate regardless of their locations in the system. Furthermore, some of the toolkits also support distributed agents, i.e., agents residing on multiple hosts at the same time [102]. These systems are typically implemented as middleware that provides APIs for the agent designer. Yet, because these systems are intended as a framework in which to develop the multi-agent system, they provide few, if any, tools required for the development of the architecture of an agent.

Single-agent systems, on the other hand, focus on support for the development of the agent’s architecture, typically by providing libraries for common agent functionality, e.g., condition-action rule interpreters [98] or basic components of a particular architecture such as the BDI architecture [36]. They may also provide additional functionality for running multiple virtual agents, as in SIM-Agent [98] or Swarm [70], or for operating
robots, e.g., Saphira [58]. These systems are usually designed for either virtual agents or for robotic agents, and in the latter case typically only for single agents. Furthermore, these systems do not provide tools to implement and run multiple distributed agent architectures, as would be possible in a multi-agent system.

Of highest importance for our purposes is that most architecture development toolkits either provide libraries which must be integrated into code by agent designers or are based on a particular architecture design. While the former setup is flexible and allows for the specification of a variety of architecture types, the agent architecture designer will have to acquire detailed knowledge of the libraries and their underlying assumptions to be able to use the libraries effectively. Furthermore, a significant amount of programming is required to design even simple agents. Although the toolkits based on specific architecture designs alleviate this problem by providing a specific architecture paradigm in which new architectures can be designed with more ease, this approach makes it impossible to implement architecture types that are not based on the given paradigm.

We believe that versatile agent architecture design tools should be open with respect to the employed architecture paradigm, and should allow for a design environment for architectures that is appealing to users and easy to use, rather than requiring designers to understand possibly very complicated library calls. Furthermore, tools should also allow for distributing the architecture over multiple hosts while allowing the user to manipulate the architecture as if it were running on a single machine. This makes it possible to run complex, computationally demanding architectures in parallel, reducing the overall computation time. Finally, tools should allow for the design of virtual and robotic agents alike, as well as single and multi-agent systems, in which the same architecture could control either a robot or a virtual agent in a single- or multi-agent simulation environment without having to restructure the architecture or recompile the control code.

Based on the requirements discussed above, we propose ADE, an integrated architec-
ture development tool for both virtual and robotic, single and multi-agent systems. ADE provides a combination of features unavailable in other toolkits. These features include a fully graphical interface, a distributed development environment which allows for interactive design of possibly distributed single and multi-agent architectures and support for any design methodology.

In this chapter we introduced the motivation behind the development of the APOC architecture framework and the ADE agent-development tool. The next chapters present a detailed look at the APOC framework, its properties and its uses.
The APOC architecture framework is the theoretical foundation on which all the work in this thesis is based. In the following, we present related research, define the concept of an architecture framework, describe APOC, and illustrate the properties of our framework.

3.1 Background

We begin by describing some of the research which shows the emergence of the need for a unified theoretical framework for the implementation, design, and analysis of complex agents. The works in this section can be roughly divided into two categories: those which make explicit theoretical claims related to our framework, and applications whose underlying design principles illustrate some of the concepts present in APOC.

The idea of an architecture framework as a tool for agent architecture analysis and design has not been extensively pursued. However, some of the characteristics present in a framework, as described in the Introduction, can be found in several places in the literature. These works can be broken down into two categories: frameworks and design methodologies, such as RS [66] and RCS [6], and flexible architectures, such as GRL [50] and CAMPOUT [86].
3.1.1 RS

The RS system proposed a model of computation for sensory-based robotics, based on interaction among “concurrent computing agents.” RS provides a generic specification of these agents through schemas. A basic schema in RS is defined as [66]:

```
basic-schema ::= [ Schema-Name: ⟨N⟩
    Input-Port-List: ⟨⟨Iplist⟩⟩
    Output-Port-List: ⟨⟨Oplist⟩⟩
    Variable-List: ⟨⟨Varlist⟩⟩
    Behavior: ⟨⟨Behavior⟩⟩
]
```

where
- N is an identifying name for the schema;
- Iplist, Oplist are lists of ⟨Portname⟩ : ⟨Porttype⟩ pairs for input and output ports, respectively;
- Varlist is a list of ⟨Varname⟩ : ⟨Varname⟩ pairs for all internal variables; and
- Behavior is a specification of computing behavior.

Schemas can be instantiated to create the schema instances, which are the actual agents. Communication among instantiated schemas is performed through universal message-passing links, which connect from an output port of a schema instance to the input port of another schema instance. Ports are associated with data types and only ports associated with the same type can be connected. Communication can be performed both synchronously and asynchronously.

Schemas can be grouped to form assemblages called complex schemas. These are specified as:

```
assemblage-schema ::= [ Assemblage-Name: ⟨N⟩
    Input-Port-List: ⟨⟨Iplist⟩⟩
    Output-Port-List: ⟨⟨Oplist⟩⟩
    Variable-List: ⟨⟨Varlist⟩⟩
    Network: ⟨⟨Network⟩⟩
]
```
where \( N \), \( \text{Iplist} \), \( \text{Oplist} \), and \( \text{Varlist} \) have the same meaning they have in individual schemas and \textit{Network} is a structure which creates and connects the schema instances that form the assemblage.

The RS definitions of computational elements, communication procedure, and assemblage formation are very general. Therefore, a number of robotic architectures, e.g., schema-based architectures, can be expressed in RS. However, more complex architectural processes, e.g., reducing the retrieval time of a piece of data if that data is frequently accessed, as happens in ACT-R, would be very difficult to implement within the structure of the RS model.

APOC incorporates the generality and flexibility of RS while providing additional features which make it more suited for generic agent design. Among the additions APOC brings to the RS formalism are:

- The option of performing operations on data within a link. This includes changing a data type, e.g., from string to numerical or double to integer, thus removing the restriction on port connections;
- The availability of links with a changeable timed delay; and
- The possibility of extracting information non-intrusively from a computing agent.

3.1.2 RCS

The RCS system [6] can combine various architectural designs into a single unified architecture. This system describes a design methodology which has been successfully used in the implementation of several systems from autonomous vehicles to task factory control. RCS defines a control paradigm whose structure for RCS-4 can be seen in Figure 3.1. In the figure:

- WM is the current world model: the system’s internal representation of the external world;
- VJ is the value judgment system, which provides cost, benefit, and risk analysis on actions in the current situation and places values on objects, events, etc.;
- BG is the behavior generation module, which chooses a behavior for the agent; and
- SP is the sensory processing module.

Figure 3.1. The components of the RCS control paradigm: WM is the current world model, VJ is the value judgment system, BG is the behavior generation module, and SP is the sensory processing module.

Indices in the image indicate levels of complexity: RCS decomposes a task into sub-tasks, with more complex tasks receiving higher indices in the structure. Since more complex tasks are less time critical and require more computation time to complete, lower layer (smaller indices) will update faster than the higher layers.

RCS imposes no architectural constraints on the architecture developer. However, while RCS allows for the use of such diverse methodologies as SOAR [61, 90] and neural nets at different locations in an architecture, it does not provide an explicit way of integrating the different methodologies. APOC also does not impose any architectural constraints on the architecture developer. Additionally, it provides a definition of the
components of an architecture, i.e., computational and communication elements, which presents the architecture developer with a context which facilitates the integration of various methodologies into a single architecture. Thus, APOC gives the architecture developer a structured context in which RCS-based system development can be carried out.

3.1.3 GRL

GRL makes another step towards an architecture framework by constructing a new programming language which incorporates some of the desired characteristics of an architecture framework. Programs written in GRL are ultimately networks of signals which are computed in parallel and continuously updated. The supported signals can be:

- Constants;
- Signal sources for sensor and effector interfaces;
- Applications of a primitive procedure to a set of signals; and
- Applications of a finite-state transducer to a set of signals.

GRL is a functional programming language based on SCHEME; programmers have to supply scheme code for sensor and effector interfaces as well as for transducer value computation. This language is targeted towards behavior-based architectures and therefore uses behaviors, defined as data abstractions, as building blocks for architectures. Arbitration schemes in GRL are implemented as higher-order functions. The example below, described by Horswill [50], shows an implementation of a weighted average between the motor commands of two behaviors: move-toward-goal and avoid-obstacle.

```
(define-signal (weighted-average . behaviors)
  (/ (apply weighted-sum behaviors)
    (apply + (activation-level behaviors))))

(define-signal motor-output
  (weighted-average move-toward-goal avoid-obstacles))
```
By treating arbitration mechanisms as “higher-level functions,” GRL allows for the combination of different arbitration mechanisms in one architecture and moves closer to a complete integration of architecture with arbitration mechanisms. APOC completes the process by encapsulating arbitration mechanisms in specialized components in the architecture. Another improvement found in APOC consists in the provision of explicit, standardized communication facilities among components.

3.1.4 CAMPOUT

The CAMPOUT architecture is described by Pirjanian et. al. [86] as a behavior-based architecture related to ALLIANCE [79], DAMN [87], BISMARC [52], and MOBC [85]. CAMPOUT is a hybrid architecture, combining reactive and deliberative components. A typical system is seen in Figure 3.2, adapted from Pirjanian [86]:

17
The main significance of CAMPOUT to APOC lies mainly in its treatment of behavior-coordination mechanisms (BCMs). BCMs are intended to be implemented as operators, inserted into the architecture and used to compose behaviors. APOC also considers BCMs integral parts of the architecture and provides mechanisms for their customization.

The outputs of CAMPOUT behaviors are multi-valued preferences. APOC builds on CAMPOUT capabilities by supporting any output formats, including multi-valued preferences.

Other evidence for the movement towards an architecture framework is provided by practical systems incorporating one or more characteristics which would be useful in such
a framework. Examples from applications include:

- The RETSINA infrastructure [101] provides a generic communication protocol among agents, though lacking the degree of connection customizability present in APOC;
- JADE [22], as well as many other packages, provides similar functionality;
- Mozart has the capability to track resource constraints and to account for them in its agents [105]; and
- ZEUS provides users with several choices of co-ordination protocols for a multi-agent environment [77].

It is evident that many systems exhibit characteristics desirable of an architecture framework. However, there is no framework which would provide a theoretical foundation for these systems as well as a means of comparison among different designs. In this context we propose APOC as an attempt towards the creation of a theoretical architecture framework for complex agents.

3.2 The APOC Architecture Framework: Theory

APOC is an acronym for “Activating-Processing-Observing-Components,” which summarizes the functionality on which the APOC agent architecture framework is built: heterogeneous computational units called “components” which can be connected via four link types to define an agent architecture. APOC components are based on the behavior nodes described by Scheutz [92]. The four link types defined in APOC are intended to cover important interaction types among components in an agent architecture: the “activation link” (A-link) allows components to send messages to and receive messages from other components; the “process control link” (P-link) enables components to influence the computation taking place in other components; the “observation link” (O-link) allows components to observe the state of other components; and, finally, the “component link”

---

1From an APOC perspective, these would be the same as arbitration mechanisms, since arbitration is part of the system, whether that system designates a single agent or a system of agents.
(C-link) allows a component to instantiate other components and connect to them via A-, P-, and O-links.

Components can vary with respect to their complexity and the level of abstraction at which they are defined. They could be as simple as a connectionist unit, e.g., a perceptron, as complex as a full-fledged condition-action rule interpreter, e.g., SOAR [61,90], or they could represent entire agents in a multi-agent environment. For a better understanding of APOC and ADE components and links, a more detailed description of the APOC building blocks is provided below.

3.2.1 APOC Components

An APOC component is a very general, autonomous control unit having an internal structure which can be tailored to various roles within architectures. Each component can perform three generic operations: updating its own state, influencing another component, and controlling an associated process.

An associated process is itself very general in nature, being able to stand for either a computational process, e.g., visual processing, or a physical process, e.g., motor control. An associated process can be in one of five states at any given time: READY, RUNNING, INTERRUPTED, FINISHED, or NOPROC. The READY state means that the process has not begun execution. The RUNNING state indicates that the process is executing. The INTERRUPTED state means that the process has had its execution paused. This state was provided to allow processes to resume execution without a complete restart wherever feasible. For example, if a robot arm motion is interrupted, but the arm is not moved we can resume the motion from the place of interruption. The FINISHED state indicates that a process has completed its execution. Finally, if a component does not have an associated process, the associated process state is defined as NOPROC.

An APOC component with an associated process can control that process via three
primitive actions:

1. **START** will begin a process in the **READY** state. The effect of the **START** action on an interrupted or a finished process will depend on the nature of that process.

2. **INTERRUPT** will interrupt a process in the **RUNNING** state.

3. **RESUME** will resume the execution of a process in the **INTERRUPTED** state.

Each generic APOC component $C$ has a well-defined internal structure, consisting of the following:

1. An **activation level**, which is an indicator of the internal state of the component. The activation level can be affected by inputs, previous states, and operations performed either by the component, e.g., completion of an effector action, or on the component, e.g., interruption of associated process.

2. An **update function**, which maps inputs, previous component states, and the state of the associated process to outputs, a new component state, and perhaps a new state of the associated process. The update function $F$ of a component is defined as a mapping

$$F : ST \times PST \times I\mathcal{N}^k \rightarrow ST \times OUT^l \times OP^{3+k+l}$$

where $ST$ is the set of internal states of generic components, $PST$ is the set of possible states of the associated process as mentioned above, $I\mathcal{N}$ is the set of input port states, $OUT$ is the set of output port states, and $OP$ is the set of operations a component can perform with $k$ and $l$ being the number of input and output ports, respectively. This set of operations can be further subdivided into:

- Operations the component can perform on its associated process and, perhaps, on the associated processes of other components: $POP = \{\text{START, INTERRUPT, RESUME, NOOP}\}$.
- Operations the component can perform on itself and/or other components: $COP = \{\text{INSTANTIATE, TERMINATE, NOOP}\}$, and
- Operations the component can perform to change its priority: $SOP = \{\text{INCR, DECR, NOOP}\}$.

Overall, $OP = POP \cup COP \cup SOP$. It should be noted that in all sets, NOOP is used if no operation is performed. Thus specified, $OP$ defines the set of possible architectures, topologies of components, operations, and input and output states.

3. A **priority level**. The priority level of component $C$ is used to determine whether the component can influence the associate process of another component. Conversely, it also determines whether another component can influence the associated process of the component.
4. An instantiation number which is used to determine whether components of C’s type can be instantiated by other component in the system. Conversely, instantiation numbers of other components are used to determine whether or not C can instantiate new components in the system.

5. Input and output ports, denoted by in and out respectively. An input port can be connected to an output port of the same or another component via an APOC link. A set of connected components creates a network of components, or an architecture schema. Each port can be used to instantiate a component if a component has not been instantiated already, terminate a connected component, or manipulate the associated process of the connected component. in and out are k- and l-tuples, respectively, of triples \( (m, p, n) \) which reflect the states of k input and l output ports of the component, where \( m \in I \cap O \) is the message received from or sent to port \( p \) of component \( n \); a triple of the form \( (m, p, \emptyset) \) indicates that port \( p \) is not connected to any other component.

6. An associated process, as discussed above.

7. A state variable, \( op \). \( op \) is a \((3+k+l)\)-tuple, with the first three elements representing other parts of the component: the process, priority, and maximal number of instances of that component type which simultaneously exist. The following \( k \) elements of the tuple are its input ports, while the last \( l \) are its output ports.

The basic structure of an APOC component is shown in Figure 3.2.1

![Figure 3.3. The structure of an APOC component](image)

APOC limits the operations which can be performed on associated processes of components in an architecture by allowing such an operation only if the priority of the component performing the operation is higher than the priority of the component on which the
operation is being performed. The latter cannot override the operation being performed on it unless it modifies its current priority to a level higher than that of the component attempting to operate on it. However, priority changes can only be achieved in a step-wise manner as defined by the order in the set of priority levels up to the maximum level. Thus, sudden priority boosts cannot be achieved. Priority reduction occurs in a similar manner, down to the minimum level. If a component requires the ability to always control the process of another component, its default priority can be set at a high value. If, on the other hand, such control should only be available to the component under certain circumstances, our approach to priority increase and reduction ensures a level of stability within the system, as priorities will usually not change significantly due to short-term changes in the environment.

In the case where multiple components attempt to gain control of the associated process of a component, priority arbitration is performed. Priority arbitration in APOC is done by simply choosing the component with the highest priority among those attempting to gain control of the associated process. If there is more than one component at the highest priority level, the component whose process is affected attempts a resolution of the associated process commands it receives. If all commands sent from components of highest priority are identical, then that command is executed. Otherwise, no operation will be performed on the associated process.

An important feature of the APOC framework is the creation of components by other components. Components can only instantiate other components up to those components’ instantiation limit. For instance, every time the \textit{instantiate} operation is performed a check is performed to see if the instantiation limit for that component has been reached, at which point no more components can be instantiated. If the limit has not been reached, a new component is created and the instantiation number for its type is increased. Similarly, every time the \textit{terminate} operation is performed, the current instantiation number
is decreased. Components can only terminate other components if they have instantiated them.

The instantiation and termination of components by other components requires that APOC make a clear “type-token distinction” between component types and their instantiated components (tokens), which leads to a “type-token distinction” at the level of the architecture. In order to be able to instantiate another component, an APOC component needs to have a description of that type of component. This description is implicitly stored in the update function and consists of (1) the type of component which can be instantiated and (2) the port on which the instantiation can take place. As mentioned above, an instantiation can only be performed on a port if a component is not already connected to that port.

Overall, the APOC instantiation process is similar to a “bootstrapping” process of virtual machines in standard computers on power-up: one initial component is instantiated and has an initial state specified. That component then instantiates all other components for the original state of the architecture.

The APOC “type-token” distinction distinguishes between “component types,” which define components, and “component tokens,” which are instantiated in the running virtual machine. This forms the basis of a resource management mechanism at the architecture level: every architecture fixes the maximum number of instances for each component type that can be present simultaneously, reflecting the fact that agents only have finitely many resources. Thus, for “component types,” the inst tuple consists of the number of tokens of that type present in the architecture when it is first instantiated, \( \text{inst}_{\text{default}} \), and the maximum number of tokens which can simultaneously be existing in the running virtual machine, \( \text{inst}_{\text{max}} \). Architectures can therefore be defined at varying levels of detail, contingent on such factors as the complexity of the components and the degree to which the running virtual machine is allowed to modify itself.
Each generic component instantiated in a virtual machine is a self-sufficient entity. Its behavior is governed by the specification of initial state, associated process, and update function. The state of a generic components is fully determined by the definition of three additional sets: $ACT$, the set of activation levels, $PRO$, the set of processes, containing NOPROC, and $PRI$, the set of priority levels. Then the state of a generic APOC component can be defined as

$$\langle act, pri, pro, inst, F, in, out, op \rangle$$

where $act \in ACT$ is the activation level, $pri \in PRI \times PRI$ is a pair containing the current and the maximum priority level, $pro \in PRO \times PST \times POP$ is a triple containing the process state and the process associated with component as well as the operation performed on that process, $inst \in \mathbb{N} \times \mathbb{N}$ is a pair containing the current instantiation number and the maximum number of instances of a component of that type, $F \in \mathcal{UF}$ is the update function.

APOC components are connected to other components through one of the four previously mentioned APOC links. Each component has associated with it a pair $\langle id, max \rangle$ where $id$ is the instantiation number of the component, and $max$ is the maximum number of components of that type that can be instantiated in an APOC architecture. Since the restrictions on the set of possible update functions of APOC components are determined by the functionality of these four links, it is sufficient to discuss the four link types.

### 3.2.2 APOC Links

The A-link

Activation links are the most general means by which components can exchange information. The state of an A-link is given by the tuple

$$\langle id, max \rangle$$
where $S$ is the component providing the data, $R$ is the component receiving the output, $\text{act}$ the data transmitted through the link, $F$ the operation performed on that data, and $t$ is the time it takes for data to traverse the link.

The purpose of an A-link is to connect two APOC components and serve as a transducer. An A-link can be used in a variety of different ways. In the simplest case, $F$ can be defined as the identity function. In this case the link functions as a mere connection between an input and an output port of two APOC components, i.e., the input to the link is identical to its output. Furthermore, an A-link can be used to transform the input, e.g., in case of numerical values it could “scale” the input by a particular factor, analogous to the “weights” on connections in neural networks. It is also possible, to implement a “timed link,” i.e., a delay, with which the value at the output port of $S$ arrives at the input port of $R$.

A-links provide opportunities for learning at the architecture level. First, by modifying the operator on a link, neural-network-based learning, e.g., Hebbian and back propagation learning can take place within an APOC-based architecture. Second, by allowing modifications to the delay factor on an A-link, associations can be learned and priming can be obtained for information retrieval as in the ACT-R cognitive architecture [7], e.g., by decreasing transfer time between two components which are active at the same time. ART network-based learning [32] can also be modelled within in APOC, as shown in Section 5.4.1. Q-learning can be modelled in a separate APOC component, connected through O-links to all sensors, and through A-links to all behaviors. Finally, architecture specific learning can be modelled through instantiation and deletion of architecture components, as shown in Chapter 3.
The P-link

Priority links are intended to explicate the capacity of generic components to control other components’ associated processes. They are the only means by which an APOC component can directly control processes of another component; the second component automatically loses control of its process if the first has a higher priority. Since no link has a process associated with it and components can only be connected to other components via links, the only other method through which an APOC components could control the process of another component is indirect, by having the second component “agree” that upon receiving a predetermined value through an A-link, it will change its process state. The state of a P-link is given by the tuple

\[ \langle S, R, pri, op, t \rangle \]

where \( S \) is the component attempting to take control of the process associated with \( R \), \( R \) is a component whose process may be influenced by \( S \), \( pri \) is the priority of \( S \), \( op \) the operation that \( S \) attempts to effect on the process associated with \( R \), and \( t \) is the time it takes for data to traverse the link.

A P-link effectively passes the process control request of an APOC component on to the component it is connected to through the P-link. Priorities can be used to implement many types of control mechanisms, in particular, hierarchical preemptive process control. In embodied agents, such as robots, they could be used to implement emergency behaviors: the component with the associated emergency process would have the highest priority in the network and be connected to all the other components controlling the agent’s behavior, which it could suppress in case of emergency (thus implementing a “global alarm mechanism” as described by Sloman [97]).
The O-link

Observer links are intended to allow components to observer other components’ inner states without affecting them. The state of an O-link is given by the tuple

\[ \langle S, R, D, t \rangle \]

where \( S \) is the component observed by \( R \), \( R \) is the observed component, \( D \) is the information passed from \( S \) to \( R \), and \( t \) is the time it takes for data to traverse the link.

The C-link

Component links are used to instantiate and remove instances of APOC components at run-time. They are the only type of component that can instantiate or terminate an APOC component. They are also used to instantiate the other link types between APOC components and are themselves only instantiated by APOC components. The state of a C-link is given by the tuple

\[ \langle S, R, D, L, t \rangle \]

where \( S \) is the component attempting an instantiation operation, \( R \) is the component instantiated by \( S \), \( D \) is information about the links which can be instantiated through this C-link, \( L \) is the set of links already instantiated through the C-link, and \( t \) is the time it takes for data to traverse the link.

We have now described the constituents of the APOC framework. In the following chapter we shall demonstrate the use of the framework to bring together in one formalism several architectures and even applying APOC to introduce new concepts in agent architecture design.
CHAPTER 4

APOC AS AN ANALYSIS AND COMPARISON TOOL

This chapter begins the description of uses of the APOC framework in the context of agent architectures. We will first explore APOC as an analysis tool. In the subsequent sections we then show how different architectural mechanisms can be expressed in APOC. This establishes a basis for architecture analysis, as architectures can be compared with respect to their architectural and communication requirements.

4.1 APOC: An Analysis Tool

APOC is a useful tool for the analysis and comparison of agent architectures. Its expressiveness allows it to express many agent architectures in a unified way. Cognitive architectures such as SOAR [61], ACT-R [7], and behavior-based architectures such as subsumption, motor schemas, and situated automata, [12, 27] can be expressed in APOC and can be used together in different parts of one agent architecture. Furthermore, APOC has a notion of “cost” defined for components and links that allows for the systematic assessment of “structural cost” and “processing cost” of the whole instantiated architecture at run-time. Consequently, it is possible to analyze properties of architectures and their subarchitectures in terms of their cost. For example, the action selection mechanism in Maes’ ANA architecture requires global control despite some claims that it uses only local mechanisms [67]. Analysis results, can be used to compare the tradeoffs of various architectures with respect to some particular function, e.g., two different architectures imple-
menting an “target-finding task” can be compared with respect to their performance-cost ratio.

APOC can also be used to analyze agent architectures at different levels of abstraction. There are two parts to this type of analysis. The first part focuses on the level of abstraction at which components should be implemented, based on the complexity and modularity of their functionality. A question answered at this level of analysis is, for example, whether the implementation of the production system in SOAR should be as a single APOC component, or each production should be a component. At this level of analysis the architecture designer decides which components use associated processes and how complex these processes are going to be.

If an architecture is to be described at a low level of abstraction, e.g., at its implementation level, then APOC components assume the role of the basic components of the implementing (virtual) machine and do not have associated processes, as in implementations of Boolean networks.

At higher levels, however, APOC components may not be sufficient to specify all details, e.g., of the involved processes at lower levels, or it may not be desirable to give a complete specification of all details. In that case, the associated process of a component can take over the details implicitly, while the component itself is viewed as and becomes part of a higher-level description. For example, a “behavior” in a behavior-based architecture can be expressed in terms of an APOC component, whose associated process operates on the agent’s effectors, while the controlling component reflects the behavior’s state and its higher-level activities, such as participation in action selection.

Type-token specifications and the capability for architecture modification over time form the second dimension of architecture analysis with APOC. Consider the basic example of run-time component instantiation, presented in Figure 4.1.
Figure 4.1. Type diagram, initial instantiation, state after first request, and final state of sample APOC architecture

In Figure 4.1, T1, T2, and T3 represent three component types, with types T1 and T2 utilizing the action performed by type T3. The numbers in parentheses in the type diagram indicate $inst_{default}$ and $inst_{max}$ for each of the three types. Components I1 and I2 are instances of types T1 and T2 respectively, while components I3 and I4 are instances of type T3. When the architecture is first instantiated, components I1, I2, and I3 are instantiated, since all three types have $inst_{default}$ set to 1. When the first explicit request for execution comes from one of the utilizing components, in this case, I2, that component is connected to the existing instance I3. The next request results in the instantiation of I4, with the requesting component, I1, using I4 to perform its operation.

Extending the previous example to neural networks, a layered neural network can be specified by connecting APOC components via activation links in two ways: either by having the whole layered network as part of the architecture specification, or by defining components that will, in turn, construct the layers and then instantiate them. In the latter case, the components that construct the layers at run-time can be viewed as “representation” of these layers at the type level by instantiating all components that are supposed to be part of the layers together with their connections.
The left side of Figure 4.2 depicts the type description of the neural network, i.e., a whole class of potential neural network structures: an instance of type T1 can create 100 instances of type T2. Similarly, instances of type T2 can create 100 instances of type T3. The right side of the figure then shows one possible run-time structure which can be obtained from the type description depending on the choice of update functions.

It is possible to specify C-link behaviors which produce permanent structures. APOC components have the capability of instantiating other APOC components through the C-link, as well as making all the necessary connections. In most circumstances, the instantiating components also use the C-link to delete the instantiated structure. However, the instantiating C-link can itself be deleted. This allows the newly instantiated components to function independently, e.g., performing a specialized function such as recognizing a specific stimulus. The scenario in Figure 4.2 as presented is both dynamic and reversible, i.e. the structures are created and can be destroyed at run-time.

Thus far we have shown some of the tools APOC provides for the analysis of architectures. In the following sections we explore the types of architectures which can be modelled in APOC and provide translations to the APOC formalism for several behavior-based architectures, including subsumption [27], schema-based [12], and DAMN [88], cognitive architectures, including SOAR [61] and ICARUS [63], as well as a middle-
layer system, which can act as a connection between the two types of systems, Contention Scheduling [39].

4.2 Expressing other architectures in APOC

There are many ways in which behavior-based architectures can be expressed in the APOC framework. A sequential architecture with a single computational process, for example, could be viewed as a single APOC component, where the architecture’s functionality is entirely encoded in the update function $F$. In this section we show one possible translation to the APOC framework for several architectures.

4.2.1 Behavior-Based Architectures 1: Subsumption

The subsumption architecture [27] is a layered system, in which individual layers work on individual goals concurrently and asynchronously. Layers consist of nodes, each node being the representation of a behavior. Each behavior is implemented as an augmented finite state machine (AFSM).

Subsumption architectures can be translated into the APOC framework in straightforward manner by defining their components, the augmented finite state machines (AFSM), as follows:

1. The state table is directly incorporated into the update function of an APOC node;
2. Environmental/data inputs map onto A-links;
3. Inhibitor connections use A-links and simple, specialized APOC nodes to decide whether to pass on information or whether to block it;
4. Reset and suppressor connections can be implemented via P-links, coupled together with assigning nodes a priority proportional to the layer in which they are found. Thus, nodes in higher levels can control the execution of nodes lower in the layer hierarchy; and
5. A-links are used for message passing.
4.2.2 Behavior-Based Architectures 2: Action Network

Based on Minsky’s “Society of Mind” [71], the Agent Network Architecture (ANA) is viewed as a set of competence modules [67]. Competence modules are connected through three types of links: successor, predecessor, and conflictor. Two general conditions are imposed on the architecture: (1) all nodes in the network have the same activation threshold, and (2) if no active nodes are found in the network, the activation threshold is lowered by 10%.

The functionality of each node in the ANA can be identically replicated within a corresponding APOC node. A unique activation threshold has to be chosen for the entire network. The activation threshold modification function is then set to bring about a 10% decrease in threshold every time no active nodes are found.

A special node is implemented with observer and activation links to all other nodes. The function of this node is threefold: (1) to compute the average activation after each time-step, (2) to send this activation back to each node, which can then perform its own normalization, and (3) to observe active nodes within the network and decide whether a lowering of the activation threshold is necessary.

Incoming activation links are then structured to form seven categories of ANA links: predecessor links (excitatory), successor links (excitatory), conflictor links (inhibitory), sensor links or activation by state (excitatory), goal links (excitatory), protected goal links (inhibitory), and the average activation link.

In addition to the structural mapping, several global parameters of the ANA architecture need to be determined:

1. \( \pi \), the mean activation value after each timestep;
2. \( \Theta \), the initial value of the global threshold;
3. \( \Phi \), the constant determining the weighting of environmental sensor inputs and successor links;
4. \( \gamma \), the constant determining the weighting of goal inputs and predecessor links; and
5. \( \delta \), the constant determining the weighting of protected goal inputs and confliector links.

4.2.3 Behavior-Based Architectures 3: Motor Schema-based Systems

In schema-based approaches [13, 73], motor schemas operate as “concurrent, asynchronous processes each of which instantiates a behavioral ‘intention’ ” [82]. This principle translates directly into APOC:

1. Each perceptual and motor schema maps onto an APOC component, with the computation that defines the schema in the update function of the component or the associated process, based on the complexity of the schema;
2. Motor and perceptual schemas are connected by A-links;
3. Sensors trigger, via C-links, the instantiation of the respective perceptual-motor schema combination;
4. It is necessary to include a “summation node” component performing the fusion part of all schemas for the effector output. This component is always instantiated and also connected to the output of motor schema components via A-links. Note that in APOC the associated process of this component could be directly in charge of controlling the effectors.

The system in Figure 4.3 presents an implementation of a schema based system. The system represented is the ‘ACQUIRE’ system presented by Arkin [13]. Since the mapping from that system to the run-time APOC environment is direct, only the APOC representation is given here. In the figure, \( d_1 \) through \( d_4 \) represent, respectively, detect-attractor, detect-obstacles, detect-robot, and generate-direction, while \( a_1 \) through \( a_4 \) are Move-to-goal, Avoid-static-obstacle, Avoid-static-obstacle, and Noise.
4.2.4 Behavior-Based Architectures 4: DAMN

DAMN [88] is a voting scheme in which a central arbiter tallies votes from all behaviors to select the best behavior. The translation into APOC involves the following:

1. Each behavior maps onto an APOC component. The computation that defines the behavior is implemented in the update function of the component or its associated process, based on the complexity of the behavior.
2. Each motor controller is mapped to an APOC component, with the motor control being part of the update function.
3. The arbiter is a specialized APOC component with arbitration code most likely in the associated process.
4. Votes are passed to the arbiter through incoming A-links.
5. Commands are passed to motor controllers through outgoing A-links.

4.2.5 Behavior-Based Architectures 5: DAC

The DAC system [83] uses a neural-network based approach to implement controllers for a robotic agent. In DAC, each behavior is implemented through a neural network. When any node in a layer has an activation value of 1, it automatically triggers a motor response. In the original system presented, the two behaviors are connected through an
inhibitory element which gives preference to the avoidance behavior over the approach one.

Due to the use of neural networks for control, the DAC architecture exhibits implicit behavior selection. The adaptive component of DAC is also implicit, as DAC architectures adapt by modifying neural network weights as a result of environmental interaction. The changes are made through a modified Hebbian learning algorithm.

DAC can be modelled in APOC as follows:

1. Each sensor maps onto an APOC component.
2. The Target Detector and Collision Detector each map onto an APOC component.
3. Each neuron maps onto an APOC component.
4. All connections among nodes are performed through A-links, where links within a neural network implement, in their operator part, the learning mechanism.

4.2.6 Behavior-Based Architectures 6: GRL

GRL is a language for behavior-based which makes the generalization of treating arbitration mechanisms as higher-level procedures. The GRL translation into APOC can be seen below:

- Each procedure maps onto an APOC component.
- A-links are used for data transfer from the environment to behaviors and among behaviors.
- O-links are used for data transfer from behaviors to arbitration components.
- Arbitration schemes are defined as APOC components. These components receive inputs from the arbitrated components, process them according to internal rules and produce overall outputs for the system. The internal rules can implement any arbitration mechanism: competitive, cooperative, or a combination.
- Sequencing can be obtained through the use of “flag” variables within components. Other components, which depend on prior computation, use O-links to observe the flag variables and only start their computation once the flag variable has changed to a predetermined value.

The GRL definition of a behavior is as follows:
(define-group-type behavior
  (behavior act-level motor-vector)
  (activation-level act-level)
  (motor-vector motor-vector))

The weighted sum operator which would be required for a motor-schema implement-
tation is:

(define-signal (weighted-sum . behavs)
  (apply + (weighted-motor-vector behavs)))

(define-signal (weighted-motor-vector beh)
  (* (activation-level beh)
      (motor-vector beh)))

In APOC, with each behavior and the arbitration algorithm being embedded in sep-
arate components, this systems creates the structure in Figure 4.4, where $b_1$ to $b_n$ are the behaviors used in the system.

![Diagram](Diagram.png)

Figure 4.4. An example of an APOC translation of a GRL structure

4.2.7 Behavior-Based Architectures 7: L-Alliance

L-Alliance is a relative of the Alliance system [79, 81]. In L-Alliance [80], adapta-
tion is provided through the variation of several parameters during two phases: a learning phase and an adaptive phase. In the learning phase, agents learn about their capabili-
ties and those of their teammates without concern for task completion. In the adaptive
phase, agents are still able to modify their behavior-selection strategy, but the changes are directed strictly towards goal achievement.

One global parameter, the activation threshold, needs to be fixed for all nodes. All other parameters are local to each node and can, therefore, be computed as part of the update section of a node in the robot architecture:

1. **Sensory feedback**: a binary parameter indicating whether the behavior associated with the node is applicable to the current sensory configuration.

2. **Inter-robot communication**: a binary parameter indicating whether another agent has sent a message regarding the behavior associated with this node.

3. **Suppression from active behavior sets**: a binary parameter indicating whether another behavior is currently active in the agent.

4. **Learned robot influence**: a binary parameter whose value is based on a threshold function. In the active learning phase it indicates whether another agent is attempting the behavior associated with the current node. In the adaptive learning phase, it indicates that the agent’s “boredom level” is above a threshold or that the agent believes that it can achieve a task in less time than the agent currently attempting that task.

5. **Robot impatience**: a real-valued parameter indicating the time the agent is willing to allow another agent’s messages to influence its own motivation.

6. **Robot acquiescence**: two real-valued parameters indicating the time before the agent yields the task to another and the time before the agent gives up on the task when left on its own.

All the above parameters are then combined according to the update rule of the node, following the definitions of the L-Alliance architecture, to compute a motivational factor. Behaviors associated with nodes whose motivations exceed the threshold will then compete for execution based on a predefined behavior selection policy, such as “shortest job first” or “pick behavior at random.”

4.2.8 Behavior-Based Architectures 8: Behavior Column Architecture

The Behavior Column Architecture (BeCA) is an architecture that consists of a network of blackboard-based cognitive and motivational nodes [47]. Each node is made up of five basic components: internal (or “elemental”) behaviors which are knowledge sources,
a blackboard, activity state registers of the internal behaviors, the interface/communication mechanisms, and the competition/control mechanism.

For a translation into APOC, either a whole BeCA node can be mapped onto an APOC component, or each of the basic components of the BeCA architecture can be translated into an APOC component. In that case, internal behaviors are condition-action rules which map in a straightforward manner to update functions in APOC components. Internal behaviors connect to the blackboard via O-links to observe the current state of the problem solving process, and A-links to place new solution elements to the blackboard. The activity state registers are connected to internal behaviors through O-links and observe the state of these behaviors. Environmental and system information is fed to a BeCA node through A-link connections to the communication node, which, in turn, is connected through A-links to the blackboard in order to store environmental information and to other nodes in the system. The competition/control mechanism is connected to each internal behavior through an O-link which observes the activation of the node, and a P-link which can activate or inhibit the behavior.

The adaptive part of its behavior-selection mechanism comes from parameter modifications in internal behaviors. Two quantities can be modified: the strength of connections between internal behaviors and conditions, i.e., the efficacy with which a behavior can satisfy internal perceptions, external perceptions, and drives, and a combination factor between internal and external signals. Both types of modification are performed as part of the update section of the APOC component representing an internal behavior.

4.2.9 Behavior-Based Architectures 9: Hybrid Coordination

Hybrid coordination [33–35] is another approach towards bridging the gap between competitive and cooperative systems. In this approach, behaviors are grouped in sets of two, with one behavior designated as dominant. The two behaviors send their outputs,
consisting of a directional vector and an activation value, to a hybrid node, which combines them in a manner preferential to the dominant behavior. A similar process is applied recursively to hybrid nodes until an overall unique vector is generated as the overall behavior of the system.

The hybrid coordination architecture maps cleanly onto the APOC framework. Each behavior node, as well as each hybrid node, can be mapped onto an APOC node. Communication, both of desired actions and activation values, is achieved through A-links. The adaptive component of this architecture consists in the degree of cooperation among behaviors, which is controlled by the activation level of the dominant behavior.

An important contribution of this approach is that the hybridization process is applied ‘on top of’ an existing architecture. This is similar to our approach for arbitration where arbitration mechanisms can be embedded in architectures. However, whereas hybrid coordination requires communication to be consist of an activation value and a vector, thus making it difficult to use with components which use different information for communication such as subsumption components, in APOC, components can be defined to coordinate components designed in any methodology.

An APOC translation of each of the architectures discussed thus far can be seen in Figures 4.5 and 4.6.
Figure 4.5. The architectures modelled in APOC as discussed in Sections 4.2.1 through 4.2.9 - Part I
Figure 4.6. The architectures modelled in APOC as discussed in Sections 4.2.1 through 4.2.9 - Part II
4.2.10 Middle Layer: Contention Scheduling

The contention scheduling scheme, as a mechanism for action selection, forms the middle layer in a three-layer architecture, in which the bottom layer is responsible for carrying out “actions,” and the top layer consists of a supervisory system that monitors the progress and possibly corrects the processes in the layers below. The middle layer is itself divided into three parts: a schema network, an object network and a resource network, each of which is hierarchically layered.

The Schema Network

The schema network consists of goal directed schemas and goals. Each schema is made up of a set of several partially ordered goals, all of which have to be satisfied before the overall goal of the schema is achieved. Each goal, in turn is composed of one or more schemas, any one of which may be used to achieve the goal. A numeric activation is associated with each schema. This activation varies over time as a result of several influences that are exerted on the schema. It is the activation of each schema that ultimately leads to the selection of an action. If the activation of a schema is greater than a given threshold, then the schema is allowed to pass activation down to its component schemas, i.e., provide top-down influence. Other types of influence in contention scheduling come from the environment, from the schema itself, lateral from other schemas, and from random noise, all of which contribute to the activation of a schema. The environmental influence acts as a set of triggering conditions: a schema is only allowed to be active if the current conditions allow its action to proceed. An additional requirement for schema activation is that its goal must not have been achieved prior to the time when it becomes eligible for activation.

The bottom of the hierarchy is composed of basic schemas, which correspond to simple actions, such as “pick-up-object.” In the case of the basic schemas, an activation that
is greater than the threshold leads to the execution of the associated action. Completion of this action leads to the satisfaction of the goal immediately superior to the schema in the schema/goal hierarchy. The goals for each higher-level schema are stored as part of a list in the respective schema and checked off as each goal is achieved.

In relation to the APOC framework, a general description of the schema network is obtained through the decisions below:

1. APOC nodes are divided into two categories: goals and schemas. Goals are boolean nodes that indicate achieved/unachieved status. Schemas are sets of actions that lead to the achievement of goals.

2. Within each basic node representing a schema, A-links from the environment form a special class of inputs. These links pass a non-zero activation to the schema before it can become active.

3. Each goal node has an activation threshold of zero. Since, in contention scheduling, goal nodes do not have activations, this allows a goal node to simply pass the activation received through A-links to its component schemas.

4. Only A- and O-links are used in the network, as they most closely parallel the links described in contention scheduling. By definition, the A-link performs the function of activation-passing described in contention scheduling, while O-links provide a convenient mechanism for signaling that a condition has been achieved. As a result of this implementation decision, the priority of each basic unit does not affect computation.

5. The structure of the architecture with respect to A-links is hierarchical, i.e., no activation loops can be present in the schema network.

6. The basic nodes representing schemas in the bottom layer of the A-link hierarchy implement basic actions.

7. Upon completion of its action, a schema node sends a signal to the corresponding goal node via an O-link, causing the goal to switch state from unachieved to achieved.

It is worth noting that, in contention scheduling, subschemas are not treated as sub-components. Instead, the relation among schemas is governed through lateral inhibitory links. The following scenario is therefore feasible in the contention scheduling framework:

A high level schema is active and passes activation down to its subschemas. However, at the motor-schema level, a schema unrelated to the high level schema is highly activated
by the environment and is therefore wins the competition at the motor-schema level and begins executing. This activation is a possible explanation for a number of errors exhibited by people in daily activities. Consider for example the case where a person opens up the refrigerator door in order to take out the orange juice. However, the first object seen when the refrigerator is opened is a milk carton. Contention scheduling can account for the person picking up the milk carton and pouring a glass of milk instead of the desired glass of orange juice. In order to support this type of behavior, C-links are not used in the APOC description of contention scheduling. The use of C-links leads to direct activation of sub-behaviors, and would thus eliminate a characteristic of the contention scheduling scheme.

The Object Network

Another subsystem of contention scheduling is the object network. This network parallels the schema network in many respects. Each object has an associated activation value used to determine environmental influence on schemas and in deciding which object to use to achieve a task when more than one applicable object is available. A different activation value is stored for each possible use of the object. In the object network, activations are affected by lateral influence, self-influence, influence from schemas, and random noise. The lateral, self-, and schema influences are summarized in two assumptions:

The influence of a schema’s activation on that of an object representation (for a particular function) is dependent on the extent to which the object representation is employed, serving that function, in the triggering conditions of the schema. (PA 10, [39] p. 312)

Object representations compete within functional domains. This competition is effected by a lateral influence on the activations of competing object representations, and a self influence on all object representations. (PA 11, [39] p. 312)

The APOC description of the object network is even more concise than that of the schema network. It can be expressed in two:
1. Basic nodes are nodes whose relevant information consists of a set of numeric values. Each numeric value denotes the activation of the object represented by the node with respect to a possible use of that object.

2. Only A- and O-links are used in the network. As a result, the priority of each basic unit does not affect computation.

The Resource Network

The resource network and the object network serve similar functions. The same way actions require objects in the environment to which to be applied, they also require effectors in order to be completed. A resource and a schema influence each other if the resource can be utilized by the schema. When an action is executed the most active appropriate resources are allocated to it. Basic level schemas specify restrictions on objects and resources to which they may be applied; objects and resources take the role of arguments which are filled in for each basic schema as it becomes active in accordance to the specified restrictions. The description of the schema network in APOC is analogous to that of the object network.

Basic Parameters

In contention scheduling the activation of the various components is governed by the following several parameters:

1. Rest level activation: the activation of a schema without input;

2. Persistence (decay function): the function that governs the return of schema activations to a rest level after the net input becomes zero;

3. Random noise: a random value added at every update to the activation to help break ties if nodes have the same activation level;

4. The balance parameters. self:lateral, internal:external, and competitive:non-competitive. These parameters specify the proportions of total activation from various sources; self-excitation vs. later inhibition, internal contribution vs. external contribution, competitive contribution vs. non-competitive contributions; and

5. Activation threshold: the number which, when exceeded by a schema activation, allows the schema to be eligible to execute its associated actions or action sequence
By definition, rest level activation, persistence, competitive:non-competitive ratio, and self:lateral ratio are the same in all three networks.

4.2.11 Cognitive Architectures 1: SOAR

The structure of the SOAR architecture, as described in [61], is depicted in Figure 4.7. Five main components are present in SOAR:

1. A Working Memory, which is a container with information about Objects, i.e., goals and states of the system, Preferences, which are structures indicating the acceptability and desirability of objects in a particular circumstance, and a Context Stack, which specifies the hierarchy of active goals, problem spaces, states and operators.

2. A Decision Procedure, which is a function that examines the context and preferences, determines which slot in the context stack requires an action (replacing an object in that context), modifying the context stack as required.

3. A Working Memory Manager, which is a controller that determines which elements of the working memory, contexts and objects, are irrelevant to the system and deletes them.

4. A Production Memory, which is a set of productions that can examine any part of working memory, add new objects and preferences to it, and add new information to existing objects.

5. A Chunking Mechanism, which is a learning mechanism for new productions.
Since descriptions of architectures in APOC can be done at various levels of detail, there are several possible translations of SOAR to APOC, with varying levels of detail hidden in the process associated with each APOC component. However, to make better use of the intrinsic power of the framework, we describe SOAR at a fairly detailed level. Thus, instead of choosing to describe the Working Memory as an APOC component, we describe Preferences, Objects, and Goals as the basic components of an APOC-based implementation of SOAR, imposing the structure of the SOAR architecture through APOC links. Similarly, the production memory is mapped at the level of each production as an APOC component.

Two types of links can be distinguished in Figure 4.7. Some links have associated operations, which denote the fact that through those links elements can be either created
(+ or deleted (-). The other links are simply data transfer links; these translate directly onto the APOC O-links. Thus, the following O-link connections occur in an APOC implementation of SOAR:

- The Chunking Mechanism is connected to all Preferences, Objects, and the Context Stack. The items observed are the working-memory elements created in the subgoal being processed.

- The Chunking Mechanism is connected to all Productions in order to trace the productions fired during the subgoal being processed. Thus, the item observed is a boolean variable indicating the status of a production.

- The Working Memory Manager is connected to all Productions. Observed items are preferences and objects produced, whose information can be gathered from the production such that a direct O-link to those objects can be created.

- The Working Memory Manager is connected to the Decision Procedure. Observed items are the contexts produced, whose information can be gathered such that a direct O-link to those objects can be created.

- The Working Memory Manager is connected to all Preferences, Objects, and the Context Stack. The contents of each context of the Context Stack are compared against the identifiers of elements of the Preferences and Objects sets. Thus, the items observed are the elements of contexts and the identifiers of objects.

- The Decision Procedure is connected to all Preferences and the Context Stack. The contents of each context of the Context Stack are observed and processed. Preferences are observed for content and checked for matches against the context currently being processed.

The creation/deletion functionality of SOAR maps directly onto the APOC C-link. The creation process may require additional information to be passed to the newly created node. For example, the conditions under which a new production fires need to be sent to the production when a generic production is created and objects need to be given identifiers. An A-link is then created through the C-link and used for information passing.

The deletion process requires that information be known about the situation state of working memory, e.g., determining if an object is used in any context on the context stack. This information is retrieved in APOC through the O-link mechanism. Thus, a C-link from the “Decision Procedure” or “Working Memory Manager” to a preference, object, or a goal creates an O-link upon the creation of a component representing a preference,
object or goal. The new O-link link is thereafter used to observe the new component as described above.

4.2.12 Cognitive Architectures 2: ACT-R

ACT-R is a cognitive architecture for whose description we chose the recent description of version 5.0 by Anderson, et.al. [7]. Its description includes the brain regions which map onto each functionality of the architecture. The architectural layout is shown in Figure 4.8.

![ACT-R Architectural Structure](image)

Figure 4.8. The ACT-R architectural structure

The main components of ACT-R are described below:

- A **Perceptual – Motor System** which interacts with the environment, receiving visual input and sending commands to the effectors. Perceptual information elements (chunks) are also created in this system.

- An **Intentional Module**, also known as a **Goal Module**, which holds representations of intentions and keeps track of them, so the behaviors serve the goal. Abstract and compound chunks are created here.
• A Declarative (Memory) Module which holds and retrieves records of chunks.

• A Procedural Module which performs two main functions: partial matching on the conditions to determine the rules which are eligible for execution, selection of one rule among those eligible for execution to fire in the current cycle. ¹

In the simplest translation, each of the above elements can map to one APOC component. Chunks are also created as APOC components, which, in the ADE environment leads to the creation of a distributed memory structure among the participating computers on which the ACT-R architecture is distributed. A distributed matching system can then be implemented within an ACT-R architecture by creating a partial matcher on each machine. On each update cycle, the partial matcher reports which objects on its host match rule antecedents. The partial matches on each host are reported to the partial matcher from the host on which the rule resides, which then completes the matching process.

Communication among various elements of the architecture is made through a buffer associated with each element. Each buffer contains one piece of data (a chunk) which is accessible to the system at large. In APOC the buffer is simply a field in each component and its value is communicated to other architectural elements through the use of O-links.

The chunk retrieval process is activation based, with the activation of a chunk given by

\[ A_i = B_i + \sum_j W_j S_{ji}, \]

where \( B_i \) is the base level activation of the chunk, \( W_j \) is the attentional weighting of the elements that are part of the current goal, and \( S_{ji} \) are the strengths of association from the elements \( j \) to chunk \( i \). It should be noted that activation values map directly to the act element of APOC components. The base level is given by

\[ B_i = \ln(\sum_{j=1}^{n} t_j - d), \]

¹Like SOAR, ACT-R uses productions in the form of condition-action rules to encode all processes and skills necessary to achieve goals. However, in ACT-R, a single rule fires on each update cycle.
where \( n \) is the number of times element \( i \) in memory has been accessed (practiced), \( t_j \) is the time since the \( j^{th} \) practice of item \( i \) and \( d \) is a parameter estimated at 0.5. Retrieval time for chunks is then given by

\[
\text{Recognition time} = I + Fe^{-A_i},
\]

with \( I \) and \( F \) being experimentally determined time constants. The retrieval time of a chunk can be adjusted in APOC by varying the delay on the link going to the component representing that chunk.

4.2.13 Cognitive Architectures 3: ICARUS

ICARUS is a cognitive architecture with the capability of learning hierarchical skills. The ICARUS representation of skills is related to both production rules and STRIPS operators. ICARUS divides memory space into conceptual memory and skill memory. Conceptual memory is the residence of states, such as a description of a desk, and relations, such as a description of “on top of.” Skill memory contains the system’s knowledge about actions, such as “put object A on object B.” ICARUS also divides memory into long-term and short-term memories. Each element in long term memory is a symbolic description with an associated numeric function which computes the value of that description in terms of the current sensory value reading. Each element in short term memory is an instance of a long term memory element.

Long-term Conceptual Memory

Long-term conceptual memory contains definitions of concepts, such as car, and of relations, such as “in left lane.” To translate ICARUS long term memory into APOC we use the following rules:

- Each concept is represented as an APOC component. The characteristics of each object are embedded within its equivalent component. For example, a component
which represent a numeric concept, such as speed or distance, embeds the arithmetic function which computes the quantity associated with that concept within its update function $F$.

- Higher-level concepts, which have other concepts as positive or negative preconditions check for the achievement of those preconditions in $F$.

- Higher-level concepts whose preconditions need to be checked against specific objects connect via A-links to the lower-level concepts representing those preconditions. The links are used to send across the parameters to which the tests of lower level concepts are applied.

- Higher level concepts connect via O-links to lower level concepts to ascertain whether their positives and negatives are satisfied.

- Higher level concepts can create “instances” of their knowledge: the Lane-To-Right component can instantiate a Lane-To-Right-Instance, identifying a particular line found to the right. Therefore, concepts are connected to each of their instances through a C-link.

Long-term Skill Memory

Long-term skill memory contains knowledge about ways to act and achieve goals, such as how to overtake a car moving slowly ahead. To map long-term skill memory to APOC we use the following rules:

- Each skill is represented as an APOC component.
Each skill connects through O-links to sub-skills/rules in order to verify their completion.

Each skill component connects through O-links to concepts in order to verify that the pre-requirements (start:) and the continuing requirements (required:) are met (if necessary).

Distinctions between ordered: and unordered: are implemented in the update function, $F$, or in the associated process.

The evaluation function for a skill decomposition is defined in the update function, $F$.

Short-term Conceptual Memory

Each instance of a long term concept which can be created based on current sensory information is represented as an APOC component.

Short-term Skill Memory

This memory contains the skills the agent intends to execute. Each element represents an instance of a long-term memory skill and has concrete arguments. Each element is represented as an APOC component and is linked through A-links and O-links to the percepts and short-term memory concepts which form its arguments. In primitive skills, actions are mapped onto effectors.

Perceptual Buffer

Percepts are represented as APOC components. For example, the literal (#speed car-007 20.3) is represented as a component. A concept connects via O-links to percepts in order to read their values.

To learn hierarchical skills in APOC, an analysis component is connected to all skills and checks preconditions. This component constructs the precondition hierarchy, as described in [62]. In APOC terms, a new component is created for each common precondition of two or more skills and it connects to the skills which have the common precondi-
tion. The link structure thus created determines the memory hierarchy and therefore the skill hierarchy.

A simple example of a translation is presented below. Consider the function

\[
\text{in-lane (?car ?lane)} \\
\text{(lane ?lane ?left-line ?right-line)} \\
\text{(car car?)} \\
\text{(#xdistance ?car ?left-line ?dleft)} \\
\text{(#xdistance ?car ?right-line ?dright)} \\
\text{(< ?dleft 0) (> ?dright 0))}
\]

An APOC translation following the above rules can be seen in Figure 4.9.

It should be noted that there are two instances of the #xdistance concept in this description. We chose this implementation in order to illustrate the flexibility of APOC and to show how an APOC-based architecture could make use of the facilities available in the system, in this case assuming there are enough resources to duplicate a functional unit, in order to maximize system performance. For more details on how resource constraints are used in APOC, see [].

In Figure 4.9, the instance of in-lane, i-l001, sends object data to the lane instance component, l001 through the A-link. Then it observes through the O-link to see whether the object sent is a lane. Similar processes take place with the other instantiated components. For this example, we left the magnitude comparisons, < and >, in the update function, \( F \), of i-l001, due to the simplicity of the functionality represented.

4.2.14 Cognitive Architectures 4: PRODIGY

PRODIGY is a mixture of planning and learning, consisting of a general purpose planner and several learning systems, as seen in Figure 4.10.

Knowledge in this system is represented in terms of operators. A central module decomposes a given problem into subproblems. In PRODIGY, a planning domain is specified as a set of objects, operators, and inference rules which act on those objects. To
translate a PRODIGY system to APOC, we use the following rules:

1. Each operator type maps onto an APOC type component.
2. Each bindings component (instantiated operator) maps onto an APOC component.
3. Each object in the knowledge base maps onto an APOC component.
4. Each control rule maps onto an APOC component.
5. Each goal maps onto an APOC component.
6. Goal and bindings components have a cost field and a computation of cost implemented in the update function, \( F \). Operator types do not have costs; in APOC types are not part of the traversed graph, resulting in a slightly different, though functionally equivalent, structure from the graph described with PRODIGY and shown in Figure 4.11.

7. Wherever applicable, each component computes its own cost in the update function, \( F \). The cost of the top component then represents the cost of the plan.

8. Operators have O-links to object representations or other operators, which determine the meaning of the operator.

9. The central module is implemented in its own APOC component, with connections to all other components in the system. This component has P-links to the Back-Chainer and Operator-Application to determine which executes at each step. Recursion is obtained by repeated application, e.g., by cycling through the architecture.

10. The Back-Chainer is an APOC component.
11. The Back-Chainer can create bindings components and pass as arguments to the new components the ids of objects/operators to which the new instantiated operators should connect.

12. The Operator-Application is implemented in its own APOC component, which has O-links to all objects and operators. This component is connected through an O-link to the Back-Chainer in order to observe the state of the tail plan.

13. Each of the other elements of the system, such as EBL and Hamlet, are separate components, which connect to PRODIGY through both O-links and A-links. These components connect only to those components required for their functionality. For example, QUALITY receives the current plan from PRODIGY and attempts to refine the plan and generate new control rules which will allow PRODIGY to generate better plans. QUALITY may need to connect to all components which are part of the plan, or it may simply operate on an abstract representation of the plan and system state, created by PRODIGY.

14. An external user can create a plan using APOC components. The components which need to use information from this plan can connect to the user-created plan through O-links.

The initial state of a problem is the initial state of the system.

The above examples illustrate how existing models of adaptive and non-adaptive behavior selection, as well as cognitive architectures can be expressed within the APOC formalism in a unified way.

We have seen so far that various architectural mechanisms can be expressed in APOC and that architectures can be analyzed in the framework. In the next chapter we investigate how these characteristics can be used in the process of designing an architecture.
APOC AS A DESIGN TOOL

Section 4.2 has illustrated how various architectural designs translate into the APOC framework. The various translations allow for the definition of a large variety of mechanisms within the same architecture. Concepts from one formalism can often be transferred to another by virtue of a unified representation in APOC. For example, semantic nets, neural nets, conditions-action rules, or conceptual hierarchies can all be defined in a similar way. It is possible to study different designs of mechanisms, e.g., how to arbitrate behaviors, or how to actively manage finite resources at the architecture level. Since algorithms are generally implemented in APOC components, APOC automatically yields a way of distributing computations in terms of asynchronous computational units and their communication links.

It should be noted that the migration of algorithms to architectural components makes it possible to have different algorithms operating in different parts of the architecture, as well as changing these algorithms over the lifetime of the agent. Furthermore, the resource requirements and computational cost of the architecture can be determined and compared to other architectures implementing different algorithms for the same task.

Consider, for example, the use of SOAR features in an ACT-R system. SOAR architectures contain a Chunking Module for production-based systems, described in [91]. This module observes the environment and creates chunks of new, specialized productions which are linked to the goal being pursued. These new rules can then be used in special-
ized contexts. If both architectures are developed in APOC, then the *Chunking Module* can be added to an ACT-R system, where it is connected via O-links to chunks in *Declarative Memory* representing objects and to goals in the intentional module. The new, more specialized rules are automatically used by ACT-R: the rule-selection process is based on a partial match, and the best match should be provided by the specialized rule.

Behavior-based robotics provides another setting where APOC flexibility affords the architecture designer the option to combine architectural methodologies. Schema-based systems and subsumption-based systems have little in common, but in APOC, the two systems can coexist. The two systems can either be in independent parts of the architecture, e.g., by implementing motor control in a subsumption-based manner while visual motor control is implemented using a schema-based approach, or by using a control mechanism to switch between the two methodologies in controlling the same subsystem.

As an example of the need to express different architectural mechanisms in a unified manner, we start in the next section by presenting in detail the arguments for allowing the modification of behavior-selection mechanism in the context of behavior-based architectures. The following two sections then illustrate how APOC can be used to provide this capability to the agent developer. Due to its generality, APOC also supports the description of problems seemingly unconnected to agent design, such as cellular automata. Through APOC, ideas from these problems can be incorporated into the agent-design process. Some examples of other constructs which can be expressed into APOC are presented in Section 5.4.

5.1 Architectural Mechanisms for Behavior Selection

Agent architectures can be classified along several lines based on their behavior-selection strategies. One such categorization separates architectures into cooperative and competitive. In this chapter we discuss the behavior-selection problem. Then we look at
several classifications of behavior-selection strategies, showing how APOC architectures from different classes can be expressed in APOC. We continue by adding a new dimension to agent architectures, that of a dynamic behavior-selection strategy. Next we explore APOC as a tool for enriching agent architecture design through the use of concepts from seemingly unconnected areas such as cellular automata. Finally, we describe the roles APOC can play as an analysis and design tool.

The problem of “behavior selection” in agents has been under investigation for several decades. Two different ideas have come to converge under the term “behavior” and this duality is reflected in the term “behavior selection.” The first meaning refers to the observable actions of an agent. In the second meaning, “behavior” means an architectural component or a mechanism consisting of several, possibly interconnected components, which implements all or part of an observable behavior.

In this thesis, we will use the term “behavior selection” in its second meaning. This allows us to focus on the architectural features of an architecture and analyze the architecture in terms of its constituent components and links. At the same time, focusing on the architectural meaning of the term allows us to avoid at this time the issues of emergence, in which the observable behavior is only partially due to the selected architectural behavior. One example where such a misunderstanding can arise is simple motor control on a slippery surface; due to wheel slippage, the trajectory of the robot may be random, even though, at the architectural level the behavior selected directed the robot to move forward in a straight line.

Another terminological problem with the term “behavior selection” is the sometimes synonymic use of the term “action selection” [67, 104]. There are two reasons for our use of the term “behavior selection” over “action selection.” The first is that most of our experimental work was done in the context of behavior-based architectures, where the basic constituents of architectures are behaviors. The second reason is that the term action
itself can have multiple meanings. It can refer to behaviors in both the above meanings, but it can also refer to simple motor control, as illustrated by Cooper and Shallice [39]. In the latter case, the term behavior is reserved for more complex processes.

In the context of our work, the term “behavior selection” will refer to the process by which a set of components which implements a behavior in the second sense will lose control of the agent’s effectors and another component or set of components will gain control of the effectors. The new set of components will most often implement a new behavior in the second sense. This switch leaves open the possibility that the switch be between different implementations of the same observable behavior. Control of the effectors implies, in APOC terms, that a component has an exclusive link to the effectors, either by virtue of its being the only such link in the system, or by having the effectors ignore input on other incoming links. Losing control then implies either the deletion of the link, or the inhibition of its value at the effectors. Similarly, gaining control of the effectors by a component entails either the creation of a link from the component to the effectors, or uninhibiting the link so that its output reaches the effectors.

Our definition of “behavior selection” has several advantages over other definitions, such as Tyrrell’s [104]. By focusing on the selection of a set of components which have control of the agent’s effectors, we avoid the possible misunderstandings which can arise from the multiple meanings of the word behavior. Moreover, it is irrelevant to the selection process whether the components implement behaviors or actions in any of the meanings of each term. A second advantage is that the definitions of “behavior selection,” losing control, and gaining control cover the entire process of changing the component capable of exercising control over the effectors. A third advantage is that this definition of “behavior selection” generalizes across both architectures where behavior selection is done explicitly through special components, such as schema-based architectures, and architectures where behavior selection is determined by structural features of the archi-
tecture, such as the interconnection of components in a subsumption architecture. We will refer to the first type of behavior selection as explicit, and to the second as implicit in the remainder of this chapter. Finally, this definition of behavior selection is not dependent on the level of abstraction at which architectures are specified.

Based on this definition, we can now categorize different proposals for behavior selection mechanisms in a systematic way based on whether they are cooperative or competitive, implicit or explicit, and adaptive or non-adaptive, and discuss their different properties.

5.1.1 Cooperative versus Competitive Behavior Selection

Following Pirjanian [84], cooperative behavior selection requires mechanisms which achieve some sort of “behavior (or command) fusion,” integrating information from different sources in the architecture before it is passed on to the effectors to produce the current behavior.

Examples are voting mechanisms [87, 89], superposition techniques [12, 19, 53], fuzzy command fusion mechanisms [2, 110], or multiple objective behavior coordination methods [51, 86, 100]. Note that in the extreme case, cooperative mechanisms never actually “select” behaviors, because the set of components implementing the agent’s basic behaviors is permanently connected to the integration component, which in turn is permanently connected to the effectors, as in simple schema-based architectures.

Competitive behavior-selection mechanisms, on the other hand, require the selection of a behavior based on the result of some competition process among different components, possibly followed by the arbitration of the current behavior, if a behavior different from the current one was selected during competition. Examples are priority-based [27], state-based [46, 48, 60, 103], and winner-take-all competition mechanisms [67, 74, 75].

Competitive and cooperative behavior selection-mechanisms are mutually exclusive...
in that the same set of behaviors cannot use both a cooperative and competitive mechanism at the same time. However, it is still possible to use them together in the same architecture as long as there is a way to decide which selection mechanism gets to select behaviors at any given time. For example, a hybrid architecture may consist of a cooperative behavior selection mechanism in the reactive layer, and a competitive mechanism in the deliberative layer, or vice versa.

We will present several examples of architectures with competitive and cooperative behavior selection in Chapter 9.

5.1.2 Implicit versus Explicit Behavior Selection

Behavior-based architectures can also be distinguished based upon how behavior selection is accomplished, i.e., whether it is implicit or explicit. Implicit behavior selection uses structural features of the architecture to select behaviors, e.g., through a hierarchical arrangement of control components as in the competitive behavior selection of subsumption architectures [27], or through the relative strengths of inhibitory and excitatory connections among components as in the cooperative example of Braitenberg vehicles [26], while explicit behavior selection uses specialized components, e.g., as the summation component in schema-based architectures for cooperative behavior selection [12] or the global algorithm which chooses a module in the ANA architecture [67,68] for competitive behavior selection.

Implicit and explicit behavior-selection mechanisms are also mutually exclusive, analogous to competitive and cooperative mechanisms. Yet, as with competitive and cooperative mechanisms, they can coexist in one architecture. A specialized component for behavior selection, $D$, may be part of a hierarchical structure that determines the order in which behaviors get to control the effectors, e.g., based on inputs from the environment. If the behavior implemented in $D$ is selected, then behavior selection proceeds according
to the policy implemented by $D$, otherwise it proceeds according to the hierarchical structure. Note that in this case, behavior selection determined by the hierarchy has precedence over behavior selection determined by $D$. We will present an example of an architecture that combines both implicit and explicit mechanisms in Section 9.2.

5.1.3 Non-Adaptive versus Adaptive Behavior Selection

Since behavior selection is typically a built-in feature of behavior-based architectures, especially in architectures with implicit behavior selection like subsumption, it is not amenable to change in most architectures, e.g., [25, 27, 67, 104]. We shall call behavior selection strategies that cannot be modified throughout the lifetime of an architecture instance non-adaptive.

Some architectures, however, allow for the adjustment or adaptation of behavior selection strategies [47, 80, 109]. We shall call such architectures adaptive. Adaptation in these systems generally consists of either modifying internal parameters which affect the choice of the behavior or modifying the set of available behaviors from which a choice can be made based on a triggering condition. Thus, any modification in the behavior-selection strategy in these architectures is achieved within the context of a fixed strategy and typically only parameters of a specialized component are modified.

One adaptive approach with explicit behavior selection proposed a “hybrid cooperative-competitive” behavior selection strategy [34]. Here, adaptation occurs through reinforcement learning, during which fusion parameters of decision components that integrate the outputs of two behaviors are learnt. Analogous to the above mechanisms, modifications of behavior selection strategies are achieved by varying two parameters of specialized integration components. ¹.

Finally, there are also proposals for implicit adaptive mechanisms (e.g., [83]), where

¹Since the employed fusion mechanism only gives rise to competitive behavior selection in the limiting case (analogous to schema-based approaches), it will be classified as “cooperative,” see also Section 4.2.9
a variant of Hebbian learning in a neural network is used to learn the fusion parameters (i.e., the weights on connections from different behaviors that are combined to yield the overall motor output).

While the previous two dimensions are concerned with properties of the architecture layout, the distinction between adaptive and non-adaptive behavior selection is pertinent to the run-time instance of an architecture. Table 5.1 summarizes the proposed categorization of behavior selection mechanisms in common architectures along all three dimensions.

### TABLE 5.1. EXAMPLES OF BEHAVIOR-SELECTION STRATEGIES CLASSIFIED ALONG THE THREE PROPOSED DIMENSIONS: COMPETITIVE VS. COOPERATIVE, EXPLICIT VS. IMPLICIT, AND ADAPTIVE VS. NON-ADAPTIVE.

<table>
<thead>
<tr>
<th>Non-Adaptive</th>
<th>Competitive</th>
<th>Cooperative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit</td>
<td>Agent Network [67], Bayesian Decision Analysis [60], Foka [46], Probabilistic methods [103]</td>
<td>Lorenz [65], Schema [12], DAMN [88], Balch [19], Jenkins [53], Multiple Objective Behavior Coordination [51, 86, 100], Fuzzy fusion [2, 110], Action Voting [49]</td>
</tr>
<tr>
<td>Implicit</td>
<td>Subsumption [27], Baerends [17]</td>
<td>Braitenberg [26]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adaptive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Competitive</td>
</tr>
<tr>
<td>Explicit</td>
</tr>
<tr>
<td>Implicit</td>
</tr>
</tbody>
</table>

5.2 The Argument for Dynamic Changes of Behavior Selection Strategies

In the following, we consider seven different cases, in which adaptive behavior selection can be advantageous for behavior-based agents, in particular for robots, all of which will be verified experimentally in Sections 9.1 and 9.2.
5.2.1 Case 1: Selection of Sensory Information

Selecting a subset of all available sensory information for perceptual processing can be useful to reduce overall processing and lead to better performance, especially in robots, where sensory input is often unreliable. Sonar sensors, for example, often produce erroneous readings, cameras fail in dark environments, shaft encoder counts do not reflect the actual distance traveled on slippery surfaces. Fortunately, many of these errors can be detected, at least to some degree, by means specific to each sensory modality. For example, frequent and sudden large changes in sonar values, low brightness levels in camera images, discrepancy between motor encoder counts and sonar readings relative to landmarks, all may be indicators of erroneous sensor readings. Behavior selection can then be dynamically adjusted so that it will be based only on the more reliable sensory inputs by automatically eliminating sensory input that is not reliable, either temporarily or permanently, as in the case of a broken sonar sensor that always returns the same reading. By the same token, redundant or irrelevant sensor information, if detectable, can be ignored.

5.2.2 Case 2: Emergency Responses

Embodied agents will typically need fast mechanisms to deal with emergency situations. Such “global alarm systems” [97] must be connected to the sensors and effectors in such a way that they can interrupt any behavior and take control of the agent’s effectors. In other words, behavior selection in a system with alarms might be competitive at the level of the alarm mechanisms, but could be cooperative for other behaviors, i.e., when the alarm is not activated, thus retaining advantages of cooperative behavior selection. While alarms are typically directly associated with emergency behaviors, it is also possible to use alarms only to change the behavior selection strategy temporarily until the state that triggered the alarm has changed, e.g., a particular urgent goal has been accomplished.
5.2.3 Case 3: Infeasible Behaviors

Combining behaviors based on sensory inputs via a fixed mechanism can at times prompt the agent to attempt infeasible behaviors. A simple example is illustrated in Figure 5.1, where an agent, A, is trying to reach an item, the black circle, in the environment. The target item is partially blocked by obstacles, denoted by white circles. While moving down and moving to the left are both feasible behaviors, their straight-forward combination, as it might occur in a schema-based system, produces a behavior (diagonal move) which is not feasible given the current state of the environment.

![Figure 5.1. Examples of an infeasible combination of feasible behaviors (go left and go down for agent A) and a feasible behavior from combination of infeasible behaviors (go right and go down for agent B) in order to get to the goal state represented by the black circle. White circles denote obstacles.](image)

A possible and simple solution to the combination problem is to temporarily change the behavior selection strategy: instead of combining behaviors, individual behaviors are given priority.

5.2.4 Case 4: Extending the Behavioral Repertoire

The converse problem to “infeasible behaviors” is a situation where no sequence of individual behaviors can accomplish an agent’s task (e.g., as in the case of agent B in Figure 5.1, whose basic behaviors are limited to move-forward, move-backwards, turn-left, and turn-right). In that case, temporary combinations of behaviors, e.g., via behavior-fusion might permit the agent to achieve the task. In the example from
Figure 5.1, $B$ would combine **move-forward** and **turn-right** to perform a diagonal movement.

### 5.2.5 Case 5: Mappings between Context and Behavior Selection Strategies

The last two cases illustrated that it is sometimes beneficial to switch between cooperative and competitive behavior selection. In general, such switches will be context-dependent, and can therefore be defined by a mapping between context and a set of behavior-selection strategies or parameters of behavior selection strategies. The context will generally include states internal to the architecture in addition to sensory information.

There are several applications for such a mapping. If several behavioral sequences exist to achieve a particular task, it can be used to implement a context-dependent preference mechanism. For example, a robot assembling cars might perform actions sequentially if run in “demonstration mode,” even though these actions could be performed in parallel. Another example is that of a learning mechanism which could try to establish the best behavior-selection strategy either by systematically trying out all available behaviors, which would allow the agent in the above cases to associate particular environmental setups with behavior selection strategies, or by systematically varying the parameters of a behavior-selection strategy.

In general, mappings between contexts and behavior-selection strategies might be a way to represent solutions for whole classes of problems in a very compact way. They could be either fixed or learned through experience, and might improve the agent’s overall level of adaptivity without the need for a deliberative layer, in which a planner operates on a representation of the environment in order to find a solution to a particular problem, i.e., a plan representing a sequence of actions or behaviors to achieve the task.
5.2.6 Case 6: Attentional Mechanisms

Attentional mechanisms in animals and humans serve the purpose of channeling sensory information and focussing processing on salient or important stimuli. They can also redirect processing resources dependent on the “focus of attention,” for which they often require control of the agent’s effectors as well, e.g., to be able to look in the direction of a loud noise.

Attentional mechanisms can be integrated into behavior-based architectures by allowing them to control dynamic switches among behavior-selection strategies; the behavior selection strategy to which they switch can be either fixed or learned.

5.2.7 Case 7: Learning Behaviors

Often an agent can determine how well it is doing at a given task based on some observable internal or external state, e.g., how much energy is left that can be used for locomotion or how close an agent is to a given goal state based on sensory input. In general, if a performance measure is available within the agent’s architecture, it is possible to define a simple, unsupervised, reinforcement-learning component that can learn a mapping from contexts to behavior-selection strategies based on the performance measure. In the simplest case, the component could automatically switch at random among different behavior selection mechanisms during a “learning phase,” recording context performance pairs, and then eventually always pick the best behavior-selection mechanism based on the learned associations for the given context. Alternatively, learning could be triggered by other components or by repeated failure at achieving a task and proceed in a systematic fashion, e.g., low-cost behaviors are always tried first.

---

2The performance measure can be implicit in the architecture or explicitly represented (e.g., as a numeric value).
5.3 Dynamic Behavior Selection Mechanisms

The examples in Section 4.2 illustrate how existing models of adaptive and non-adaptive behavior selection can be implemented within the APOC framework in a unified way, using a few basic computational components and the four link types among them. A common characteristic of current architectures with adaptive behavior selection is that the adaptation process does not require the agent architecture to undergo any structural change. Rather, parameters of specialized components which function as arbiters are modified to change behavior-selection strategies. In this section, we eliminate the unnecessary restriction of adaptation processes operating only on parameters of specialized components and allow adaptation both to be cooperative and to occur at the structural level of the architecture itself, i.e., to be implicit. Furthermore, we allow for any of the eight possible kinds of behavior-selection mechanisms we distinguished in Section 5.1 to occur at any place in the architecture.

First, we demonstrate adaptive-implicit cooperative and competitive behavior selection by structurally changing an architecture instance at run-time to employ different behavior selection strategies at different times. Specifically, we illustrate the dynamics of such changes by defining architectures for virtual agents performing a two-resource foraging task in a simulated environment. The agents switch between competitive and cooperative behavior-selection strategies.

5.3.1 Switching from Cooperative to Competitive

Consider the example in Figure 5.2. The type level specification shows behaviors $B_1$ through $B_m$ receiving input from the environment. Their outputs are then fused in the fusion node $\Sigma$ to produce the effector commands to be executed by the agent. Type $M$ is a node which observes the state of the agent. If the conditions require it, a node of type $M$ can send a message to a node of type $D$, which can instantiate a node of type $C$. The
function of a type $C$ node is solely to act as an arbiter among the available behaviors by inhibiting some of the outputs going from the behaviors to the fusion node. At run-time and under normal functioning conditions, the state of the system is cooperative, as shown on the lower left of Figure 5.2. Node $d$, which is an instance of type $D$, observes the state of the agent (not shown) but does not interfere with the behaviors performed. When needed, however, node $d$ instantiates node $c$, which leads to a change in the behavior selection mechanism among nodes $b_1, \ldots, b_m$ (lower right in Figure 5.2).

![Diagram showing the dynamic change from cooperative to competitive behavior selection]

Figure 5.2. Dynamic change from cooperative to competitive behavior selection: the type specification is shown on top, a standard cooperative system (A2) is on the lower left, the dynamic system (A1) in its default, cooperative state is on the lower middle, and the competitive state of the system on the lower right.

In particular, consider the case in which one action, $b_1$, represents the water-seeking action of the agent. If the agent’s water level drops below a certain critical threshold, $d$ will instantiate a node $c$, which in turn will allow only $b_1$ to send its output to $\sigma$,
thus suppressing all other actions, e.g., food seeking. This effectively turns what was a cooperative system into a competitive cluster.

5.3.2 Switching from Competitive to Cooperative

The reverse process is illustrated in a slightly different setup in Figure 5.3. Here, the default mode of operation of the system is competitive and it is enforced through node \( c \). However, node \( m \) can send a signal (through the P-link) to node \( d \), which can lead to the deletion of node \( c \), changing the arbitration mechanism to cooperative.

Figure 5.3. Dynamic change from competitive to cooperative behavior selection: the type specification is shown on top, a standard competitive system (A4) is on the lower left, the dynamic system (A3) in its default, competitive state is on the lower middle, and the cooperative state of the system on the lower right.

In our experiments, we will consider architectures where \( m \) can monitor the sensory input to action nodes that are not currently in control, e.g., a water-action node that is
not active, because the agent is moving towards food. If water is nearby, but the agent is going after food, \( m \) will temporarily turn on cooperative behavior selection, which in turn will allow the agent to get water on its way to food.

5.3.3 Multiple Behavior-Selection Mechanisms in APOC

The above examples illustrate the way in which a change in the behavior-selection mechanism of an agent can be effected in APOC. In general, each policy could be implemented as the process section of an APOC node. Whenever a specific policy is to be used, an instance of the corresponding APOC node is instantiated and connected to those nodes already present in the architecture to which the policy is to be applied.

In particular, the cooperative-to-competitive switch is enforced by connecting the policy node to the controlled node through P-links. The policy node can thus directly affect the process associated with each controlled node, e.g., in the case of competitive behavior selection, allowing only one active process in the group of nodes it controls.

Consider the example in Figure 5.4 as a generalization of the two cases presented above. In this figure, the meta-processing layer illustrated succinctly by node \( m \), sends a decision to node \( d \) as to which behavior selection scheme should be used. Based on this decision, node \( d \) instantiates either node \( c_1 \), or \( c_2 \). Since both nodes are connected to behavioral nodes \( b_1 \) through \( b_m \) through O-links, they can extract from these nodes the information required for the selection scheme they implement, provided that the nodes themselves are equipped to handle the scheme. For example, \( c_1 \) could implement a voting mechanism for behavior selection and it observes a structure, representing votes, present within each node implementing a behavior. The information collected is tallied and \( c_1 \) can then pass the corresponding command to the effectors.
5.4 Expressing Other Concepts in the APOC Framework

A natural extension of the type-token distinction exemplified in Section 4.1 is the implementation of “developing architectures.” An introduction to the concept of developing architectures and examples of their usefulness are presented in the following sections.

5.4.1 Developing Architectures

With developing architectures the designer only provides “limits” for the development of the architecture, e.g. resource limitations, or limiting a particular type to using A-links,
but not the particular architectural layout. The possibility arises that useful functional capacities which were not present in the original architecture would develop through interaction with the environment.

The neural network example in Section 4.1 also demonstrates this “bootstrapping” process in APOC mentioned in Section 3.2.1, which effectively allows designers to specify “growing” or “developing” structures, e.g., a layered network that will eventually develop into a fully connected network. In particular, the process can give rise to adaptive architectures which can change over time, e.g., as a result of some sort of learning process.

Figure 5.5 shows an example of such an architecture, an ART network [32], which can learn to categorize its inputs over time by adding new “categorization components” to the architecture to represent the newly learned categories. In this example, $E$ represents the external inputs, in this case coming from a sensory component; $G$ represents gain control; $T1$ represents the component type for components in the input layer; $T2$ represents the component type for components in the category representation layer; and $R$ represents the reset of short-term memory.

![Figure 5.5. ART networks in APOC. See text for an explanation of the notation.](image)

We have affirmed that APOC supports developing architectures and that, as a result, agents can be created whose architectures are at any point in time partially determined
by their interactions with the environment. In the remainder of section we present two detailed examples of environmental interaction. The first is an elaboration of the ART network example presented earlier, while the second illustrates a simple planning example.

5.4.2 Environment Driven Changes in ART Networks

In the previous section we introduced the idea of “developing architectures” with an ART network, a scenario we now explore in more depth. Figure 5.5 provides the basic structure needed for the development of the network.

The relation of the type-level links to the run-time machine depends on the behavior of each C-link. With the proper C-link definitions, e.g., C-links to instances of type $T_1$ copy over all links from the parent component, the final structure is shown in Figure 5.6. This structure mirrors the example presented by Carpenter [32]. In the figure, the way the behavior of the C-links in the type-structure is defined determines the actual link structure for incoming links to the $T_1$ and $T_2$ groups. In this case, it would be appropriate to have connections to each of the components in those groups. However, since any link structure can be obtained in APOC, the links are represented in a more abstract manner.

One advantage provided by the APOC framework is that the number of categories may vary according to the resources which the system can allocate to the process. For example, the network could start with 10 components of type $T_1$ and vary that number up to the maximum of 50 according to environmental circumstances.
In the above example, the external inputs are defined by a single unit, \( E \). However, it is reasonable to generalize upon the example presented and consider the possibility of subjecting the external inputs to resource constraints. This can be done with relative ease in APOC given the broad definition given to APOC components. Since each APOC component can be broken down to a minimal complexity, \( E \) can be viewed as being composed of several units, each of which takes care of one environmental input. The quantity of information used for categorization can then be modified dynamically based on resource constraints and/or environmental complexity. The resulting instantiated architecture is shown in Figure 5.7.
5.4.3 Incremental, Resource Constrained Planning

Another situation where APOC architecture development capabilities can be successfully used involves planned sequences of actions. Consider the example in Figures 5.8 and 5.9, where a physical agent has a choice of three different actions: “move right”, “move left”, and “move straight”. The agent’s goal is to move from its current location at point A to a point B, say. To support the planning process, the agent has an environment simulation module, in which plan actions can be simulated instead having to execute them in the real world.

Figure 5.7. Sensory processing example showing the final architectural state with resource-constrained environmental inputs

Figure 5.8. Planning example: specification
Figure 5.8 contains a four-component assembly which defines the type relationships in the architecture:

- component $P$ represents a planning type;
- component $R$ represents a type which simulates a right turn in the environment simulation module;
- component $L$ represents a type which simulates a left turn in the environment simulation module; and
- component $S$ represents a type which simulates a finite forward move in the environment-simulation module.

The component links in the figure indicate that components of type $P$ can instantiate components of types $R$, $L$, and $S$. Conversely, components of types $R$, $L$, and $S$ can instantiate components of type $P$. The implication of this circular C-link structure is that components of type $P$ can instantiate components for all actions which are currently feasible, exploring multiple potential futures simultaneously.

The run-time system changes which can occur in the architecture shown in Figure 5.8, are described next. The $P$-component checks for completion of the system-wide goal, e.g., reaching $B$. If the goal has not been reached, the $P$-component makes a decision on which of the $L$-, $R$-, and $S$-components should be instantiated and adds the necessary incoming activation links from the “environment simulation module.” Each instantiated component receives activation inputs from an environment simulator and computes its activation level. The $P$-component then uses the information gathered through the $O$-links to decide which component is most likely to lead to goal completion and therefore which component will carry out its action, a process similar to the arbitration found in contention scheduling [39]. The best component then gets to instantiate another $P$-component, and the process continues until the goal is completed.

It should be noted that the run-time evolution of the system is dependent on the internal computations of the components and the environmental input. With a different
definition of \( P \)-components, the system could be used to explore the state space in a limited depth-first-search manner with backtracking. For example, the \( R \)-component always instantiates a \( P \)-component first. This, in turn, instantiates another set of the three components, where the \( R \)-component instantiates a \( P \)-component, and the process is repeated. Here, too, each \( P \)-component checks whether the goal is satisfied, continuing the recursive instantiation process until the depth limit, i.e., the maximum number of components of type \( P \) that can be instantiated, is reached. If the goal is not yet reached, the \( P \)-component finishes its processing. Its parental \( R \)-component will terminate as well, and the \( P \)-component that instantiated the terminated \( R \)-component will continue with the instantiation of the \( S \)-component, and later, possibly, the \( L \)-component until a path to the goal is found.

\( C \)-link behavior already provides a benefit in resource allocation, in that an \( L \)-type component would not be instantiated if the agent finds itself next to a wall on its left side. Additionally, once the action associated with a particular step in the planning process is executed, \( C \)-links allow the release of those resources associated with unfollowed branches in the planning tree.

If actions, such as “turn right,” are known not to fail, it is beneficial to remove all components of the graph once the physical agent has performed their associated actions. If, on the other hand, actions are susceptible to failure, keeping the components along the execution path in use, up to the resource limitations of the system, can lead to additional advantages. For example, if an action fails, the backtracking process only involves allowing the component one \( C \)-link higher in the structure from the failing component to re-execute its action, e.g., if a “turn right” fails, the planner may delete the \( R \) component and try to find an alternate path by either moving forward or turning left.

Another advantage provided by the APOC framework relates to the system’s knowledge of the world. Planning processes usually create plans for several steps in advance
of the related physical execution. In the physical world, conditions are changeable. Con-
sider the case where point $B$ is defined as the known location of a resource the agent needs
to obtain. If, at the conclusion of the physical actions, the resource is not found, either
because its location has changed or the planning process was erroneous, it is now easy to
retrace the last few steps that the agent has performed, until the state of the environment
matches the expected state of the planning process – perhaps even returning to its starting
point, $A$. With a system like the one in Figure 5.9, this retrace can easily be accomplished
by following the $C$-links backwards through the graph.

![Environment Simulation Module](image)

**Figure 5.9. Planning example - sample instantiation**

Figure 5.9 shows the first three steps of a planning planning process developed from
the structure of Figure 5.8. In two instances, components were left uninstantiated: the
$R$ and $S$ components did not instantiate a $P$ component for the second step, while the $S$
component was left uninstantiated for the third.

The general nature of the framework can be put to use in other scenarios. We continue
with an example which illustrates the ease with which other specifications can be translated into APOC. Then we look at a simple example where each APOC component is a simple agent, in the context of cellular automata.

5.4.4 Use Case Map Example

By allowing the architecture to be specified at various levels of abstraction, the APOC architecture specification can be so abstract as to be a direct mapping from a Use Case Map (UCM) description of the system. This allows the analysis of top-down decomposition models such as described by Buhr, et al. [30].

Use Case Maps are defined through six types of components: paths, waiting places for stimuli or events, denoted with circles in UCM diagrams, timers, which are waiting places with an upper bound on waiting time, denoted with “clock faces,” bars, which are markers for ends of paths, as well as beginnings and ends of concurrent paths, basic paths, which are paths starting at a waiting place and ending at a bar and directions. For a direct, though not necessarily most efficient mapping from UCMs to APOC, each waiting place, timer, and bar is mapped to a separate APOC component.

The two examples in Figures 5.10 and 5.11, taken from Buhr, et al., [30] illustrate how such mappings may be performed. The UCM diagram is a generic example, but it can be applied to the following scenario of action selection:

- The left waiting place represents inputs from external stimuli;
- The left bar represents computation based on the internal state of the agent;
- The right waiting place represents a synchronization component for the data. The synchronization is based on internal state with data coming from a higher level planner; and
- The right bar represents the end of computation where all three pieces of data have been gathered and an action has been selected for the agent.

It should be noted that the APOC links depicted are, with two exceptions; generic links, the actual type would depend on the specific system depicted. Also, the conversion
has been done in a manner which illustrates the correspondence that can be drawn between UCM and APOC component, rather than the most concise manner APOC allows. For a more efficient translation, for example, the \( b \) components could be combined with C-links into a single \( B(2,3) \) type component.

Figure 5.10. Example of a Black Box UCM Representation in APOC

Figure 5.10 illustrates a high-level UCM description and requires an abstract APOC component to represent the entire system. Figure 5.11 illustrates how a more detailed map can be ported to APOC.
5.4.5 Cellular Automata Simulation

For cellular automata, we start by defining the basic entities in the APOC framework. A cellular automaton is an array of identically programmed automata which interact with one another [107]. Each cell is defined by

- state: a variable that takes a different separate for each cell; cell state can be easily implemented as part of the update function of an APOC component;

- neighborhood: the set of cells with which the cell interacts; the neighborhood of a cell can be described in terms of those cells to which it is connected via C-links; and

- program: the set of rules which define how the state of the cell state changes in response to its current state and that of its neighbors; this set of rules is easy to implement in the update function of an APOC component.

Consider the simple example of the game of life as illustrated in Figure 5.12, with the following rules:

1. A living cell with only one or no living neighbors dies from isolation;
2. A living cell with four or more living neighbors dies from overcrowding;
3. A dead cell with exactly three living neighbors becomes alive; and
4. All other cells remain unchanged.

![Game of life example](image)

Figure 5.12. Game of life example

The APOC implementation of cellular automata requires each cell to keep track of its overall position in the system in the form of an integer coordinate pair. With that provision, the implementation is made as follows:

![APOC implementation](image)

Figure 5.13. Game of life - sample APOC initial configuration

- First, specify an initial configuration through the type configuration, as seen in Figure 5.13. The left side indicates the most concise method of specifying the game. For that specification to be feasible, type L must contain the initial configuration of the system. The component arrangement on the right is specified according to the life example above for ease of understanding. All components in the type specification are identical to one another, with the exception of the location, which is specified for each instance produced during the development of the system. It should be noted that the specification on the left is rather more general, as the resource restriction is a general one. This implies that the 50 units could be instantiated by a single cell and its descendants if permitted by the configuration, as opposed to the ten instances per “type” indicated in the right side configuration.
At instantiation, each component attempts to instantiate a “dead” cell at the eight positions around it. This simply means that a cell of coordinates \((x, y)\) attempts to instantiate cells which are given coordinates \((x - i, y - j)\), with \(i \in [-1, 0, 1]\), \(j \in [-1, 0, 1]\), and \(i\) and \(j\) not simultaneously 0. If a cell with those coordinates already exists, the component simply connects to it through a C-link, as illustrated in 5.14.3

![Game of Life: APOC configuration](image)

Figure 5.14. Game of life: APOC configuration

- Each cell checks the number of incoming C-links and changes its incoming C-links and changes its status according to the rules described above. A ‘dying’ cell deletes all its outgoing links.

- Finally, a dead cell with no incoming C-links deletes itself.

With the set-up described above, a dynamic game-of-life system whose size need not be limited at design time is easily implemented via APOC constructs. The system described above will have the live-cell configuration shown in Figure 5.15 after one iteration.

---

3For the sake of readability, only C-links are shown in Figures 5.14 and 5.15. In the actual system, each link is accompanied by an O-link
The basic concepts described in this section can be extended to other cellular automata problems. The cellular automaton example is yet another illustration of how the structure of a system can change due to interactions among its components. In the following section we investigate the use of hierarchical information in architectures which have undergone structural changes due to interactions between the agent and its environment.

Environmental interaction can have different set of effects on agent architectures. For many agents, prolonged environmental interaction can lead to the development of complex systems, performing specialized functions geared towards the survival of the agent. In such circumstances, gaining an understanding of the role that various components play in the architecture becomes an extremely difficult task in the absence of specialized architecture analysis tools. It is in this context that we provide a first look at a process to analyze these self-developing architectures.
5.5 Hierarchies in Non-Hierarchical Architectures

Architectures defined in APOC often do not have explicit hierarchies defined at design time. However, C-links impose a degree of structure on these architectures. It is this structure which can be exploited in order to analyze the architectures in an attempt to determine the complexity of the computation performed in the update function or in the associated process of each component.

Even though APOC architectures are specified at the level of types of instances, thus hiding the underlying structure to a certain degree, certain relations can be distinguished among both type-level and token-level components connected via C-links. These relations can be broken down to two different classes:

- C-link connection from a complex component, type or token, to a simpler component. For example a type component representing a complex goal connects to another type component that represents a subgoal.
- C-link connection between two related components of similar complexity (e.g., as seen in the planner example above)

This breakdown can be used to identify substructures as defined by the network of C-links within an architecture. Such a structure can be identified by following an outgoing C-link from the type-level description of the architecture or the running virtual machine. Two possibilities arise: that the chain (graph) of links eventually ends in a component with no outgoing C-links, or that the original component is encountered a second time. With this in mind, we define the abstraction level of a component \( n \) in the run-time architecture as the maximum number of consecutive C-links that can be followed out of \( n \) until either a component with no outgoing C-links is encountered, or component \( n \) is reached again without repeating any C-links along the way.

Following this definition, the algorithm for determining the abstraction level of a component \( C \) takes two arguments: a directed graph \( G \) and component \( n \). The graph

---

4Hierarchies can be explicitly defined, as in the case of subsumption-based architectures.
$G = (V, E)$, where $V$ is the set of type components and $E$ is the set of C-links. The return value of the algorithm is the level of abstraction of component $C$.

```
FIND-ABSTRACTION-LEVEL(G,C)
pathLength ← 0
for each vertex $u \in V[G]$
    if $u$ has no outgoing C-links
        do $pathLength_u ← \text{FLSP}(C,u,G)^5$
        if $pathLength_u > pathLength$
            $pathLength ← pathLength_u$
        $pathLength_C ← \text{FLSP}(C,C,G)$
        if $pathLength_C > pathLength$
            $pathLength ← pathLength_C$
return $pathLength$
```

In the algorithm above, FIND-LONGEST-SIMPLE-PATH is a modified shortest path algorithm [40] and returns only the length, in number of links, of the longest simple path between $C$ and $u$. In the next section we look at how abstraction level information applies to some of the examples presented in this paper and what that information could tell us about the system if no prior knowledge of the system existed. In the next section we will see how this hierarchical information can be used in some of the examples presented in this section.

5.5.1 Uses of Hierarchical Information

In this section we consider how the abstraction level concept applies to some of the examples presented earlier in the chapter and we investigate the type of information that abstraction levels can provide to an observer without prior knowledge of the systems.

In Figure 4.2 there are only three type components arranged in a strict hierarchy with respect to C-links. Applying the abstraction level algorithm from Section 5.5 to the type level description gives component $T1$ an abstraction level of 1, component $T2$ an abstraction level of 1, and component $T3$ an abstraction level of 0. In this case the implications of the respective abstraction level numbers are clear: component $T1$ represents either the
most “abstract” type in the system, or the level at which external inputs enter the system; component $T2$ represents an intermediate layer; component $T3$ represents either the basic action or the output layer of the system.

Planning In Figure 5.8, there are four components, each of which has an abstraction level of 2. Since no component has a higher level of abstraction than another, the only information which can be extracted from the architecture is:

- Since there are C-links travelling in both directions between $P$ and each of $L$, $R$, and $S$, there is an interdependence between component $P$ on one hand and the $L$, $R$, and $S$ components on the other;
- Since all components have outgoing C-links, all four components have some capability of processing environmental/internal information which may lead to the instantiation of a new component; and
- Since no component has an abstraction level of 0, it is likely that all four components have the ability to determine whether the goal of the system has been achieved.

Cellular Automata Simulation The cellular automata illustrates one possible problem with the abstraction level analysis. If the analysis is performed on the left side of Figure 5.13, it seems that there is one type, which is self-sufficient. It receives information from the environment, processes it, and makes a decision based on that processing. However, if the analysis is performed on the more detailed original specification, different levels of abstraction are obtained for each component.

In this case, performing the analysis on the run-time virtual machine yields little additional information: since C-links run bi-directionally between adjacent cells, the only inference possible is that there is an interdependence among all adjacent components in the run-time virtual machine. In an environment where links only went from the instantiating component to the instantiated one, however, a dating process could be performed through a hierarchical analysis of the virtual machine.
This is an example of the type of situation for which a more refined notion of “level of abstraction” than the one provided above is required to be able to extract automatically the functional organization of an APOC architecture. Nevertheless, the previous examples show that with a notion as simple as the one defined valuable information can be already extracted from the C-link architecture.

We have seen in the last two chapters the characteristics of the APOC framework, how various architectural designs can be expressed in APOC, and how novel architectural concepts are opened up through the use of the framework. In the next chapter, we look at the APOC Development Environment, ADE, a software tool which offers agent designers a way of using many of the characteristics of APOC.
CHAPTER 6

THE ADE DEVELOPMENT ENVIRONMENT

6.1 Background

We start with a brief overview of the main characteristics of ADE, which we then compare to the features of several other agent architecture environments.

6.1.1 The Basic Characteristics of ADE

ADE stands for “APOC Development Environment” and is a toolkit implementing the APOC framework [9, 93, 94]. ADE inherits in practice many of the characteristics exhibited by APOC. These characteristics include the option of including various architectural designs in a single architecture, e.g., by using perceptrons [72] in the same architecture as a full-fledged condition-action rule interpreter such as SOAR [61, 90].

ADE also allows computational components of an agent’s architecture to be created and destroyed during the life-time of an agent by other architectural components, as will be seen in Chapter 9. This allows agent developers to specify agents that use minimal resources for task completion, develop specialized subsystems, or are adaptive (e.g., become more deliberative over their life-span).

In addition to expressing and implementing existing architectures, ADE can be also used to define new concepts and implement new architectures. For example, in the previous chapter we introduced “dynamic architectures” which are capable of modifying
themselves over time by altering their own description, e.g., as part of a learning process.
An ADE implementation of a dynamic architecture is provided later in this thesis.

ADE provides functionality which aides in the implementation of agent architectures for simulated and robotic agents. An integrated server-client subsystem allows components of the architecture to connect directly to robots, as in the robotic examples in Chapter 9, or remote agents in a simulated environment in order to control them, allowing a single distributed architecture to control multiple agents or devices. ADE was particularly structured with the goal of designing complex robotic agents in mind. The features of ADE which were driven by this goal include

- Support for building more complex components out of simpler ones using a “grouping mechanism” for components;
- Support for “online inspection and modification” of all parts of the architecture. For example, components and links can be removed and new ones can be added in the running virtual machine; and
- Support for distribution of the architecture over multiple hosts in a platform independent way, as real parallelism is required for fast, real-time processing in complex agents.

A graphical user interface provides the architecture developer with easy access to different parts of the architecture. This interface allows the user to inspect and, perhaps, modify the state of architectural components, providing a increased control for the developer over the structure of the agent as well as making this control as direct as possible. APOC components can be “dropped into” a workspace where they are depicted as nodes in a graph, whose edges are ADE links. Relevant information about each computational component can be viewed and modified by clicking on its graphical representation. The presence of four types of links indicates that different modalities of communication take place between components, which can be seen in the graphical representation of the architecture. The actual data carried through each link can also be displayed in the graphical interface. Most importantly, multiple designers can work with ADE simultaneously, allowing for a distributed, collaborative design, test, and run-time environment.
ADE is not limited to architectures of single agents. Rather, it is possible to define multi-agent systems at the level of individual perceptions and actions in terms of the ADE tool: each individual agent is modelled by a subset of APOC components, which in turn have O-links representing the perceptions of the agent, and A-links representing the actions of the agent. To model procreation in biological systems, C-links can be used to allow agents to instantiate copies of themselves. In general, ADE can be used to model both centralized- and distributed- control systems, providing additional evidence for the flexibility of the tool and its potential usefulness [93].

In conclusion, ADE combines a broad range of existing and new features, from single-agent toolkits to multi-agent frameworks, all of which are aimed at simplifying the process of architecture development for virtual and robotic agents in single and multi-agent environments. These features will be discussed in more detail in later sections describing the user-interface and supporting environment for the framework.

6.1.2 Other Agent Tools Compared to ADE

DACAT [20, 21] is an architecture-design tool providing the user with a set of competencies from which the user can choose the ones relevant to the agent being designed. The user can augment the set of competencies to provide arbitrary functionality, e.g., by grouping the competencies and resources into modules and produce a description of the agent architecture. Like ADE, DACAT provides a fully graphical environment, in which the relationship among architectural elements is visualized. However, DACAT functionality is limited to indicating the structure among components at the level of the functionality of an agent, without actually implementing it, whereas ADE allows for the implementation, running, and testing of any architecture specified within it.

IBM’s ABE [1] is a tool which provides some architecture design support, e.g., a set of adapters for agent-environment interaction, engines, which are forward chaining
inferencing tools, and libraries, which offer support for rule and fact authoring tools, to organize, group, and control the inferencing materials that are used by the engine. However, ABE imposes a rule-based design philosophy on its agents, in contrast to ADE, which supports rule-based systems, but also allows for alternative architectures not based on rule interpreters, such as subsumption architectures [27].

Many agent systems are concerned with mobile software agents, which can roam the internet. These systems, e.g., AGENTBase [4], ADVENTNet Agent Toolkit [3], AGLETS [5], BDIM/TOMAS [31], RETSINA [102], focus on supporting efficient and secure communication among agents as well as improving their mobility. However, they do not provide support for the distribution of the components of an architecture over multiple computers unless these components are implemented as agents themselves, i.e., each component as a “complete agent architecture,” which not only complicates the architectural design, but can also lead to reduced efficiency due to communication overhead. Furthermore, only limited support is provided for the design of an agent architecture beyond the communication APIs and virtually no support exists for robotic agents in these systems. In contrast, in ADE there is no difference between robots and virtual agents from a designer’s perspective. ADE also allows designers to implement any architecture methodology based on its implementation of the universal architecture framework APOC.

The AGENT FACTORY system [36, 37] is an environment for agents which use BDI architectures [55]. It is similar to ADE in that it provides support for an agent architecture design, from a high-level specification of the architecture to its implementation and deployment and, allows the definition of agents which are not strictly based on the BDI framework. Still, the main focus of the AGENT FACTORY system is on BDI-based software systems, and therefore differs markedly from ADE. Furthermore, it neither provides ADE’s seamless support for single and multi-robot systems, nor ADE’s capability
of distributing architecture components over multiple computers in an OS-independent fashion.

SIMAGENT is a toolkit designed specifically for the exploration of agent architectures. Like ADE it does not require or impose a particular architecture paradigm. Rather, it supports the specification of architectures at various levels of complexity, e.g., symbolic mechanisms can coexist and communicate with neural networks. However, SIMAGENT only provides basic library functionality for the design of agent architectures in single and multi-agent systems, e.g., a basic agent class and a condition-action rule interpreter. Unlike ADE, SIMAGENT currently has no support for distributing agents over multiple hosts or for controlling robots.

In conclusion, ADE integrates desirable features from different agent systems including a general architecture framework APOC for the definition of agent architectures, support for distributed architectures which can change dynamically, support for communications among agent and agent mobility. None of the agent systems discussed above combines all of these features within one system. Additionally, ADE provides seamless support for single and multi-agent architectures for virtual and robotic agents and a user-friendly, multi-user graphical interface that allows designers to work collaboratively on agent architectures. In the following sections, we will first present details about the implementation of the APOC framework, which gives ADE agents their architectural flexibility, and then describe the graphical user interface and the supporting environment.

6.2 ADE Building Blocks I: The APOC Framework

The APOC agent architecture framework consists of components and links among them. In this section, we describe the basic functionality of components and links in ADE in relation to their APOC counterparts. In particular, we discuss how components can be extended and connected to form an agent architecture.
6.2.1 ADE Components

Following the APOC formalism, ADE components are general autonomous control units that are able to (1) update their own state, (2) influence each other, and (3) control an associated physical or computational process. The process associated with an ADE component can be used for such functions as sending motor commands to a robot or running a parsing algorithm in a virtual agent which checks web pages for particular content. For physical processes, an ADE component can be viewed as a controller in the sense of control theory and for computational processes as a process manager in the sense of operating systems. ADE components have input and output ports, which can be connected to output and input ports of other components, respectively, via ADE links to form an architecture.

Once components are running, i.e., they are instantiated in a virtual machine, they are self-sufficient entities that behave according to their specifications as determined by their initial states, their associated processes, and their update functions. The state of an ADE component can be defined as

\[ \langle \text{act}, \text{pri}, \text{pro}, \text{inst}, F, \text{in}, \text{out} \rangle \]

where \( \text{act} \) is the activation level, \( \text{pri} \) is a pair containing the current and the maximum priority levels, \( \text{pro} \) is a triple containing the process state and the process associated with component as well as the operation performed on that process, \( \text{inst} \) is a pair containing the current instantiation number and the maximum number of instances of a component of that type, \( F \) is the update function, \( \text{in} \) and \( \text{out} \) are respectively sets of input and output links of the component.

The basic functionality of ADE components is implemented in the \( \text{APOCNode} \) class from which all user-defined components are derived. A user-defined subclass can re-
define several predefined functions to deal with the states of links and the state of the
associated process. While the state transition of the associated process is determined by
its current state, RUNNING, INTERRUPTED, or STOPPED, and the incoming information
on the P-links according to the APOC specification, e.g., a running process that receives
the SUSPEND signal will be interrupted, user-defined classes can redefine the methods
that will be called after each state transition is complete. ADE also allows users to define
methods for processing incoming A-links and O-links, as well as outgoing A-links, P-
links, and C-links, in subclasses of APOCNode. Incoming C-links and O-links are again
processed automatically according to the APOC specification.

The basic template of a user-defined class derived from APOCNode is given below:

```java
import apoc.APOCNode;
import apoc.ActivationLinkInterface;
import apoc.PriorityLinkInterface;
import apoc.ObserverLinkInterface;
import apoc.ComponentLinkInterface;

public class UserNode extends APOCNode 
    implements Serializable, Runnable, Remote {

    /* functions to control the associated process */
    public void processNoop() { ... }
    public void processResume() { ... }
    public void processStart() { ... }
    public void processSuspend() { ... }
    public void processReset() { ... }

    /* functions to process information on incoming links */
    public void inputProcessingA() { ... }
    public void inputProcessingO() { ... }

    /* functions to send information to outgoing links */
    public void outputProcessingA() { ... }
    public void outputProcessingP() { ... }
    public void outputProcessingC() { ... }

    /* additional update of the state of the component */
    public void selfUpdate() { ... }

    /* list of graphical entities to be displayed for this component */
    public Vector extendComponent() throws RemoteException { ... }
}
```
Each component also provides the user with the function \textit{extendComponent} which specifies the data of the component that can be observed, displayed, and subsequently modified through the ADE graphical interface.

6.2.2 ADE Links

ADE components are connected to other components through one of four ADE links, analogous to their APOC counterparts: A-link, P-link, O-link, and C-link.

Since each component needs to keep track of its incoming and outgoing links, ADE provides eight \textit{JAVA vectors}, four for each link type: \textit{inActivations}, \textit{outActivations}, \textit{inPriorities}, \textit{outPriorities}, \textit{inObservers}, \textit{outObservers}, \textit{inComponents}, and \textit{outComponents}. Each vector has other vectors as elements, which in turn contain individual links. In the following, we briefly describe the functionality of each link type.

\textbf{A-links} \hspace{1em} \textit{Activation links} are the most general means by which components can exchange information. The state of an A-link is given by the tuple

$$\langle S, R, act, F, t \rangle$$

where $S$ is the component providing the data to the link, $R$ is the component receiving the output, $act$ the data transmitted through the link, $F$ is the operation performed on that data, and $t$ is the time it takes for data to traverse the link. The purpose of an A-link is to connect two ADE components and serve as a transducer.

A-links can be used in a variety of ways. In the simplest case, they function as mere connections between input and output ports of ADE components, i.e., inputs to links are identical to their outputs. Furthermore, an A-link can be used transform the input, e.g., in case of numerical values it could “scale” the input by a particular factor, analogous to the “weights” on connections in neural networks.
A-links provide two functions: `setData` to place data on an element of the `outActivations` vector, and `getData` to retrieve data from a link. As an implementational decision, data passed along links must implement the `APOCObservable` interface. The advantage of this approach is that the interface provides a `get()` function which allows access to the internal data of the object. Thus, for example, an operator on an A-link can gain access to the data passing through the link, without having to first check the type of that data and then call a type-specific function such as `doubleValue()` in the case of an argument of type `Double`. In the case of numerical operators which apply to both integer and double arguments, this can be particularly useful. It should be noted that, due to limitations imposed by JAVA for remote procedure invocations, basic types such as `int` cannot be used on ADE links.

P-links  *Priority links* are intended to explicate the capacity of components to control other components’ associated processes. They are the only means by which ADE components can control processes of other components; since no link has a process associated with it and components can only be connected to other components via links, ADE components could not control any process otherwise. The state of a P-link is given by the tuple

\[
(S, R, pri, op, t)
\]

where `S` is the component attempting to take control of the process associated with `R`, `pri` is the priority of `S`, `op` is the operation which `S` attempts to effect on the process associated with `R`, and `t` is the time it takes for data to traverse the link. ADE provides five operations to be performed on processes, using the constants

- `PriorityLink.RESET`,
- `PriorityLink.START`,
- `PriorityLink.SUSPEND`,

101
- `PriorityLink.RESUME`, and
- `PriorityLink.NO_OP`.

A P-link effectively passes the process control request of an ADE component on to the component it is connected to through the P-link. Process control requests can be used to implement various control mechanisms, in particular, hierarchical preemptive process control. ADE handles priority signals by computing the maximum value of the priorities received on incoming P-links. If the maximum priority is greater than the priority of the component itself, then the request received from the component of greatest priority is honored. Alternately, if there is a tie in components of maximum priority and their requests conflict, `PriorityLink.NO_OP` is performed.

In embodied agents, such as robots, P-links could be used to implement emergency behaviors: the component with the associated emergency process would have the highest priority in the network and be connected to all the other components controlling the agents behavior, which it could suppress in case of emergency, thus implementing a “global alarm mechanism” as described by Sloman [97]. Analogous to A-links, P-links provide `setData` and `getData` functions to place on and retrieve data from a P-link.

The O-link **Observer links** are intended to allow components to observe other components’ inner states without affecting them. The state of an O-link is given by the tuple

$$\langle S,R,D,t \rangle$$

where $S$ is the component observed by $R$, $R$ is a component which requires data from $S$, $D$ is the information passed from $S$ to $R$, and $t$ is the time it takes for data to traverse the link.

The O-link operates independently of the update function in components; it retrieves information by observing a component, rather than having data placed on the link. Thus,
The only operation available on the O-link is *getData*.

The C-link  *Component links* are used to instantiate and remove instances of ADE components at run-time and are the only type of component that can instantiate or terminate an ADE component. C-links and are themselves only instantiated by ADE components. The state of a C-link is given by the tuple

\[ \langle S, R, D, L, t \rangle \]

where \( S \) is a component which attempts to create a new component, \( R \) is the component instantiated by \( S \), \( D \) is information about the links which can be instantiated through this C-link, \( L \) is the set of links already instantiated through the C-link, and \( t \) is the time between the activation of the link and the creation of the component.

A C-link contains information about the type of component it can instantiate and the kinds of links it can instantiate to connect to that component. A component can create and trigger the function of a C-link by issuing a *createCLink* command. This command creates a new C-link, triggers the link to create a new component, and automatically adds the newly created link to the *outComponents* vector of the component.

Two operations are available on a C-link: activation and deactivation. On activation it can: (1) create a component, or (2) create a component and links to that component, or create links to the component previously created by the link. In order to separate functionality, several functions are provided for C-link activation and deactivation. Some of the available activation functions are illustrated below:

```java
((ComponentLinkInterface)outcomponents.elementAt(0)).activatecomponent();
((ComponentLinkInterface)outcomponents.elementAt(0)).activateLink(ComponentLink.PLink);
((ComponentLinkInterface)outcomponents.elementAt(0)).activateAll();
```
The first example creates a component and connects the controlling component to the first via a C-link. The second example presupposes an existing component created through the C-link. The C-link simply creates a P-link to the created component. If such a component does not exist, the function call fails. Finally, the third example creates a new component, and all the links which are defined as being available for instantiation through the C-link. Each C-link keeps track of the architectural elements created through activation functions.

To reverse the action of a C-link, several options are available. *deactivatecomponent* deletes the component instantiated through the C-link if the only incoming link to that component is the current C-link; otherwise the operation fails. The second deactivation example deletes a link created through the C-link, while the final example deletes all elements–components and links instantiated through the C-link whose *deactivateAll* method is called.

```java
((ComponentLinkInterface)outcomponents.elementAt(0)).
deactivatecomponent();

((ComponentLinkInterface)outcomponents.elementAt(0)).
deactivateLink(ComponentLink.PLink);

((ComponentLinkInterface)outcomponents.elementAt(0)).deactivateAll();
```

Since C-links are the only mechanism through which architectural changes are effected, they play an important part in resource allocation and arbitration. If A-links or P-links are used in conjunction with a C-link, activation- and priority- based mechanisms can be used to trigger the action of the newly instantiated component. For example, a *PriorityLink.START* signal can be sent through the P-link to the new component as soon as both the component and link are created.
6.3 ADE Building Blocks II: The User Interface

The ADE environment was designed to allow users to access, inspect, and modify an architecture at any time during its development process: from the original design of the architecture, to the testing of components, to the execution of the complete architecture. The system is divided into an architecture layout section and a virtual machine section. In the former, the components of the architecture are specified and the connectivity among them is established, thus defining the overall architecture layout. In the latter, the running architecture is maintained, which is updated dynamically and subject to run-time modifications.

To reflect this conceptual division into architecture layout and running virtual machine, the workspace of the GUI is divided into two subspaces, which can be manipulated and viewed independently: the left half shows the architecture layout of the system, while the right half shows the run-time virtual machine. Figure 6.1 shows a screen shot of the basic run-time environment. In the following sections, we describe the functionality of each subspace individually.

6.3.1 Architecture View

In the architecture view, boxes represent the types of components that can be present in the run-time virtual machine, i.e., the instantiated architecture. Users can add components with a component tool and display their information by double-clicking on them. For each component, ADE shows at least three parameters which need to be specified by the user: the type the component represents, the number of components of that type present when the architecture is first instantiated, and the maximum number of components of that type which can be present simultaneously in the running virtual machine; see Figure 6.2. For user-defined components, which are extensions of the basic component type, as defined by APOC, ADE can display additional information about the component, such as the
Figure 6.1. ADE Interface showing a type-level description and a run-time view of the architecture

image taken by a robot’s camera as shown in Chapter 7.

Links in ADE are created through a link-creation tool that allows users to simultaneously specify links of each of the four available types. Each link can be configured individually as specified below. In the graphical interface, edges indicate the direction of the links in the architecture by the direction of the arrow. This is the same as the direction of information flow, with the exception of O-link edges, where the information flow is opposite to the direction of the arrow. By clicking on the arrow, information is be obtained about the types of links, e.g., A-link, that can be instantiated between the connecting components as well as the link parameters, e.g., delay or operator type for an A-link, in the run-time virtual machine.

All APOC links share the characteristics below:

- *A time delay.* The delay slot is always available for editing. It is used to specify the number of update cycles which are performed on a link from the time a piece of information enters the link to the time it is available at the other end. The default duration of an update cycle is 100ms, though this can be modified by the user. All links created simultaneously have the same delay.
Figure 6.2. APOC Component Specification Prompt. The three fields indicate the component class, the number of components present in the initial virtual machine, and the maximum number of components simultaneously present in the virtual machine.

- **Input and output ports.** Port specifications are provided through the X and Y input and output slots associated with each link type in the panel. In ADE ports are organized into ordered sets. The X value specifies the set of inputs (outputs) that the link connects to. An *UNASSIGNED* X value indicates that the link will attempt to connect to the set with the lowest index which has unused ports. The Y value specifies a unique port within the set determined by X to which the link is to be connected. An *UNASSIGNED* Y value indicates that the link can connect to any port within the respective set.

Figure 6.3. ADE Link Specification Prompt, showing customization options for each link type.
Each link type also requires a specific setup, which is activated once the particular link type is selected. The A-link definition requires the selection, and, perhaps, set-up of an operator which will act on the information passed through the link as specified in ADE. The operator can be specified by choosing from a list of available operators, both standard and user-defined. By default, the “identity operation” is selected.

P-link definition consists of choosing a default signal. The default signal is transmitted along the link upon link activation if the controlling component does not specify its own signal. Typically, this is set to “no operation.”

The O-link allows users to specify what elements of a component should be sent across an O-link observing that component. The elements which are eligible for observation in ADE must be declared as instances of a particular ADE-defined observable type. The user can use one of several defined classes, including APOCObservableInt, or by creating a new class which extends the APOCObservable interface. A menu presents the user with the available options. By declaring a variable as being of the observable type, it will automatically be included in the menu.

The C-link, in addition to delay and input and output port specifications, contains both an identifier for the type of component which can be instantiated through the link and one or more definitions for the types of links which can be instantiated through the C-link.

The interface also contains a “grouping” tool, which allows the user to collapse the view of several components to a single one. Each component retains its functionality, but the assemblage can be regarded by the user as a single functional unit.

In this section we described the basic characteristics of the elements which constitute ADE architectures. Having looked at the building blocks, we next describe the graphical environment in which the architecture of the agent is displayed.
6.3.2 Virtual Machine View

In the running virtual machine, boxes indicate actual computational components present in the instantiated architecture and edges represent instantiated APOC links. Multiple arrows can be present along each edge, indicating each type of instantiated link. The arrows are color-coded to differentiate among the four link types: A-links are represented by blue arrows, P-links by cyan, O-links by red, and C-links by yellow.

Users can insert components directly into the running virtual machine if the insertion operation does not violate the architectural restriction on the number of components which can be simultaneously present in the instantiated architecture. If a violation is detected, the instantiation operation fails.

Links can also be inserted in the running virtual machines. If a link is not available in the description of the architecture then the insertion operation fails.

In both cases, to prevent inconsistencies, users may modify the instantiated architecture in the running virtual machine only within the restrictions specified in the architecture layout. If new links need to be added which are not part of the architecture specification, then the architecture needs to be modified in the architecture layout space, before it can be reinstated.

6.3.3 Other Functionality

In addition to the architecture construction facilities described above, ADE provides tools which aid in visualizing and understanding the relationship between various parts of the architecture. Thus, an “architecture analysis mechanism” which aids in understanding structures resulting from interaction between the agent and its environment can be accessed by clicking on the “Abstraction” button in Figure 6.1. This computes the abstraction level of each element, as described in Section 5.5 and displays it next to each component. Another tool provided through the “Graph” button allows for the visualiza-
tion of variations of observable variables within a component over time (e.g., activation level and priority, see Chapter 7).

A grouping mechanism is also supported by ADE. Components can be selected and grouped together. A group can then be collapsed and represented as a single component in the architecture. Links drawn to and from the new component thereafter connect to all components in the group.

6.3.4 Operating Modes

ADE has three operating modes: an editing mode, a synchronous agent update mode, and an asynchronous agent update mode.

Edit Mode In edit mode, a user can modify both the architecture layout and the architecture present in the run-time virtual machine. Supported architecture layout operations are: adding/deleting a type of component, modifying the maximum number of allowable components of a type in the running virtual machine, adding a link between two component types, and deleting a link between two component types. In the running virtual machine, a user can add a component, delete a component, add a link between two existing components, and delete an existing link within the constraints imposed by the architecture layout.

Synchronous Mode The synchronous mode provides the means for a synchronous update of the agent architecture. In this mode, each component in the running virtual machine completes one update cycle and waits for the other components to complete their cycle. Specifically, the updates are performed asynchronously and upon completion a wait operation is performed on an external signal, which can be provided by the user or by the system, before the next update cycle is performed.
Asynchronous Mode  In asynchronous mode, all components and links update asynchronously based on their internal timing without synchronizing their state with other components. This is particularly interesting for distributed applications, where synchronization is not required and would result in a severe performance bottleneck. In some architectures, e.g., subsumption [27, 28] asynchronous update is even part of the architecture specification, and thus forces the agent designer to make no assumptions about the timely update of states and delivery of information. Note, however, that each node will still attempt to update at its update frequency if permitted by the operating system.¹

6.4 ADE Building Blocks III: The Supporting Environment

ADE’s supporting environment provides the infrastructure to distribute agent architectures and to operate virtual as well as robotic agents. It consists of a global registry, which dynamically keeps track of the elements of the distributed environment, and four types of servers: system servers, such as APOC virtual machines and graphical user interfaces, agent servers, which provide a “body description” for virtual agents or the interface to robots, and utility servers, which provide additional distributed services that are not part of the agent architecture.

6.4.1 Registry

The registry is a repository of available services as provided by the various servers. In particular, it provides updated information of the location of all participating APOC, GUI, agent, and utility servers, and maps APOC components that request a particular service to agent or utility servers, thus acting as a transaction broker between a client requiring a resource and available resources.

In ADE, these transactions can be of the following types:

¹On realtime operating systems, this update frequency can be guaranteed.
1. An APOC server requires another APOC server to instantiate a component of a given type (which resides on that server);\(^2\)

2. A graphical user interface requires a server whose information it needs to display;

3. A server requires a graphical user interface on which to display its information;

4. An instance of an APOC component which directly controls agent sensors and/or effectors requires an agent server on which to apply its function; and

5. An instance of an APOC component which represents/uses the functionality of a utility server requires a utility server from which to fetch its data.

In each of the above cases, the client contacts the registry and requests the desired resource either by specifying the type of resource required, if no specific instance is required or by identifying a specific resource, by virtue of its unique ID or location within ADE.

6.4.2 Servers

A server in ADE is a computational unit that represents a resource of the ADE system. Each ADE server is an independent computational resource that typically runs in its own operating system process and can be started independently. As a result, each ADE server has a \textit{main} method that sets up the service. After startup, each server first contacts the \textit{registry} and specifies the maximum number of client connections that it can support, as well as any additional restrictions regarding the connection, such as domain names which are allowed to connect to the system.

In the following sections we describe each of four server types--APOC server, GUI server, agent server, and utility server--and their functions within ADE.

APOC and GUI Servers Each APOC server is an independent entity, with capabilities for instantiating and deleting new components and links. APOC servers control their

\(^2\)Components are typically instantiated in the APOC virtual machine that keeps their JAVA class description. Multiple such descriptions in different virtual machines are possible, however, and allow for the implementation of load balancing mechanisms at the agent architecture level.
locally instantiated components and maintain connections to other APOC servers as well as to all available GUI servers in the ADE system in order to be able to notify the GUI servers whenever a new component is instantiated or an old one is deleted. Conversely, the GUI servers need to pass on user actions, such as adding a node to APOC servers. Therefore, each GUI server also maintain connections to all available APOC servers.

Upon start-up, an APOC server contacts the registry and requests connections to all GUI servers currently registered. Upon successful completion of the connection requests, direct two-way communication channels are established between an APOC server and each GUI server. This process is mirrored in the startup process of a GUI server, thus allowing new GUI or APOC servers to be added dynamically to an ADE system at runtime.

Agent Servers  
Agent servers provide access to the body of a virtual or robotic agent by establishing connections to its sensors and effectors. They are independent computational resources and can, therefore, be started independently. After startup, they automatically connect to the registry announcing their service and then wait for clients to connect. Users can define their own agent servers by extending the ADEServerImpl class provided with the ADE framework:

```java
public class UserServerImpl extends ADEServerImpl implements Serializable, ClientInterface
    
    public UserServerImpl (String registryHost, int registryPort,
                          String registryName, String myName) throws RemoteException
        super(registryIP, registryPort, registryName, myName);

    ...

    ...

    Any server could be integrated into the ADE system as long as it contacts the registry initially and provides a remote service, called requestConnection, through which it can
be contacted. By extending the *ADEServerImpl* class, however, the registry connection is handled by the ADE system automatically.

The default constructor of the *ADEServerImpl* class takes four parameters: the name of the computer on which the registry is running, the port on which the registry can be contacted (by default 1099, the JAVA RMI port), the name under which the ADE registry is known to the JAVA Naming service, and the name under which the agent server will register with the ADE registry.

Utility Servers  Utility servers provide a service which may be needed by one or more APOC components. Generally speaking these servers implement computationally expensive operations and are implemented as separate components of ADE as an additional aid to the distributed nature of the system. Utility servers follow the same startup process as agent servers, but may require additional connections to agent or utility servers in order to obtain the data they are supposed to process. These connections are established through the registry.

6.4.3 Example: Generic ADE Set-up for a Robotic Agent

To illustrate the different relationships among the four server types and the registry, we briefly sketch the generic setup for the control of a robotic agent in ADE.
Figure 6.4. The relationship among ADE components in a generic set-up for a robotic agent

Figure 6.4 shows the overall system configuration, where continuous lines represent components which are always present in an ADE environment, while dashed lines are components used only in particular instances such as the specific robot environment. Thus, a minimal system consists of the registry, one APOC server and one GUI server. Note that the GUI server is not strictly necessary. It is typically used to load and start the agent architecture and it is possible to configure ADE in such a way that no GUI servers are needed for its operation.

Utility servers may be added if computationally intensive functionality, e.g., image processing is required by the agent. Some of the utility servers may require other utility servers for their operation, e.g., a utility server performing 3D object recognition might require a utility server that provides basic image analysis services.

Additional agent servers may also be added to the system before startup or at runtime, e.g., to provide a simulation environment for the robot or to provide a “simulated body model” for the robot if it is also part of a virtual environment. For example, ADE could be used to implement a combined real-virtual environment [18].

This example is only one of several possible configurations for robotic setups. Dif-
ferent agent servers could be used, for example, for different sensors and effectors of the robot, or for different robots in a multi-robot setting. Depending on the task requirements, ADE can be dynamically adapted to work with any configuration of servers.

6.4.4 Client-Server Communication

Once a client-server connection has been established, e.g., between an APOC virtual machine and an agent server after mediation by the registry, the client is responsible for the maintenance of that connection. Maintenance for client-server connections consists of the client calling a server function \texttt{(updateConnection)} at regular time intervals to ensure that the connection is alive. The length of the interval is set up at connection time, but can be modified at any point by the client. This mechanism effectively ensures the integrity of the ADE system and allows it to react to possible failures. It also supports interactions in real-time domains (such as in the robot setup mentioned above).

6.4.5 ADE Configuration File

In order to simplify the startup of an ADE system, which may be quite complex and involve numerous different hosts, on which services need to be started, ADE uses a global configuration file. In a typical setting, this configuration file is run in one GUI server, which needs to be started manually. This server then bootstraps the rest of the ADE system from a single host.\footnote{The registry is started separately by the user, not through the bootstrapping process.} In the bootstrapping process, the GUI server reads the configuration file and uses a secure shell to open connections to each host in the “hosts list”. Once a connection is established to a remote host, the “SSH\_COMMAND” is executed on the remote host to start an APOC server.

The following parameters can be set in the configuration file:

1. REGISTRY: a duple containing the name of the machine on which the ADE registry is running and the name under which it is registered with the JAVA Naming service.
2. SSH: the secure shell program used to start remote servers.

3. SSH_COMMAND: the command to be executed on the remote host in order to start a remote server.

4. HOSTS: the computers that are available to host services of the ADE system.

5. COMPONENTDIRS: the directories containing APOC component definitions. In addition to the default components included in ADE, users can add their own component definitions, which can then be used within the whole ADE system.

6. OPERATORDIRS: the directories containing operator definitions for A-links. In addition to the default operators included in ADE, users can add their own operators, which can then be used within the whole ADE system.

7. INPUTFILE: an agent definition file to be loaded at start-time.

Architecture Definition Files  As mentioned in Section 6.3, ADE uses XML files to save and load architectures. A generic dtd file is included in the system, which specifies the format of the architecture definition files [42].

Each non-empty file contains at least one type which can be present in the system. Optional contents of the file are type-level link description, instantiated tokens of each type, links between these tokens, and the name of a system permission file.

The System Permissions File  In constructing distributed architectures, it is often important to ensure that certain components are instantiated in a unique location. For example a camera control component should be instantiated on the computer on which the camera is mounted. To achieve this mapping of components onto computers, ADE provides the option of specifying in the system permissions file on what hosts individual components can be instantiated. The system permissions file can also be used to impose load distribution constraints on an ADE system.

General functionality  The File menu item provides the user with options of saving an architecture in the current state and loading an architecture from a previously saved file.
The *Edit* menu item allows users to change several system attributes. ADE searches for modules which can be inserted into an architecture in a set of user specified directories. The *Edit* menu provides users with means of viewing, adding, and deleting directories from the ADE search path. It also provides a means of customizing the behavior of the system by specifying what functions should appear in the interface. For example, in an academic environment, an instructor may choose to make the asynchronous run mode unavailable to students for a particular project.

The *View* menu provides tools for customizing the view of the architecture. For example, units, links, or particular types of links, such as A-links can be hidden to allow the user to inspect the structure of individual parts of the architecture.

The *Mode* menu item provides the user with a choice of three modes: (1) Edit mode, in which architectures and architecture descriptions can be modified “off-line” by the user, (2) Synchronous mode, in which every instance of a type updates exactly once and then waits for all other nodes to finish their update, and (3) Asynchronous mode, in which all instances update independently.

*Network Setup* provides a variety of functions related to the distributed nature of the environment. *Show available hosts* displays the set of computers on which computation can be distributed. *Add Host* and *Delete Hosts* modify that set, while *Reset Hosts* empties the set. Finally the *Registry Setup* function allows users to specify the computer which hosts the registry and the name under which the registry is running.

The *Help* menu displays information about various parts of the environment. This includes javadoc files for ADE components and links, as well as for other utilities, such as various A-link operators.

Having described the building blocks of ADE, we now discuss in the facilities provided by ADE for the design of agent architectures.
6.4.6 Optimizations

In order to make the system as efficient as possible, several optimizations were used in the writing of the ADE toolkit. The first optimization focused on optimizing communication between architectural components. If two components reside on different machines, they communicate through JAVA RMI calls. These calls, however, can quickly become a bottleneck for system performance. Therefore, ADE components were given the capability to check if their incoming and outgoing links reside on the same APOC server. Similarly, links were given the capability to check if their start and end components reside in the same APOC server. If a link and component reside in the same APOC server, communication between the two is then performed through regular function calls.

A second optimization was the decoupling of graphical information, such as component location in the GUI, from architectural components. This allows for a reduction in network traffic and therefore improved performance.

In this chapter we have looked at the relationship between APOC and ADE, illustrating how theoretical foundation of the development environment translated to desirable practical features. We have also presented the structure of ADE. In the next chapter we focus on ways in which the features of ADE can be used to help the agent designer.
CHAPTER 7

USES OF ADE

In this chapter we explore ADE and the facilities it provides to agent developers. These facilities range from the ability to describe architectures at various levels of detail, to providing tools for the addition of graphical elements to components in order to visualize component functionality. We conclude by providing several examples of the use of ADE from the experiments of Chapter 9.

7.1 ADE as a Tool for Designing, Testing, and Running Agent Architectures

ADE serves at three-fold role in the development process of agent architectures: (1) as a design tool for developing agent architecture layouts and their implementations, (2) as a platform for run-time control of agents, and (3) as a tool for testing agent architectures. Although these three roles typically overlap in the practice of developing agent systems, we address each role separately to emphasize different characteristics of ADE.

7.1.1 ADE as Design Tool

ADE is built on the premise that a graphical representation of an agent architecture that can be viewed at different levels of detail is crucial in the design process of agent architectures. The graphical interface allows users to add components and links, specify and modify their properties, and arrange them in their preferred layout. Since architecture descriptions are saved in XML format to a file, they can be also viewed and edited using
standard XML authoring tools. Furthermore, they can be inserted into existing architectures, thus allowing for efficient reuse of common subarchitectures.

By using JAVA as the implementing language, ADE is platform independent, i.e., it runs on any operating system for which JAVA virtual machines exist and thus, provides essentially the same environment on different machines. More importantly, once components are defined and compiled on a particular system, they are available on all systems with JAVA support. This is particularly useful for distributed design setups that involve multiple computers with different operating systems, all of which can seamlessly interact in ADE.

As mentioned in the Introduction, one of ADE’s advantages over other agent development environments is that it is not based on a particular architecture paradigm, but rather that it is capable of implementing in a unified way any agent architecture, e.g., cognitive architectures such as SOAR [61, 90], ACT-R [7], as well as behavior-based architectures such as subsumption [27], motor schemas [12], situated automata [54]. It therefore possible to compare the tradeoffs of different architectures with respect to some particular task. For example, two different architectures implementing a target-finding task for a robot can be compared with respect to the number of components used or the total time required for achieving a goal.

ADE also allows for the display and design of agent architectures at different levels of abstraction, depending on the complexity of the update functions in employed APOC components and the topology of the connections among them. It is, for example, possible to use Boolean update functions in components to implement logic circuits, such as gates, inverters, and flip-flops. A network of such components could then, for example, model the low-level architecture of an embedded processor or robot controller. By the same token, it is possible to use APOC components that implement very high-level functionality such as condition-action rule-interpreter, a planner, a reasoning engine, and a search
algorithm. In the latter case, ADE will only be able to display a high-level view of the architecture, while the details of how the rule-interpreter, planner, etc. work are hidden in the associated process of the component implementing them.

It is also possible to use the associated process of an APOC component to run algorithms that are not defined and implemented within the ADE framework. Input to and output from these processes is effected through A-links within ADE via their input and output streams. Other components can communicate with such “external processes” in a transparent way, i.e., without having to know that they are external to ADE. Again, the details of the implementation of such processes is necessarily hidden and cannot be visualized in ADE.

7.1.2 ADE as Run-Time Tool

A running architecture, i.e., the architecture instantiated in the APOC virtual machine can be inspected and modified at any given time through the graphical user interface. Double-clicking on a component, for example, reveals information about the component. The information displayed is provided by the extendComponent function, which is user-definable in classes derived from APOCNode.

Figure 7.1. Node information display for a robot representation. Buttons are available to select the display of various robot sensors

The example in Figure 7.1 illustrates the information which can be viewed for a component representing a robot body description. Clicking on any of the buttons brings up additional windows displaying internal information of the robot. For example, clicking
the “Camera” button will bring up a panel displaying the current image of the camera and the processed image, as seen in Figure 7.2.

![Camera panel](image)

Figure 7.2. Camera panel displaying the original picture (left) and the post-processed image, the contour of the ball (right)

Similarly, general information about links can be obtained by double-clicking on the link’s arrow. Figure 7.3(a), for example, shows the information displayed by O-links. Clicking on the Link Info button, results in a display of more detailed information about the actual data passed through the links. In Figure 7.3(b), the fields being observed are displayed in the first line (act and pri), with the data currently in the link displayed below. Specifically, an O-link of delay 2 is displayed after two update cycles. Thus, there is information in the first two slots of the link, but no information is available yet for retrieval from the link. This is indicated in the figure by the top slot being empty. It can be seen that the activation value of the observed node changed from 0.27688 to 0.28092 in between updates of the O-link, while the priority remained constant.
Users have the option of altering values displayed in text fields, and can thus directly influence the behavior of the system to test various aspects of an architecture. It is also possible to instantiate components and links or delete them in the running architecture, as long as the operation does not conflict with the resource limits in the architecture layout.

7.1.3 ADE as Test Tool

The synchronized updating mode of components and links in ADE is particularly useful for testing purposes, as an architecture can be “frozen” at any point in the life-time of an agent and inspected. Users control when the next update occurs, and can inspect and modify architectural parameters between updates. It is even possible to change the architectural layout and continue the update process with a modified architecture.

ADE provides several tools for tracking, displaying, and analyzing information and information flow in the architecture. The Graph tool, for example, can be used to track the values of any variable of type \textit{APOCObservable} in a component over time. Graphs of the temporal evolution of these variables can then be saved or printed. Figure 7.4 shows an example of the graph tool, tracking the activation value of an APOC component.
Other tools, such as the *Abstraction* tool, for example, allow users to automatically group components according to their level of abstraction as determined by their C-link structure. This is intended to help users in isolating structures that might have formed in a self-modifying architecture over time for later reuse.

Most importantly, it is possible in ADE to insert inspection components into an architecture without influencing the processing of existing components. These inspection components can observe any part of the architecture via O-links and report the data to the user, either through the graphical interface, or by saving the data to a file.

Testing of architectures in ADE can also make use of the fact that communication among components is done exclusively through links. As already mentioned, the same architecture can be run in both a robotic agent and a simulated virtual agent as long as agent servers exist that allow the architecture to connect to the respective “body representations” of the agents, i.e., to the physical robot or the simulation environment. This has the advantage that an architecture can be tested in a simulated environment before it is run on a physical robot. The concept can be taken further by connecting a physical robot to the simulated environment and observing any divergent actions between the robot and the simulated agent previously controlled by the same architecture. For example, due to delays in command execution with a robotic agent, since effectors will not move instan-
taneously after a command is issued.

In the next section we present a preview of the use of ADE in a robotic context, with more extensive experiments to follow in Chapter 9.

7.2 ADE Robotic Example

The project described in this section was a joint venture with the Mechanical Engineering department. Thus, we focused our work on developing a robust interface to the robot assembled on the mechanical engineering side of the project. The robot was a hexapod robot, making it an interesting subject for both mechanical engineering, e.g., for the implementation of low-level servo control for each leg, and computer science curricula, e.g., for the implementation of an ant-like foraging behavior.

A hexapod is much better suited for a mechanical engineering - computer science collaboration than other robotic platforms. For example, the LEGO mindstorm kits are too limited in capacity for any interesting project, the ActivMedia Pioneer robots are too complicated for courses where robotics is not the main focus, preassembled robots are of little use on mechanical design course, and robots that have to be built from scratch are similarly not well-suited for a computer science course, which, for example focusses on behavior-based techniques or higher level control. With this in mind, the hexapod robot was designed to be:

1. Complex enough from a mechanical standpoint, so that mechanical engineering students could experiment and test various aspects of design and control.

2. Complex enough that artificial intelligence and behavior-based robotics techniques could be used in applications using the robot

3. Programmable in a general purpose language such that high-level programming control of sensors and effectors is achievable

4. Used by multiple students in an interactive manner as part of a course (i.e., the robot should be accessible in a controlled, distributed manner through a wireless link).
It is clear in this context that ADE can be used to satisfy the last two requirements listed above. With these desiderata, the same robotic platform can be used in courses in two or more departments, strengthening inter-disciplinary research and learning.

The section is organized as follows: we first give a brief overview of the system design. We then show how the robot can be controlled from within a JAVA program in the context of ADE. We conclude with a brief example of a typical setup of the system.

7.3 System Overview

The overall system consists of one or more host computers and the hexapod, which is connected to the serial port of a dedicated server computer through a wireless airlink. A schematic of a typical setup is given in Fig. 7.5. Roughly, the system can be categorized in terms of software and hardware systems, where the software system mainly involves the coding and implementation of the robot’s control scheme, while the hardware system consists of the physical robot, sensors, actuators and other computer hardware required to interact with the robot. The software aspect of the overall system, relevant to ADE, is discussed in the next sections.

1The way in which that connection is established is described in Section 7.4).
Figure 7.5. **Client request for connection to a server**: 1. The client contacts the AgeSRegistry (running on some host) and requests a connection to a server. 2. The AgeSRegistry locates the server and relays the request. 3. The server returns a remote “stub” object. 4. The AgeSRegistry relays the remote object to the client. 5. The client and the server maintain a direct connection.

The control strategy used in this project is separated into high-level and low-level control. The interface between the two is a set of motion commands which are called by the high-level control and interpreted and converted into motor commands by the low-level control. This strategy allows for the system to be used independently by students with training in different fields without rendering the overall system inoperable.

**High-Level Control**  The process of writing control code for robotic agents can be difficult, especially if the programmer is unfamiliar with the programming language of the robot or the process of developing robot controls. Both cases arise in classroom environments, where students encounter and apply concepts for the first time. In order to simplify the process of writing control code, an interface was developed between the hexapod and ADE.

ADE was particularly well-suited for the development of the high-level interface to the robot due to its facilities for developing control in a high-level language (JAVA), distributing computation among a potentially large number of computers, and the potential
for the development of graphical tools to be used in testing, viewing, and modifying control code parameters. The structure of the ADE-based system for this experiment and a typical start-up sequence are illustrated in Figure 7.5.

When servers are used to provide access to a robot, they provide an interface between that robot and its environment (i.e., operating system, programming language) and the students. In building a server, instructors define JAVA functions, which make direct calls to the native functions of the robot. In order to use the robot, the students then write control code which contacts the AgeSRegistry, requests the required server, and calls the educator-defined JAVA functions on the returned object.

A detailed example of the server implemented for the hexapod is presented in Section 7.4.

Computer Hardware Computation time is a critical resource in real-time applications, especially when visual processing is involved. Two features of ADE which are a result of its JAVA implementation proved beneficial in constructing this system:

- Communication facilities. JAVA provides Remote Method Invocation (RMI) tools which make remote calls look and feel like local procedure calls.

- Platform independence. If a JAVA virtual machine is running on a computer, the hardware/operating system of that computer is not a barrier to communication and/or program execution. Several tests have been run in which Solaris, Windows, and Linux computers were used together in the same system.

In creating the hexapod system, initial development and testing was performed on a DELL computer running RedHat 9. Once the basic functionality was in place, the code was then seamlessly migrated to the Windows computer which was connected to the hexapod.

Two computers were involved in our set-up. The hexapod was initially connected via cable to the serial port on the first computer. The program which allows the hexapod to communicate through a wireless connection was downloaded to the Isopod and started
executing upon download completion. This process was repeated whenever changes were made to the hexapod program. Several tests were otherwise run without a redownload.

A second computer, with a wireless transmitter-receiver connected to its serial port was used to start both the registry and the server designed to connect with the hexapod. The hexapod communicates at 115200 bps, using eight data bits, one stop bit and no parity checking. The hexapod server configured the serial port to the required parameters and checked the serial port (through JAVA routines) for incoming information. Two-way communication was established upon receipt of a status information packet from the hexapod.

7.4 The ADE Environment for High-Level Control

In this section we describe the server for the hexapod robot, its functionality, and how that functionality is achieved practically within the ADE JAVA environment.

Since the hexapod communicates through wireless RS-232, the hexapod server has to run on a computer with a wireless device connected to its serial port. Once that port is known, the server attempts to initialize the port and establish communication channels with the robot as shown:

```java
try {
    serialPort.setSerialPortParams(115200, SerialPort.DATABITS_8, SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
} catch (UnsupportedCommOperationException e) {
    System.err.println("Could not set port parameters – failure");
}
try {
    //write the message to the output
    outputStream = serialPort.getOutputStream();
    tempInputStream = serialPort.getInputStream();
    inputStream = new BufferedInputStream(tempInputStream);
    serialPort.notifyOnDataAvailable(true);
} catch (IOException e) {
    System.err.println("Could not get streams – failure");
}
```
The robot is programmed in its native language to receive simple commands and execute them. In our case, these commands consist of a fixed-length set of bytes, prefixed with an instruction identifier. For example, start bytes of ’11’ indicate motion and start bytes of ’33’ indicate camera operation. Commands are sent to the robot in ASCII characters, so, for ease of use, we provide the conversion in the server, allowing the students to use the more natural numeric format when specifying distances and angles as seen below:

```java
public byte[] getCameraPicture(int direction, int angle) {
    command = new byte[]{'3','3', (byte)(direction/100+'0'), (byte)((direction/10)%10 + '0'),(byte)(direction%10 + '0'),(byte)(angle/100+'0'), (byte)((angle/10)%10 + '0'),(byte)(angle%10 + '0'),'0','0','0'};
    return procJPEG;
}
```

It should be noted that in this case, due to the connection speed, the reading of the picture is done asynchronously and the previously read picture is returned upon invoking `getCameraPicture`.

For motion commands, a limit of 100 inches was imposed on the distance the robot could travel in as a result of a single command. Limits are also imposed by the physical design of the robot on the step size. The function provided in the server is `sendCommand` with six arguments: the type of motion, e.g., straight or curved, a correction variable indicating whether corrections should be made to the proprioceptive coordinate system of the robot during the motion, the direction, either forward or backward, the distance to travel, the stepsize and the final orientation of the robot. The function and its arguments can be seen below.
public void sendMotionCommand(byte motion, byte correction, byte direction, byte finalpos, byte stepsize, byte angle) throws RemoteException {
    command = new byte[]{(byte)'1', (byte)'1', (byte)((motion+'0')), (byte)((correction+'0')), (byte)((direction+'0')), (byte)((finalpos/10 + '0')), (byte)((finalpos%10 + '0')), (byte)((stepsize/10 + '0')), (byte)((stepsize%10+’0’)), (byte)((optional/16+’0’)), (byte)((optional%16+’0’)};
    try {
        outputStream.write(message);
    } catch (IOException ioe) {
    }
}

Other commands can be built on top of sendMotionCommand, by fixing certain parameters as shown:

public void moveForwardWithoutCorrection(byte distance) {
    sendMotionCommand(0,0,0,0,distance,1,0);
}

By providing similar functions for the basic operations of a robot, such as turning and moving backwards, the students’ focus can shift from learning the details of the robot to understanding and applying the concepts taught in the course.

A second important benefit that arises from the use of the ADE utility server is a significant decrease in the computational load placed on the hexapod. The server makes hexapod data available to remote computers, potentially several orders of magnitude more powerful than the processor present on the hexapod. By performing the computation relevant to the hexapod on these machines, users can attempt more complex tasks than possible on the hexapod’s own processor.

7.5 Example of a Typical Setup

Here we briefly describe how a typical setup of the system would proceed as follows, assuming that the communication program has already been downloaded to the hexapod:
first, a registry and server have to be started on a remote computer. The startup of the registry and server is a straight forward process: \texttt{java com/AgeSRegistryImpl} starts the registry and \texttt{java com/hexapod/HexapodServerImpl <\texttt{registry\_hostname}>} starts the hexapod server and uses \texttt{AgeS} mechanisms to automatically connect the server to the registry.

The control program connects to the registry and requests a connection to the hexapod server. This is a standard procedure and code can easily be provided to the students. The code is shown below.

```java
HexapodServer robot;
ConnectionUpdater cu;
String serverObjectName = "rmi://"+<\texttt{registry\_hostname}>+"/AgeSRegistry";
AgeSRegistry talk = (AgeSRegistry)Naming.lookup(serverObjectName);
if ((\texttt{robot}=(HexapodServer)\texttt{talk.requestConnection("admin","admin","Hexapod"))} != null) {
    \texttt{cu = new ConnectionUpdater(this,\texttt{robot},3500,"\texttt{MotionTracker}");} else {
    System.out.println("Server returned no agent instance!");
    this.robot = null;
}
```

After the connection to the server is established, students can simply call procedures defined in the hexapod server as shown:

```java
\texttt{robot.getCameraPicture();}
\texttt{robot.moveForwardWithoutCorrection(20);}
\texttt{robot.turnRightWithCorrection(45);}
```

We tested the system, for example, with a control program that uses several hexapod functions in a cyclic manner:

- Image retrieval. This function gets an image from the hexapod camera. The image is subsequently displayed on the screen by the control program.
- Hexapod motor commands. A variety of motor commands were implemented, which could relay motion parameters to the robot. For example, a forward motion of 20 inches can be achieved by calling the \texttt{moveForwardWithoutCorrection((byte)20)}.  

Camera motor commands. These commands were used to determine the direction in which the robot camera was pointed. An example command given to the robot was move-to-45-degrees in a coordinate system in which the rightmost position was considered to be zero.

The communication time for motion commands was negligible and the hexapod responded promptly to the commands issued by the remote computer. The communication bottleneck consisted of the image transfer, which, at 115200 bps took several seconds to complete.

In this chapter we have described the ADE toolkit and provided a brief glance into its utility in interfacing with robotic systems. The next chapter focuses in experimental validation of the APOC/ADE approach to designing agent architectures.
The driving force behind the development of both APOC and ADE is the creation of an environment well-suited to the development of complex agents, especially robotic ones. In this chapter we investigate the requirements imposed by the design, implementation, and testing of complex robotic agents and show how ADE satisfies those requirements. As the research was geared towards the creation of a robotic waiter, the requirements will be illustrated through examples from the “robowaiter” domain.

A complex robotic agent needs to perform tasks of varying complexity in the robowaiter domain: from simple effector control, such as navigation or handing out a drink, to fast situation evaluation (e.g., can I afford to take another order, or do I need to complete the ones I currently have), to planning, for example, the sequence in which cleaning a spill, taking a new order, and fulfilling an old order will be performed.

The variety of these tasks and the scope covered by them create several problems for the agent designer. We discuss some of these problems in the following section.

8.1 Complex Robotic Agent Problems Stemming from Multitude of Tasks

The development of a complex robotic agent consists of at least the following stages:

- Analysis of the required functionality of the agent
- Breakdown of the functionality into functional components
Design of the architecture, in terms of individual component functionality and connections among the functional components

Implementation of the architecture

Testing and fine-tuning architectural parameters

Any of the above steps may be repeated several times before a suitable architecture is found. In the following sections we discuss each development stage and, where applicable, demonstrate the usefulness of ADE with respect to that stage.

8.1.1 Functional Analysis of the Architecture

Functional analysis is often a speculative process, based on observable behaviors of the system, such as moving towards a door or carrying out a conversation. In the case of the robowaiter scenario, a process of successive top-down decomposition of behaviors can be used to obtain an initial architecture.

Thus, the initial task of waitering can be viewed to consist of such tasks as: greeting a customer, taking an order, fulfilling an order, cleaning a spill, and dealing with a drunk customer. Each of these tasks can be further broken down into simpler ones.

Greeting a customer can be viewed as a task whose purpose is to identify new customers and enter into a conversation with them. A schematic of the behavior can be seen in Figure 8.1. The greeting behavior can only be initiated if a new customer has been identified. In an architectural description, this would be specified by a behavioral precondition, shown on the left of the behavior as a hexagon. This precondition can be satisfied as the postcondition of a “Find New Customer” behavior, which occasionally points the camera towards the entrance and attempts to identify any person who is either entering or waiting by the entrance.
Usually, greeting a customer marks the beginning of a longer interaction which will include taking an order from that customer, fulfilling that order, and taking a payment. In order to identify the customer and keep track of future interactions, e.g., remembering the order placed by the customer, the greeting behavior creates a customer model. This model contains customer specific information:

- Identifying information, e.g., a facial picture, name, and the table at which the customer was seated.
- The order placed by the customer, which can be a link to a separate data structure.
- A time stamp of the last interaction with the customer.
- Additional information, e.g., state of inebriation, possibly determined by the customer’s rate of speech.

Once a new customer has been identified, interaction with the customer cannot be initiated unless the robot is in close proximity of the customer. Therefore, the second part of the greeting behavior is a “Go to Patron” behavior. This behavior can also be broken down into simpler behaviors, as seen in Figure 8.2
Figure 8.2. The schematic of the “Go to Patron” behavior for a robot waiter.

The “Go To Patron” behavior suggests the necessity of a planning capability in the robot architecture. It should be noted that such a capability may not be necessary in sparse environments in favor of simpler techniques (as recent experiments in our lab have indicated [59]).

Finally, the greeting behavior requires interaction with the customer, as represented by the “Greet Patron” behavior. Interaction with the customer determines several capabilities of the architecture: speech processing, a speech generation, parsing, and a syntactic analysis. These functions are discussed in more detail in Section 8.2. Once the greeting is completed, a “Greeted Customer” goal, represented as an octagon, is achieved.

The above example of top-down decomposition for the “Greet Customer” behavior is one example of functional analysis of the architecture, with more examples to be provided in Section 8.2 below. The final stages of the decomposition process denote simple behaviors, which already could be considered for implementation as separate components in the architecture. Thus, functional analysis transitions seamlessly to the breakdown of functionality into components.

Top-down decomposition graphs can provide a first look at data flow within the architecture and preliminary decisions regarding the type of link to be used can be made here.
Thus, even though, due to the nature of the initial design, APOC and ADE do not play major roles in the original development process, the process of optimizing an architecture according the ADE properties can begin at this stage. In subsequent iterations through the cycle, however, ADE play much increased roles.

After a first test of the architecture is performed, a functional analysis of the system may determine that additional functionality is required in the system. In this case the analysis process needs to start with an analysis of the existing architecture and a determination whether the additional functionality should be achieved through modifications of existing components (e.g., by optimizing an aspect of a visual processing component in order to improve the real-time characteristics of the system). Sometimes, however, new components need to be added to the architecture, e.g., if sonar-based navigation is not accurate enough due to environmental noise, a laser-based navigation component may be added.

8.1.2 Breakdown of Functionality into Components

The original breakdown of functionality into components can be a very flexible process due to the use of ADE. System developers have several options regarding the detail expressed in the architecture, such as deciding to use a single component for a rule-based system or implement it across several ADE components. They can also decide which parts of the system need to be implemented anew and which existing components, perhaps developed for other systems, can be re-used. This type of flexibility allow system designers to choose the optimal strategy for each agent based on the task of the agent, computational resources, and physical characteristics of the robot.

The creation of this model is now a basic operation and can be represented as a component in the final architecture.

In some architectural paradigms (e.g., subsumption [27]), adding components to a
completed architectures can be a complex process. In ADE, however, the addition of a new component is facilitated by several tools, discussed in the next section.

8.1.3 Architecture Design

Architectural design decisions are made differently for the original design of a system than for subsequent alterations to the system. Thus, we will explore each decision making process separately in relation to ADE.

Original Design The original architecture design is interweaved with the processes of functional analysis and component breakdown. Many interconnections will be inherently defined in those processes. There are two cases where decisions still have to be made. The first regards the choice of using an A-link vs. an O-link. In the original design, considerations needs to be given to whether the data being passed through the link needs to undergo any change between sender and recipient and whether computation will be more intensive on the sender or the receiver side (as A-links reside on the sender machine, whereas O-links reside on the receiver machine).

The second decision to be made is whether computational elements should be re-used or dedicated components are better suited to the task.

Architecture Redesign It is essential for an efficient design process and a fast turn-around time that redesigns of an architecture not require a complete redesign of the architecture.

Two features of the ADE work towards achieving the above requirement. The first is the encapsulation of functionality within components which communicate via links. With input and output ports being grouped in JAVA Vector classes, as shown in Chapter 6, components can be designed to handle an arbitrary number of inputs and outputs. Thus,
a new component can be connected to an existing one by adding a link connecting the two components.

The second feature is the presence of O-links. A component may be added to the architecture which requires information from other components in the architecture which were not designed such that they would be able to send information to the new component. In this situation, the new component may simply connect through O-links to existing components and extract, unobtrusively, the required information. This facility also allows for the addition of higher-level functionality to simpler systems. Section 9.2 illustrates how the latter process can be achieved in ADE.

8.1.4 Implementation of the Architecture

It is unreasonable to assume that every component of a complex robotic system will be created “from scratch” for that system. Adopting such an approach would not only require an immense time commitment, but would also fail to take advantage of existing software packages which may already provide the required functionality. The ADE framework provides facilities for “wrapping” components around external software packages and therefore embedding these packages into an ADE architecture. An example of this facility, used to embed the CMU Sphinx II software package into a robotic architecture, is provided in Section 9.4.

The variety in scope among various components, together with the use of external software packages create the potential for a different problem, that of inter-component communication. For example, effector control, such as the specification of wheel velocities, is sub-symbolic, while planners generally work on a symbolic level. Still, the output of the planner needs to reach the motor controller in a form which the controller can use. ADE provides the architecture designer with two opportunities for performing the required conversions. The first conversion can be performed by sending the information
from the planner through an A-link and performing the conversion in the link itself by defining a suitable operator. If other data required for the conversion, e.g., the current position and orientation of the robot, is not be known on the link, then a new APOC component can be connected to both the planner and the current map. The new component is then able to perform the required conversion and transfer the data to the motor controllers.

A third problem which arises with large systems is that of computation time. Some system components, such as those performing visual processing, are by themselves computationally intensive. Sound processing, deliberative components, and other capabilities further add to the system’s demands for computation time. ADE provides a solution to this problem by not only providing a distributed environment for the architecture, but also allowing the agent designer to specify where certain types of computation should take place. Sections 9.3 and 9.4 show the benefits derived from this ADE capability.

8.1.5 Testing and Fine-Tuning

A different type of problem that shows up in robotic agents is that of synchronization between computation and effectors. In simulated agents, motor actions, such as turning the vision sensors fifteen degrees to the left, can happen instantaneously. However, in a robotic agent, a delay of varying length may be present between the time the command is issued and the time the camera reaches its target location. Similar delays occur in other effectors such as wheels, grippers, and robotic arms. These delays can give rise to unexpected interactions among architectural components, with effects ranging from mild inefficiencies, such as oscillatory camera motion when moving in order to focus on a fixed object, to system failure, such as losing a tracked object from camera view, to fatal failure, such as not being able to stop the robot before it hits a wall.

As the next section shows, learning mechanisms are easily embedded in ADE. However, a robotic agent should start out with a working system which learning can then
improve in order to make it best suited for its environment. This may require the agent
designer to set several parameters, such as a time delay between sending a command to
the motors and the time when the system assumes the robot is in motion, or a specific
inertial setting for a camera motion, if motion integration is used.

The JAVA foundation of ADE means that graphical displays can easily be constructed
for any component. These displays can contain control mechanisms, such as slide bars,
for quantities that require calibration. Through the use of on-line parameter variation
tools, the ‘base’ values for a variables in a component, such as the above mentioned time
delays, can easily be determined.

8.1.6 Learning

Consider the case of a person who moves to a new job, in a new city. The first
few days in the new city are stressful; almost everything is new: the streets, the people,
the working environment. However, the person has a basic set of skills which allow
for the management of this new environment: map reading for navigating the streets,
conversational skills for people interaction, job skills for the new working environment.
Thus, even though overall performance - measured objectively, e.g. in terms of time
it takes to get to work or amount of work done, or subjectively, e.g. in terms of ease
of interaction - may be less than optimal, the person is functional. Moreover, as time
goes on, the person becomes more familiar with the environment and performance in all
categories usually increases.

A complex robotic agent should be able to mirror - to a certain extent - the perfor-
mance of a person placed in a new environment: it should have a basic set of skills which
allow it to perform its duties in a new environment, but its performance should increase as
the robot’s familiarity with the environment increases. To allow for such performance in-
creases, certain modifications needs to take place at the level of the architecture. An agent
needs to have the opportunity to explore and learn, for example, the behavior-selection strategy which works best for navigation in crowded environments, or the association between tone of voice and the current mood of the person with whom the agent is interacting.

In Chapter 4 we discussed several possibilities of learning in the APOC framework (including Hebbian, Q-learning, and architecture-level learning). In Section 9.2 we show how association and architecture-level learning can be implemented in ADE. The first experiment illustrates learning of the relevant sensors for a task, while the third describes a situation where the robot can learn a behavior-selection strategy switch. In both cases the switch takes place at the architecture level, with components and links being created to achieve the desired effect, i.e., the two systems are ‘dynamic’ in the sense of Chapter 4. The second experiment shows how a robot can learn to modify the relation between visual information and wheel speeds. Sonar values and visual information are combined to determine the direction in which the robot is going to move. In this experiment, as the robot interacts with the environment, the robot learns to increase the influence of visual information on robot orientation. Learning here takes place within the architecture, i.e., in an adaptive manner, as described in Chapter 4.

An aspect related to learning is that of knowledge acquisition. Among the things which can be learned are locations of objects and people, functions of objects, relationships among objects. While knowledge acquisition can take place within a memory component, ADE offers the agent designer the possibility of having explicit representations of all these types of knowledge in the architecture. The advantages to this approach are twofold. First, the agent designer can visually monitor the knowledge base of the system as the robot interacts with its environment. Second, by not being hidden within a component, memory elements can be accessed directly by any other component which needs access to the data in that memory element. For example, if the robot needs to find a refill-
ing station for its tray, a specialized component in the architecture can determine which memory elements represent filling stations and select as a target the location of the closest station. In Section 9.4 the robot learns relations among objects as it parses a sentence.

8.2 Robowaiter

The robowaiter project requires functionality at all levels of complexity, from basic navigation with obstacle avoidance, to visual processing, planning, and, perhaps, emotion generation/evaluation. In this section we look at several subsystems of the architecture, together with software that may be used for their implementation. We start by presenting an overall view of the robowaiter architecture.

8.2.1 Overall Architecture

With all the elements described above, in this section we give a high-level schematic for the overall robowaiter architecture, which can be seen in Figure 8.3.

Figure 8.3. A high-level view of the architecture for a robot waiter.
The figure describes the relationships among the functional modules described above. In the following sections we describe each of the high level systems and their functionality. Then, we look at the robowaiter architecture in more detail, providing descriptions for some of the main subsystems in the architecture, which use functionality from several of the modules of Figure 8.3.

8.2.2 Navigation and Mapping

In its simplest form, navigation is achieved through a reactive mechanism: the robot moves randomly through its environment while attempting to avoid collisions with obstacles. A step above that comes goal directed motion. Several experiments which show goal directed motion in varying conditions are presented in Sections 9.2 and 9.2. These experiments, together with GLUE-based navigation will form the basis of the navigation system for the robowaiter.

8.2.3 Visual Processing

Several experiments were performed using visual processing together with robot and camera motion. These experiments, also described in Chapter 9 included simple goal directed motion where the goal was identified in a visual manner in Section 9.2, object tracking using camera motion with both a stationary and a moving robot in Section 9.2, and reference resolution in a multi-object environment in Section 9.4. The vision code used in these experiments was written in JAVA and its functionality was limited to blob and motion detection.

For the robowaiter architecture, we are exploring using C/C++ code to perform the visual computation and encapsulating that code in an ADE component. One option under investigation is the use of the Intel Open Source Computer Vision Library [78], which is mainly aimed at real time computer vision. The library was designed for use in object identification, face recognition, gesture recognition and motion tracking.
8.2.4 Natural Language Processing

After exploring several natural language processing programs, such as Automatic Speech Recognition [15], we decided to use the CMU Sphinx II program for our experiments, described in Section 9.4. The reasons behind our choice were the open source nature of the Sphinx project, the lack of person-specific training for the system, and the perspective of a JAVA based system, Sphinx IV, currently under development. For our experiments, we used a version Sphinx II, tailored to understand individual words. Performance was close to real-time and could be improved with fine-tuning of parameters.

In terms of voice production, we are still exploring alternatives, which currently include Festival [45] and AT&T Natural Voices [16].

8.2.5 Planning, Goal and Task Scheduling

The robowaiter needs to be able to produce partial plans for various tasks in order to produce long term goal-directed behavior and to make progress on complex tasks. For example, in fulfilling orders, the robot needs to be able to take the order to the kitchen and plan on retrieving the food or drink at a later time. Two functional elements are required: a planner, which produces sequences of actions the robot needs to accomplish in the near future, and a goal stack, which keeps track of the tasks the robot needs to achieve. For example, fulfilling an order may require the robot interacting with the customer, getting an order, taking the order to the kitchen, retrieving the order, and serving the order to the customer. That order may be generated by a planner. However, to improve efficiency, the waiter may take multiple orders between trips to the kitchen. The goal stack is then used to keep track of the uncompleted tasks of the robot.

Task scheduling also requires a determination of the number of concurrent activities the robot is able to pursue. In ADE, that number is partially determined by the resource constraints placed on each component type in the architecture. An example of how these
8.2.6 Self-awareness

In order to better interact with the customers, the robot needs an self-evaluation module. This module is used to assess the robot’s interaction with customers in terms of the effects its actions, including speech, gestures, and timeliness, have on the attitude of the customer. The evaluator observes the rest of the system in order to construct relations between actions and observations, and thus construct a credit assignment mechanism. A self-awareness component is built into the experiments of Section 9.2.

8.2.7 Emotional Evaluation

A robotic waiter needs to be able to assess the emotional state of the customer it is serving. It also needs to have different ways of handling different emotional states. For example, the robot may respond to an angry customer with a stern reply, or it could go look for a manager. Similarly, the robot may change its response based on the customer’s emotional state, e.g., by telling a joke if the customer is perceived as ‘happy.’

The process of emotional evaluation consists of a combination of visual and auditory processing, e.g., a smile and fluctuations in pitch may denote a happy customer, while a loud voice coupled with furrowed eyebrows may denote an angry customer. An emotional evaluation component would have to connect to both visual and auditory processing components. The connections would most likely be made through O-links and the evaluation component would be responsible for extracting the relevant features from the data.

An additional aid to emotional evaluation will be provided by the gesture recognition feature of the visual processor, as body language is often a clear indicator of emotional state. For example, short movements may be an indicator of anger, while ample hand gestures may indicate
8.2.8 Memory

The robowaiter memory system needs to keep track of a variety of things. For example, at any one time, the robot needs to know what customers it is serving, each customer’s order, where each customer is, where other waiter are, the load under which the system is placed, i.e., whether the system can handle more tasks or not. At other times, the robot may require additional knowledge, such as who may be able to assist in cleaning a spill.

The architecture will instantiate a separate component for each customer, as well as for each order. Each of these will be simply a structure holding relevant information, as indicated in the previous section. Similar structures are created for other waiters, managers, etc.

A coordinating component can be connected to all these memory elements and determine from the number of customers, order, etc., the load on the system. This element can then play the role of the $D$ element in the dynamic architectures of Section 5.3 and enforce changes in behavior-selection strategy.

8.2.9 Personality

The robowaiter will be provided with a database of jokes, which it will attempt to use appropriately in conversation. It will also try to learn jokes as part of its customer interaction and use those jokes in interaction with other customers. This module is introduced as a measure of eliciting interest from the people interacting with the robot.

8.2.10 Robowaiter Subsystems

In this section we break down several functionalities of the waiter either to or very close to the level of ADE components. In order to be a waiter, the robot needs to be able to take orders from its customers. The schematic of the components involved in the process of taking an order is presented in Figure 8.4.
Figure 8.4. The schematic of the “Take Order” behavior for a robot waiter.

As can be seen from the figure, the order taking process has a complex structure, potentially involving components in all the high-level constituents of the system, e.g., the order model affects memory and customer interaction requires both visual and natural language processing facilities. In the process of taking an order, the robot interacts with the customer. This interaction has to be appropriate to the circumstances in which the
robot finds itself. For example, if the customer is rude, the robot should be less inclined towards chit-chat.

To provide this functionality, we map it onto a subsystem whose precondition is that the robot have started an interaction with a customer, e.g., has greeted the customer and is engaged in conversation, and which is active until that interaction is over. The structure of the system is shown in Figure 8.5.

Figure 8.5. The schematic of the “Deal with Customer” behavior for a robot waiter.

The top-level component, receives input from the emotional evaluator and decides, based on the emotional state of the customer, how to interact with that customer. Each low-level component is a template of customer interactions, containing the basic characteristics of that type of communication.

Another important task of a waiter is cleaning up. A schematic for that task is shown in Figure 8.6.
Figure 8.6. The schematic of the “Clean Area” behavior for a robot waiter.

Prerequisites for this behavior becoming active include seeing a spill on the floor or discarded items on a table. In order to clean the area, a waiter needs to have the necessary tools: a mop for a spilled drink, a broom and dust pan for a broken glass, etc. Thus, the first sub-behavior moves the robot from its current location to the location where it can acquire the tools. The second step takes the robot, now in possession of the required tools, to the area requiring cleaning. At that point, the sub-behavior that applies to the situation, e.g., “Remove Spill” for a spilled drink, is executed in order to fulfill the “Done Cleaning” goal. It should be noted that, for ease of understanding we used a rhombus to denote another, independent subsystem. In the case of Figure 8.6, the “Plan’N’Go” subsystem represents the standard combination of functional components from Figure 8.7. This subsystem receives a set of coordinates (through an A-link) and plans a route from the current robot location to the location represented by those coordinates.
If the Clean Area behavior was triggered by items discarded on a table, the robot should, upon removing the discarded items, prepare the table for the next set of customers. This may include adding napkins to the table, bringing new sets of silverware, and refilling the salt shaker. In order to restock the table, we provide the robowaiter with a sub-system designed specifically for that purpose. This system is shown in Figure 8.8.

In this chapter we have discussed some of the problems encountered in our quest for a complex robotic agent, including computational power and synchronization among functional units. We have described how ADE can be used to provide solutions to these problems, and we have shown schematics for the intended robowaiter architecture. In the
next chapter we look at the experimental groundwork which has put us in the position to implement a robotic waiter for the AAAI Intelligent Systems Demonstration, July 2004.
CHAPTER 9

EXPERIMENTS

In this chapter we present several simulated and robotic experiments with agents whose architectures are built on APOC principles. These experiments are used to illustrate the effectiveness of APOC/ADE in agent design and suggest avenues for future research.

9.1 Virtual Multi-Agent System

ADE can be used to implement multi-agent systems in various ways. Each agent can be implemented by one or more APOC components, which themselves may reside on one or more hosts. Agents can interact with other agents through APOC links and, in the latter case, dynamically create, modify, and destroy parts of their and other agents’ architectures. Which design is to be preferred will depend on various factors, such as the task to be accomplished by the multi-agent system, the available computational resources, etc.

In the following example, we first specify the task and lay out the architectures of the multi-agent system. Then we show how the system can be implemented in ADE.

9.1.1 Agent Task

The task for the multi-agent system is an information retrieval task, in which various domains on the internet, e.g., ”.edu”, or ”.ac.at”, are to be searched for web pages con-
taining given keywords, e.g., “agent toolkits.” The results should be ranked according to additional criteria. An example of a criterion which could be used for page ranking is the rate of occurrence of key expressions such as “distributed architecture.”

9.1.2 Agent Architecture

The architecture layout of the multi-agent system and its initial state are shown in the left and right halves of Figure 9.1(a), respectively. Three types of agent can be present in the running virtual machine. The Start-up agent takes a string to be searched for, e.g., “agent toolkit,” a set of relevant expressions for that search, e.g., “virtual and robotic agents” or “distributed agent architecture,” and a set of domains over which the search is to be performed, e.g., “.edu”, “.ac.it.” Once the Start-up agent has received its data, it creates one Search agent for each of the given domains. As in all ADE architectures, agent creation takes place within the resource limits specified in the architecture layout. If the number of domains exceeds the instantiation limit of the search agents, then the search is performed in part sequentially. This illustrates how resource constraints determine in part the task scheduling capabilities of an ADE system.
The *Start-up agent* passes the relevant data (i.e., the search string, the domain name, and the relevant keywords) to the *Search agent*. Then the *Search agent* uses one or more
of a set of web-search engines, e.g., www.yahoo.com to search for the information given by the search string and collects the results. Once the search is completed, the Search agent creates Evaluation agents, each of which receives a document and the set of relevant keywords. The state of the system after all Evaluation agents have been created is shown in Figure 9.1(b). Each Evaluation agent then parses its document, rates it based on the occurrence of the relevant keywords, and returns the rating to the Search agent. The Search agent, in turn, selects all documents with a rating higher than a given threshold and returns them to the Start-up agent. When all Search agents have returned their information, the multi-agent system has completed its task.

Note that the details of how the various agents achieve their tasks are not displayed in ADE. For the sake of simplicity, they have been implemented in the update function which represents each agent. It is, however, possible to distribute the computations perform by each agent over multiple APOC components, in which case the details of their implementation can be visualized and manipulated graphically in ADE.

9.1.3 Setup in ADE

The set-up steps for the experiment are as follows:

1. Start the Registry, in this case on airolab2.cse.nd.edu
   
   java com/ADERegistry

2. Start the GUI
   
   java APOCstart

   The GUI server then automatically starts the APOCServer(s) by connecting to the remote host(s) specified in the configuration file, in this case airolab4.cse.nd.edu and airolab5.cse.nd.edu, and running the following command on each:

   java com/apoc/APOCServer <registry hostname>

   The information display features of both components and links in ADE can be used to supervise the progress of the agents in this task. Possible uses include
• inspection of the links between Search agents and Evaluation agents for the actual web address being evaluated

• inspection of the Evaluation agents for the current rating of the address being evaluated

• inspection of the Search agents for the highest current evaluation of an address

If, as a result of an inspection, the user decides that a particular address is not worth exploring, the corresponding Evaluation agent could be deleted, freeing up resources for the system to evaluate another address.

9.2 Dynamic Behavior Selection Mechanisms in Robots

Since behavior-based architectures are primarily intended for robotic agents, where real-time performance is critical, we demonstrate the utility of adaptive behavior selection for combinations of the seven cases proposed in Section 5.2 on a robot in three different experiments. The task is the same for all experiments: the robot needs to locate and reach a target location, in this case an orange ball, as quickly as possible in an obstacle environment without bumping into obstacles, such as boxes and chairs. A snapshot of the robot in its environment is shown in Figure 9.2.
Specifically, we will report results from the following three experiments:

1. *Experiment 1*: the robot learns to constrain the set of behaviors used for behavior selection based on relevant sensory inputs for a given environment. This experiment illustrates *Cases 1, 2, and 7*.

2. *Experiments 2*: the robot changes combinations and sequences of behaviors to overcome deadlocks. This experiment illustrates *Cases 4 and 5*.

3. *Experiment 3*: the robot uses attentional mechanisms to detect a lack of progress towards its goal and changes its behavior selection strategy to overcome the problems. This experiment illustrates *Cases 3 and 6*.

The experiments in this section use several components which will be part of the robowaiter architecture:

- Visual processing. The robot used a blob detection algorithm embedded in an ADE component to identify its target.

- Navigation and mapping. The navigational techniques used in this section will be a part of the robowaiter navigational system.

- Learning. The robot learns several changes to its architecture through its interactions with the environment, leading it to more efficient navigation. The components used to implement the architectural changes will be part of the learning mechanism in the robowaiter.
9.2.1 Experimental Setup

All robotic experiments were performed on a Pioneer 2DXE robot from ActivMedia, which was equipped with a SONY PTZ camera, an onboard framegrabber, and an onboard PC running Linux. The generic architecture used in all experiments, shown on the left in Figure 9.3, consists of a Sensory Processing, an Action Processing, and a Supervisory Decision System part. The Sensory Processing part processes sonar, visual, and bumper sensor information received from the robot’s 16 sonar, camera, and 10 bumper sensors, respectively, in components $b_1$ through $b_{16}$, $bv$, and $bb$ and makes the information available to other parts of the system. For visual images, for example, the centroids of color blobs are computed and converted into directional vectors. The Action Processing part implements all behaviors as well as the employed behavior selection strategies (a schema-based cooperative mechanism, by default), while the Supervisory Decision System implements additional control methods dependent on the specific functionality to be tested in each of the three experiments.
9.2.2 Experiment 1

In this experiment, the robot repeatedly moves through the obstacle environment starting in the same area in an effort to learn how to traverse it as quickly as possible. On each run, it chooses sonar sensor nodes at random and deletes them temporarily. If their absence does not make a difference in the robot’s progress towards the target location, as measured in terms of the size of the identifiable target location in the visual image, the sensor node gets permanently deleted for the given setup (Case 1). Otherwise, the node is re-instantiated. An alarm mechanism connected to the bumper sensors ensures that the robot will not inflict any damage if relevant sensors get temporarily deleted (Case 2). Over time, the robot learns to remove irrelevant sensor nodes for a given task, which frees up system resources and leads to better performance at traversing the environment (Case 7).

It should be noted that although in this experiment, sensor nodes are deleted for the duration of the task, they can be re-instantiated if another task requires them. Thus, the experiment illustrates one way of learning fixed action patterns, such as the egg retrieval of the greylag goose, at the architecture level. In performing these fixed action patterns, only the minimal sensory information normally required for the completion of the task is processed. Figures 9.4 and 9.5 show two typical setups for this experiment and Table 9.1 summarizes the results of five consecutive runs in these environments.
The architecture implementing the cooperative behavior selection strategy that dynamically restricts and widens the pool of contenders is shown in Figure 9.3 (second from the left). The Supervisory Decision System structure consists of two APOC components, $d_1$ and $d_2$. $d_1$ collects sonar information and forms a composite “picture” of the environment. $d_2$ analyzes the environment, decides which components are expendable and deletes them from the architecture. The Action Processing consists of two components: $p_1$, which sets up $\Sigma$ to compute an overall direction towards the goal, and an alarm
mechanism \( p_a \), whose function is to prevent the robot from hitting obstacles and which, therefore, overrides \( p_1 \) when the robot is in a perilous situation.

The results of the original tests show an example of how an agent can attempt to learn a minimal set of resources to be used while executing a task. However, in terms of performance, little can be said about the benefits of eliminating some sensory information from consideration. Therefore, we ran a second set of tests, in which we modified the criterion for elimination of sonar nodes. Under the new criteria, sonar deletion operations took place every two seconds and at every deletion operation the longest \( k \) forward sonar readings were discarded. A learning component varied the values of \( k \) and performed statistical analysis of the times needed for task completion. When a further deletion causes no additional improvement in performance, the learning node fixes the value of \( k \) and that value will be used whenever the agent attempts to perform that task.

Tests were run in the same set-ups as with random sonar deletions. A slight modification was made in the starting location of the robot in the left turn set-up, as the robot was started slightly to the right of the center of the corridor. The results of the tests are shown in Tables 9.2 and 9.3; times were taken using the java System.currentTimeMillis() call and are given in seconds.

In order to have a better understanding of the effects brought about by the architectural changes, we performed a t-test on the resulting traversal times. Table 9.4 summarizes the results. As can be seen from the results, the dynamic architecture modification is beneficial in both scenarios. The traversal times in the runs with one component deleted are in both cases significantly lower than in the two baseline runs (\( p < 0.05 \) using a t-test).

The better results generated by removing the longest sonar reading from consideration for two seconds at a time are explained differently in the two set-ups. In the first set-up, the robot starts moving slightly towards the left, which brings it close close enough to the left edge of the corridor that its influence is significant in determining the trajectory
TABLE 9.2

THE TIMES AND NUMBERS OF DELETED SONAR NODES FOR RUNS IN THE FIRST SET-UP OF EXPERIMENT 1.

<table>
<thead>
<tr>
<th>Run</th>
<th>All Sonars</th>
<th>1 Sonar Deleted</th>
<th>2 Sonars Deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.351</td>
<td>17.081</td>
<td>20.493</td>
</tr>
<tr>
<td>2</td>
<td>18.036</td>
<td>19.668</td>
<td>17.042</td>
</tr>
<tr>
<td>3</td>
<td>19.172</td>
<td>19.450</td>
<td>17.622</td>
</tr>
<tr>
<td>4</td>
<td>34.302</td>
<td>16.813</td>
<td>19.287</td>
</tr>
<tr>
<td>5</td>
<td>17.062</td>
<td>17.273</td>
<td>17.976</td>
</tr>
<tr>
<td>6</td>
<td>28.350</td>
<td>17.354</td>
<td>18.893</td>
</tr>
<tr>
<td>7</td>
<td>19.467</td>
<td>16.621</td>
<td>16.860</td>
</tr>
<tr>
<td>8</td>
<td>18.748</td>
<td>17.239</td>
<td>21.871</td>
</tr>
<tr>
<td>9</td>
<td>19.334</td>
<td>17.656</td>
<td>19.859</td>
</tr>
<tr>
<td>10</td>
<td>18.857</td>
<td>17.216</td>
<td>18.281</td>
</tr>
</tbody>
</table>

of the robot. If no sonars are discarded, the repulsion of the left wall causes the robot to make a slight turn to the right, causing a wide turn. However, if the longest sonar is discarded, as soon as the opening on the left can be sensed, the lack of repulsion from that sensor roughly cancels out the repulsion from the left wall, causing the robot to take a tighter turn. With two discarded sonars out of a total of eight the amount of data discarded caused the robot to get close to the left wall and therefore have a wide exit from the turn, causing a decrease in performance.

The results in the second set-up show a similar, though more pronounced, trend. In the S-shaped environment, the longest sonar reading is most often in the direction of the opening. Thus deleting the longest sonar reading from consideration caused the robot to
TABLE 9.3

THE TIMES AND NUMBERS OF DELETED SONAR NODES FOR RUNS IN THE SECOND SETUP OF EXPERIMENT 1.

<table>
<thead>
<tr>
<th>Run</th>
<th>All Sonars</th>
<th>1 Sonar Deleted</th>
<th>2 Sonars Deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DNF</td>
<td>21.859</td>
<td>20.021</td>
</tr>
<tr>
<td>2</td>
<td>DNF</td>
<td>22.442</td>
<td>bump</td>
</tr>
<tr>
<td>3</td>
<td>35.060</td>
<td>20.408</td>
<td>bump</td>
</tr>
<tr>
<td>4</td>
<td>30.546</td>
<td>31.498</td>
<td>20.507</td>
</tr>
<tr>
<td>5</td>
<td>33.001</td>
<td>30.717</td>
<td>bump</td>
</tr>
<tr>
<td>6</td>
<td>28.484</td>
<td>21.127</td>
<td>bump</td>
</tr>
<tr>
<td>7</td>
<td>37.650</td>
<td>21.754</td>
<td>bump</td>
</tr>
<tr>
<td>8</td>
<td>DNF</td>
<td>28.510</td>
<td>20.438</td>
</tr>
<tr>
<td>9</td>
<td>DNF</td>
<td>21.115</td>
<td>bump</td>
</tr>
<tr>
<td>10</td>
<td>57.610</td>
<td>25.319</td>
<td>21.034</td>
</tr>
</tbody>
</table>

TABLE 9.4

THE AVERAGE TIMES TO TARGET AND CONFIDENCE INTERVALS FOR EACH OF THE 10 RUNS WITH 0, 1, AND 2 COMPONENTS DELETED FOR THE TWO TRAVERSING SCENARIOS OF ROBOT EXPERIMENT 1.

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Conf.Int.</td>
</tr>
<tr>
<td>Baseline</td>
<td>37.06</td>
</tr>
<tr>
<td>1 deleted</td>
<td>24.38</td>
</tr>
<tr>
<td>2 deleted</td>
<td>20.55</td>
</tr>
</tbody>
</table>

enter earlier into turns towards the open area. With a small array of sonars, deleting the longest two would often cause the robot to not turn enough to avoid the inside wall of the curve it was negotiating, leading to several collisions.
9.2.3 Experiment 2

In this experiment, the robot eventually encounters a narrow passage on its way to the target location, which it has to pass (see Figure 9.6). If the target is placed in the center after the passage (left in Figure 9.6), the robot will typically be able to go through using its standard cooperative behavior selection mechanism. If, however, the target is placed off to one side, partly hidden behind an obstacle as in the right two figures in Figure 9.6, the robot will either fail to go through the passage or lose track of the ball. Various factors, such as wheel slippage, noise in the sonar values when it gets too close to an obstacle, and others, cause the robot to be unable to center in on the ball directly. By dynamically reconfiguring the relative contributions of different behaviors, in this case motor schemas, based on the context, i.e., whether or not the robot sees the target (Case 5), it is possible to extend the robot’s behavioral repertoire and recombine behaviors (Case 4) to overcome the deadlock.

![Figure 9.6. The robot’s trajectory for different setups for experiment 2: with centered target behind the passage way (left) and the target off to one side, without (middle) and with supervisory control (right).](image)

The architecture implementing the context-dependent dynamic modifications of the relative contributions of each behavior in the cooperative behavior selection strategy is the second subfigure from the right in Figure 9.3. The *Supervisory Decision System*
consists of a single component, $d3$, which processes the visual information available to the system and chooses which of two decision components will be instantiated in the architecture. The Action Processing section thus consists of two decision components: one component, $p2$, configures $\Sigma$ to use a constant attraction to the ball, while the other, $p3$, increases the attraction of the ball over time for a given time interval, after which it is reset to its original value. The later implements an adaptive behavior selection strategy, as it will change the relative contribution of ball attraction and obstacle repulsion until either the robot can advance through the passage or the time limit of the adaptation process is reached, i.e., the point at which it is assumed that adaptation is not able to resolve the impasse.

We performed ten runs with the non-dynamic architecture and the ball placed off-center on the other side of the passage. The robot failed to traverse the passage in all ten runs. With dynamic behavior selection, the robot only failed to traverse the passage in two out of ten trials. This shows that the addition of components $d3$ and $p3$, which establish the context-mapping and allow for dynamic switches among configurations of gain values for motor schemas, is another effective way to overcome local minima. Note that this benefit can be achieved using minimal computational and architectural requirements. Moreover, the dynamic behavior selection mechanisms can be added to an existing schema-based architecture with only minor modifications.

9.2.4 Experiment 3

The third experiment uses the setup of experiment 2, except that the opening between the two obstacles is now too small for the robot to be able to pass through safely. On a typical run, the robot will proceed towards the ball using the standard cooperative mechanism. Once the obstacles are reached, the robot will be locked into oscillatory behavior regardless of where the ball is placed on the other side, i.e., it will move towards and away
from the obstacles without making any progress towards the target location. In this case, even the dynamic behavior selection mechanism from the above experiment that helped align the robot properly for traversal will fail, as the passage is too narrow for traversal and the repulsive forces exerted by the obstacles consequently cause the robot to turn away.

This is where an attentional mechanism proves useful (Case 6): by noticing that no progress is made towards the target (comparing different perceptions of the target and robot movements over time), the mechanism switches behavior selection to competitive mode to avoid impossible actions (Case 3) and turns off visual input, which effectively leads the robot to move around the obstacle. Once the target reappears, the supervisory mechanism re-enables visual input and full cooperative behavior selection.

![Figure 9.7. Robot environment for experiment 3 showing the robot path](image)

The architecture implementing the performance-based attentional control, which can switch among different behavior selection strategies, is shown in the rightmost subfigure of Figure 9.3. *Supervisory Decision System* consists of two components: *d4*, which observes the vision sensor and identifies the contents of the image seen by the robot,
and \(d_5\), which monitors the progress made by the robot towards achieving its goal. If the progress is unsatisfactory, \(d_5\) interrupts the decision component which is currently active \((p_4)\) and starts the inactive one \((p_5)\).\(^1\) Thus, \textit{Action Processing} consists of two components with different characteristics: \(p_4\) is a component which uses cooperative decision making to choose a behavior, whereas \(p_5\) is a competitive decision maker.

As with the other experiments, we performed ten runs without dynamic behavior selection for both the basic schema-based architecture and the architecture augmented with the mechanism from experiment 2. In both cases, the robot exhibited an oscillatory motion at the entrance to the passage without ever getting around the obstacles. When dynamic behavior selection was used, the robot moved around the obstacle in all ten runs, showing that attentional mechanisms which can detect deadlocks are useful for controlling dynamic changes of behavior selection strategies.\(^2\)

9.2.5 Setup in ADE

The setup steps for the experiment are as follows:

1. Start the \textit{Registry};
2. Start the \textit{GUI};
3. Start the \textit{RobotServer} and the \textit{ImageAcquisitionServer}. This step and the previous one are interchangeable;

\[
\text{java com/pioneer/PioneerServerImpl <registry hostname>}
\]

\[
\text{java com/framegrabber/FramegrabberServerImpl <registry hostname>}
\]

\(^1\)Note that while in the present setup the threshold for switching from cooperative to competitive behavior selection is fixed, it could, as in the simulation experiments, be learnt using reinforcement learning.

\(^2\)Note that while the robot will typically be able to move around obstacles in the environments used for our tests, there are environments where it will get stuck, e.g., because moving around the blocking obstacle is not possible. In that case, the supervisory component will consider the attempt to surround the obstacle a failure and switch back to the standard behavior selection strategy, thus effectively restarting the target localization process.
4. Start the \textit{ImageProcessingServer};

\begin{verbatim}
java com/camview/CamviewServerImpl <registry hostname>
\end{verbatim}

5. Define the robot architecture using the graphical tool and predefined components; and

6. Run the experiment by switching to \textit{Synchronous} or \textit{Asynchronous} Mode.

The entire set-up process can be completed by using a script, thus allowing the user to issue a single command to initialize an ADE system.

Various components in the architecture require access to robot related resources, such as its sensors and effectors, as well as other information gathered from utility servers. In this experiment, one component requires visual information, a second requires access to robot sonar and motor information, while a third requires access to the robot motor effectors. Each of these components contacts the registry, requests the server it requires and sets up direct communication channels with that server. As a result the registry acts only as a resource manager, and will not become a bottleneck for communication within the system.

For visual processing, a standard blob-detection algorithm was used to identify the ball and run on a separate utility server, the \textit{ImageProcessingServer}, to improve the parallelism of processing and thus the performance of the system. Two of the servers, the \textit{RobotServer} and the \textit{ImageAcquisitionServer}, can only run on the on-board computer of the robot used in the experiment, as they need direct access to hardware components: robot sensor and effectors, as well as camera sensors and effectors. The \textit{Registry} was run on a Sun workstation, while the \textit{GUI} server and the \textit{APOCHServer} were run on PCs.

ADE architecture inspection mechanisms can be useful in several ways:

- Inspecting the \textit{VisionNode} allows the user to see what the agent sees, as shown in Figure 7.2.

- Inspecting the \textit{CooperativeDecision} and \textit{CompetitiveDecision} nodes allows the user to view the commands being sent to the motors and identify discrepancies between expected and actual behavior; and
• Inspecting the links between the *VisionNode* and the *SupervisoryNode* allows the user to see the information available to the decision node and to ascertain any delays in communication which may be present in the system.

The ability to stop and restart the robot can in this instance be coupled with inspection mechanisms. The user can thus stop the robot and analyze its position in the environment. Based on this analysis expectations about the contents of the architectural components, e.g., sonar sensor values, ball location, the user can pinpoint design flaws in either individual components or the layout of the architecture.

The above example illustrates the “real-time” nature of ADE, as its distributed nature enabled it to appropriately control a robotic agent. This feature separates ADE from most existing agent toolkits, where robot control is external to the toolkit itself.

9.3 Visual tracking

In this section, we present a system for motion detection and tracking that is especially target at the severe resource limitations of small autonomous robots. It uses an attentional mechanism to select different methods and processing constraints from several modules for the detection and prediction of motion in a scene. The components used in this experiment will be a part of the following robowaiter subsystems:

• Visual processing. A color and motion blob detection component was used in these experiments. This component will be a part of the robowaiter vision system.

• Tracking. The tracking mechanism used in this experiment will be augmented for the robowaiter architecture.

• Navigation and mapping. The navigation system used to synchronize the motion of the camera with that of the robot will be augmented and used in the robowaiter architecture.

Next, we first discuss the overall system architecture and the motivation for the design. Then we present the implementation and report very promising results from several sets of experiments with the system.
9.3.1 Background

A significant amount of research has been performed for motion detection and surveillance. Still image analysis results include a variety of approaches ranging from color analysis [108] to the use of genetic algorithms [23]. The active blobs approach relies on building 2-D mesh representations of surfaces. Deformation of a mesh via transforms together with texture mapping are then used to identify deformable objects [95]. Eigenspace approaches [24] use training images and matrix decomposition to create a database of feature images which are used to identify detail of new pictures. An efficient method for a rotating camera was presented by Dellaert and Collins [41]. The algorithm relies on differences between stored images of the surrounding area, taken at various angles.

Several of the algorithms mentioned above, e.g., as presented by Sclaroff and Isidora [95], are intended for off-line image analysis. As such, they are not designed to function under the type of computational power constraints imposed by an autonomous robotic agent. The algorithms which are designed with a view towards real-time applications, e.g., the algorithms described by Dellaert and Collins, and Black [24, 41] require training and may still prove too complex to run on simple, autonomous robots. Others that take resource-constraints seriously (e.g., [29]), fail if colors cannot be reliably detected.

What is needed is an online, real-time system that can integrate different information to improve detection and tracking performance despite limited resources. In particular, we require that no assumptions be made about the quality of the color of an object nor about its uniqueness in a scene. Nor should any restriction be imposed on the lighting conditions, e.g., to provide glare-free lighting. Finally, no assumptions should be made about temporarily occlusions of the objects to be tracked by other, possibly moving objects.

All these criteria together make it very hard, if not impossible to show close-to-perfect performance on high-performance systems, let alone on low performance embedded systems. However, it is possible to achieve a reasonable performance that might be sufficient
for many real-world applications. In the following, we first describe the architecture of our proposed system and then show the results of the experimental validation of the design.

9.3.2 The System Architecture

The system consists of five major, concurrently active modules: the color blob detection, the motion blob detection, the blob tracking, the blob selection and camera control, and the attentional subsystem. These components and their interconnections are shown in Figure 9.8. In the following, we will describe the functionality of each modules as well as the employed methods for reducing computational resource requirements.

![System Architecture Diagram](image)

Figure 9.8. The system architecture for ball tracking in a robotic agent

Fast Color Blob and Motion Blob Detection

Visual processing is one of the most computationally demanding components of a complex agent. Within visual processing, mo-
tion and color detection are the most expensive subsystems, having a computational com-
plexity for the processing of an image of $O(\text{width} \cdot \text{height})$, where width and height are
the dimensions of the image. For our experiments, each color is defined in terms of three
intervals in RGB space. The system is designed such that an arbitrary number of colors
can be passed as arguments to the color blob detection algorithm. Time cost for searching
for multiple colors is linear in the number of different colors. The algorithm for color blob
detection brings several improvements designed by Dr. Matthias Scheutz to the algorithm
described by Bruce et al. [29]). Centroids and bounding boxes for each blob are available
after processing. The blob detection code also offers a parameter for specifying a thresh-
olds for blob size. For example, detected blobs whose sizes are less than the minimum
are discarded and not returned to the user.

The detection of motion blobs is done in a similar manner, with pixels in two con-
secutive images being compared according to their differences along each dimension in
RGB space. Each dimension is afforded an error interval and if the pixel does not fall
within that interval, it is considered as part of a motion blob. Similar centroid, bounding
box, and lower threshold functionality is available as for the color blobs.

Given the nature of the task and the requirement that it run on an autonomous system,
a further optimization was made available to users. The area on which blob searches are
performed can be restricted to a rectangular sub-region of the image. This allows for
increased performance if restrictions can be made in an informed manner.

9.3.3 The Attentional Subsystem

The attentional system is used to find objects in a scene and track them if possible.
Users can define objects in terms of color, relative size, e.g., small, and motion if nec-
essary. When a new request is received by the system, the attention system reconfigures
the vision system to look for the new object in addition to previously tracked items. For
example, a request to “track the small blue car” configures the vision to search for a blue object and sets a threshold for the maximum size of blob which is to be returned to the user. For each blue object that has been identified, a tracker will be instantiated that will independently attempt to track the object. If multiple trackers are instantiated, the system chooses one tracker to follow, while attempting to keep as many blobs as possible in the field of view. In inconsistencies arise, e.g., blobs moving in opposite directions, each tracker receives a priority from the attentional system and the commands sent by that tracker are followed by the camera.

An optimization made to the code is that images taken during camera motion are discarded by the blob detection algorithm. The reasoning behind this optimization is that camera position information in our system is not always accurate during motion. Therefore, blob detection during motion may lead to misidentification of blobs. In order to discern when camera motion has ceased, the total area determined to be part of motion blobs is returned to the attentional system. When that area falls below a threshold, normal blob detection resumes.

9.3.4 The Motion Tracker

Motion trackers are units whose function is to attempt to follow blobs as they move through the environment. Each blob tracker is configured to follow a particular type of blob, such as a blue moving blob. If the blob moves outside a user-defined region in the image, the tracker sends a motion command to the camera controller in order to maintain the blob on the screen.

Trackers compute potential future blob positions based on the position of the camera and the motion of the blob from the last image to the current one. The computation defines a region in space in which the blob is expected to be found for the next image. Camera motion is directed towards this region which is then searched for the blob. If the blob is
not found within the expected region, e.g., due to a change in the speed, the search area is expanded to include the entire image at the current camera location. If the blob can still not be located, the tracker waits for a predefined number of images after which it terminates itself.

9.3.5 The Blob Selection and Camera Control

The blob selection and camera control module is concerned with moving the camera fast enough to keep the blob be designated by the attentional system centered. The camera is controlled by two PID controllers, whose gains have been experimentally determined. Of particular importance for moving objects are the non-zero ID components, which can implement an inertia effect in the camera movement that facilitates the tracking of moving objects, as shown in experiment 2 below.

9.3.6 Experimental Set-up

After presenting the details of the functional organization of the system architecture, we now describe the experimental setup to test the system and verify its design. All experiments were conducted with an ActivMedia Pioneer 2 robot with a Sony Pan-Tilt-Zoom camera. The robot has an onboard PC104 computer running RedHat Linux 8.0 with a 850 MHz Pentium III CPU and 128 MB of RAM, and a low-level motor control board with its own embedded processor. Communication with the motor control board, which sends commands to the effectors and retrieving sensor information, was achieved using an asynchronously-running server written in JAVA that wraps the low-level controller commands. The tracking system was connected to the JAVA server via JAVA remote method invocations (RMI), which allows for two different setups: the tracking system can either run on the robot or on a remote host, in which case it connects to the robot through a wireless ethernet. The performance of the tracking system was tested in both setups.
The architecture of the tracking system was developed and tested using ADE. Four ADE component types were used to implement the five tracking system components, where motion and color blob detection were merged into one ADE component for efficiency reasons.

9.3.7 Experiments

We ran three sets of experiments: (1) with a stationary camera on a stationary robot, (2) with a moving camera on stationary robot, and (3) with a moving camera on a moving robot. For each set we considered two environments: one, in which the tracked object was first never occluded, and one, where the object to be tracked was visible in the beginning, and subsequently occluded for short periods of time. We used several objects to test the system, but will report only results from tracking of a single, small blue toy car. This object was more difficult to track than the other objects we used to test the system, such as the kind of orange soccer ball used in robo-soccer. Two reasons contributed to the added difficulty: its size, as the car was much smaller than our usual target, and its color, which was close to the color of the carpet in our lab. These two characteristics combined to often make it impossible to get a unique color blob for the car.

Experiment Set 1: Stationary Camera In the first experiment, the car had a distinctive blue color which facilitated its detection through color blob image analysis. Blob information was sent to a Tracker unit. The Tracker then iterated through all blobs, using centroid, area, and boundary information to determine which blob is most likely to represent the car. Using color information as the primary means for tracking led to very reliable tracking of the car in an environment with several moving objects of different colors.

In our second experiment we added an obstacle which occluded the car for half a second to a second at a time. The obstacle was not wide enough to fill the entire image captured by the camera. The camera moved slightly past the beginning of the occlusion
and stopped with both ends of the obstacle in view. Thus, when the car emerged on either side - whether continuing its trajectory or reversing its motion, the camera continued its tracking. The visual information available to the system is shown in Figure 9.9.

![Figure 9.9. Camera images and tracker information for stationary camera tracking: unprocessed image (top), tracker information (bottom)](image)

**Experiment Set 2: Moving Camera** Here, we performed four sets of experiments: the first two were identical to the stationary camera set. However, motion detection was introduced for the first time into the system. Another new addition to the system was motion estimation: if the car position was known for two iterations through the control cycle, then an extrapolation was made of the probable camera location. The search was then initially limited to this area and, if that failed, generalized to the entire image.

The third and fourth tests, described below, tested the interplay between blob detection and motion detection in tracking a moving object. A PID controller was used to move the camera to the best position for ball tracking. For best results in our system, we have found that a P-gain of 0.26, a I-gain of 0.2 and a D-gain around 0.2 allowed the camera to track the car both in the unoccluded and the occluded experiments.

For the first two experiments we moved the car by holding it in one hand. In our third
experiment, we started the experiment in a similar manner. However, at some point we
closed the hand around the car, continuing the motion. At that point, the tracker could
no longer rely on color information for tracking, relying instead on motion information.
In order to better recover the car if/when it reappeared on the screen we kept information
about the size of the car. And compared color blob sizes against the known size of the
car.

A fourth experiment further tested the limits of our system. As in the third experiment,
at some point the car was hidden in the hand. Here, however, the other hand was at this
time also moved to and fro in front of the camera. Tracking was satisfactory, though
sometimes, depending on thread scheduling and how distinct the moving objects were in
the picture, the tracker would latch on to another moving object and start following it.
This was especially the case if two moving objects came close enough to be identified as
a single motion by the algorithm. The visual information available to the system is shown
in Figure 9.10.

![Camera images and tracker information for mobile camera tracking](image)

Figure 9.10. Camera images and tracker information for mobile camera tracking: un-
processed image(top), tracker information (bottom). Two trackers are present in some
images; the main blob is moving to the left
Experiment Set 3: Moving Camera + Moving Robot  By adding motion to the robotic platform itself, this became by far the most complex task of the system, as the motion detection became far less useful in limiting the areas which were searched for the ball. However, the motion extrapolation proved to be beneficial in this case, as the motion of the car as it moved, in both a straight line and an a random motion was followed by the agent.

The additional load of computing motion and controlling the robot wheels led the system to slower response times, and more frequent loss of tracking in the occlusion tests.

With the complete architecture in place, cycle times were taken to evaluate the performance of the architecture. Two measures were considered: the time needed for visual processing and the time needed for one complete update of the architecture. Visual processing averaged 121.65 ms per cycle with a standard deviation of 231.94. Overall cycle times averaged 146.94 ms, with a standard deviation of 103.07.

Three conclusions can be derived from these results:

- Visual processing was responsible for most of the CPU time used by the system (82.79%)
- Individual component processing times tend to vary significantly due to the threaded nature of the system
- The overall system variation in execution times is much less than its component parts, since most thread switches, with such exceptions as the robot server updates, occur among components of the architecture

9.3.8 Implementation Details and Optimization Measures

In this section we present some of the ideas, optimizations, and heuristics used in our system in order to maximize performance.

Threading and Parallelism  In autonomous agents, time is the critical resource: it is essential that operations are performed effectively and in a timely manner. An implicit
consequence of this is that threading and parallel execution should be used wherever possible.

Image capture  On the Pioneer 2 robots, the framegrabber uses DMA to transfer images to memory. On an 850 MHz processor, using a 160 pixel by 120 pixel image, each transfer takes roughly 0.1 seconds; the highest possible framerate is 10 fps. It is therefore necessary that other processes execute in parallel to the image capture. In JAVA, this translates to image capture being performed in a separate thread in the image server:

```java
public void run() {
    byte[] frame = null;
    while (true) {
        frame = frameGrabber.getFrame(quality);
        synchronized(imageGuard) {
            image = frame;
        }
        Thread.yield();
    }
}
```

In the code above, the `image` is accessible to the outside world and access to it is restricted in order to prevent reads being performed during a write. Also notable is the explicit `Thread.yield()` call after each read. We have noticed that the JAVA thread scheduler does not always preempt threads in a timely fashion, leading to severe time lags in camera movement when the `yield` is omitted.

Other threads  Robot sensory updates were run as an asynchronous thread. Each architectural component was also run as a thread, thus allowing the robot updates to interweave with computation. All threads performed a `yield()` at the end of their update cycles, providing a simple mechanism for concurrent updating.

The time to get a new image from the robot using the framegrabber is approximately 100ms. Therefore, the image analysis node should retrieve images from the robot every
tenth of a second. This was again an opportunity for threaded execution: in order to minimize both RMI calls and the time spent waiting for images to arrive, a thread in the image analysis node retrieves an image from the robot, sleeps for the remainder of time to 100 milliseconds from the start of the last access and starts a new retrieval.

Image analysis In searching for the car, motion detection was performed first, with color detection only in the regions, where motion was detected. Limitations on the area on which motion detection was performed were sometimes imposed based on estimations of the car motion. This estimate relied on knowing where the car was as two previous points in time. The difference between them was computed and taken as an estimate of how far the car may have moved since last seen. A rectangular search region is created by starting at the previously known location, adding the motion estimate plus an error correction factor, set for these experiments at 10%, and limiting the search to the region defined by the previous point, the tip of the motion estimate vector and horizontal and vertical lines drawn through the two points. This estimate amounts to allowing for two types of motion for the car: moving with a horizontal velocity component between 0 and 110% of the estimate, and moving with a vertical velocity component between 0 and 110% of the estimate.

For all but very fast and irregular motion patterns, this estimate proved to be very useful in maintaining a low turnaround time for image analysis.

In the following section we present the environment used for system development and some of the features which facilitated its testing.

9.3.9 System Development

The development of the architecture for our system was done in the ADE development environment. The JAVA base of the system facilitates the use of graphical tools in the design, testing, and running of a system. Each of the components used in the tracking
system was implemented as an ADE component.

The tests were run in two configurations. In the first test, all components of the architecture were run on the robot. In the second, a remote machine was used for visual processing and tracking.

Two steps were taken to ensure a more efficient search for the car: the motion and blob detection codes were optimized, and the colors for the car were optimized.

Due to varying lighting conditions, two color ranges were used to identify the car. The process of identifying and fine-tuning the colors identified as potential car matches was aided by the ease of adding visual interfaces to ADE components.

Figure 9.11. Control panel for dynamic color range adjustment

Figure 9.12. Control panel for dynamic adjustment of PID controller parameters
Figure 9.11 shows the panel which was added to the vision server and which was used to configure the parameters for blob detection. The panel displays three images: the unmodified camera picture (top), the results of performing blob detection on that image with the parameters set on the sliders (middle), and the results of performing motion detection on the original image.

Camera control was performed through the use of 2 PID controllers - one for horizontal movement and one for vertical movement. A similar calibration process was performed on the parameters of each controller, using another graphical tool (Figure 9.12), which allowed us to change parameters as the architecture was running.

The architecture for the above experiments was run entirely on the robot. However, due to its ADE-based implementation it was possible to also run the architecture off-board, on a 2.1 GHz PC, leaving only the robot servers and the control node to execute on the robot. A reduction of approximately 35% in the time required for one complete update of the architecture was observed.

9.3.10 Summary

In this section we showed an ADE implementation of an adaptive tracking system for autonomous robots that uses color and motion information to track moving objects, while the robot and its camera are possibly moving. The system achieves an average frame rate of 7 frames per second on an ActivMedia Pioneer 2 robot, where 10 frames per second is the maximum number of frames that can be obtained from the framegrabber in our setup. The performance evaluation showed that the system has good tracking performance under the less than perfect environmental conditions in which it was tested. These conditions included flickering lights, similar colors in environment to object tracked, and motion in the background. Most importantly, the system can run autonomously on a robot with limited computational power, which is what distinguishes it from most other systems.
9.4 Reference Resolution

In this experiment we used ADE to implement a robotic model for human reference resolution. AI handling of natural language sentences most often focuses on building the parse trees of the sentence and checks the knowledge base to see if it contains a set of elements which matches the parse tree. This approach guarantees that a correct parse will be found if one exists, although performance under real-time constraints can be limited, especially due to the potentially large number of parse trees [69]. Often, a more efficient approach, which does not require the construction of parse trees, is sufficient to find the correct semantic interpretation.

For our setup we used a simple Blocks World Domain environment, in which blocks of four different colors were placed at various locations. Relationships between blocks are described by on, under, or nextto. The two scenarios in which the system was tested are shown in Figures 9.13 and 9.14.

![Figure 9.13. Two-referent set-up](image)

A representative sentence that the system was asked to ‘parse’ was “Put the red object on the yellow object on the blue object.” In classical approaches, two parses would be created for this sentence, which are expressed in terms of logical predicates as:

![Figure 9.14. One-referent set-up](image)
- PUT(REDOBJECT,ON(YELLOWOBJECT,BLUEOBJECT)) and
- PUT(ON(REDOBJECT,YELLOWOBJECT),BLUEOBJECT).

With the environment in Figure 9.13, reference resolution for the first parse tree fails when trying to find objects which satisfy ON(YELLOWOBJECT,BLUEOBJECT), as no yellow object is found on a blue object.

Figure 9.15. A high-level view of the components employed in the architecture used in the reference resolution task.

The functionality of the system, as seen in Figure 9.15, was divided among eight APOC components types, as follows
The SphinxNode component is a JAVA wrapper created for the Sphinx II decoder and provides data from the decoder to other APOC components. The sphinx-2 program parameters were set such that individual words would be recognized and stored individually and not as part of a sentence fragment SphinxNode represents the Speech Recognition component of the architecture.

Speech Processor. The ‘SpeechProcessor’ receives data from the SphinxNode, analyzes the words and instantiates other components based on input. The speech input is then forwarded to the newly instantiated components. For example, upon encountering a verb such as ‘put’ it instantiates a representation of the verb and passes to the verb representation word information until the latter either completes its parsing operation or fails. Word Analysis is implemented in this APOC component, which also does some high-level Syntax Analysis, such as verb identification.

Verb representations. In this experiment, a ‘Put’ component is a representation of the verb “put” as an architectural component and is therefore a Conceptual Frame. When this component is created, it automatically creates a ‘Definite Description’ component.

Definite descriptions. A ‘DefiniteDescription’ component performs some Syntax Analysis/Lexicon, in that it searches for descriptions of two unique objects in the sentence, one as an object on which the action is applied and one as a target for the action. This component also instantiates relational Conceptual Frames, such as the ‘On’ component. Thus, DefiniteDescription performs the functions of Referent Analysis and Spatial Relation Analysis.

Visual processor. The ‘MotionBlobServerNode’ component combines several visual functions. It receives color and position data from the DefiniteDescription component. This data specifies the color currently being sought and the region of the image in relation to objects of the previously sought color; ‘no relation’ can be specified. The component uses the data to create color blobs for each identified object. Overall, this component corresponds to Color Blob Detection, Visual Attention, and Visual Search in the architecture description.

Blob Tracker components. A ‘Tracker’ is instantiated for each blob identified in the picture. Each Tracker component receives upon creation a blob from the MotionBlobServerNode and attempts to track that blob in successive pictures. Following its creation, the blob tracker receives information about all blobs of the same color with the original blob and attempts to determine which of the current blobs corresponds to the tracked blob. A Tracker can also send commands to the camera, although that feature was not used in this experiment. Thus, a ‘Tracker’ performs Blob Tracking and (partial) Camera Control.

Compound trackers. When a spatial relationship is specified between two objects, e.g., red object on yellow object, specialized components are instantiated, which attempt to track the compound object specified by that relationship. In this experiment, one ‘On’ compound tracker is connected to two other components. In this experiment the two other components are blob trackers; they could be other relational components, such as another ‘On’. Two functions are performed by compound trackers components: Compound Object Tracking and Conceptual Frame, as the are representations within the system of the spatial relations they track.
- Camera controller. The CameraMotionServer component (not used in this experiment) receives motion commands from the trackers in the system and moves the camera based on these commands. The function performed by this component is Camera Control.

For our experimental set-up we started with the attentional focus of the robot in the middle of the screen. The focus of attention shifted as parts of the sentence were incrementally parsed. Thus, for the two-referent set-up, the following scenario unfolds, starting with the architecture depicted in Figure 9.16.

![Figure 9.16. The original state of the reference resolution system - virtual machine view and robot perception](image)

1. PUT. A representation of the verb is created, which in turn creates a referent analysis component (see Figure 9.17).

![Figure 9.17. The state of the reference resolution system after “PUT”](image)

2. THE RED OBJECT. The visual processing component searches the image for red object and instantiates two trackers (see Figure 9.18).
3. **ON THE YELLOW OBJECT.** The visual processing component restricts its search area to those regions directly below the red objects previously identified and searches for yellow objects. A single object is identified, and a tracker is created for that object. Since a spatial relationship exists between a red object and the yellow object, a compound tracker is created, which represents the compound ‘red object on yellow object.’ At this point, since a unique object has been identified, the referent analysis component starts looking for a second unique object, the target of the action. The state of the system at this point is shown in Figure 9.19.

![Figure 9.19](image.png)

**Figure 9.19.** The state of the reference resolution system after “ON THE YELLOW OBJECT”. The object to be moved is identified.

4. **ON THE BLUE OBJECT.** A new visual search is started, with the entire picture forming the potential location. Since there is only one blue object in the picture, the target location is identified, as seen in Figure 9.20.

![Figure 9.20](image.png)
In the one referent set-up, “THE RED OBJECT” uniquely identifies the object to be moved. Thus, the system treats “ON THE YELLOW OBJECT” as the target specifier. However, there is no yellow object which satisfied the precondition of a target, which is that no other object is found on top of it. At this point, the system considers the sentence to be finished and becomes ‘confused’ when presented with the conclusion of the sentence, “ON THE BLUE OBJECT.”

The system’s failure can be resolved through the use of relationship specifiers, such as the word ‘which.’ Thus, the sentence ‘Put the red object which is on the yellow object on the blue object’ is correctly parsed for both the one-referent and two-referent scenarios.

This chapter has presented several examples of the use of APOC and ADE in the design of virtual and robotic agents in domains such as web-searching, robot navigation and simulation of human reference resolution. These examples illustrate the intrinsic power of the system and hint at the potential creation of more complex systems, some of which are already under development.
In the three parts of this thesis we have presented the APOC architecture framework, ADE, the APOC development environment, and considerations and experiments building up to the robowaiter project.

In the first part, we have shown the generality of the framework with examples of how various architectural paradigms can be expressed in APOC. We then illustrated applications of APOC in the analysis and design of agent architectures. The development of hybrid architectural structures from the combination of features from different architectural paradigms was also presented in the context of APOC.

The second part of the thesis presented the ADE tool. The direct connections between APOC theory and ADE implementation were presented, followed by details about the implementation and optimizations used in order to make ADE better suited for complex agent development. Examples of ADE functionality were then presented in the context of robotic agents.

The third part of the thesis focused on the development of a robot waiter. The requirements for the system were outlined, work done in preparation for the upcoming robowaiter development was presented, and the relation between this work and the robowaiter was shown. We conclude with a look at potential future improvements to the system, as well as work currently in progress within the ADE environment.
The first ADE optimization is related to the creation and deletion of components during an agent’s lifetime. Depending on system characteristics, environmental interaction and other factors, it is possible that some computers used by ADE will carry a significantly higher load than others. Automatic load-balancing with dynamic architectures is a goal for ADE.

A related improvement concerns inter-component communication. Since function calls are significantly faster than remote method invocations (RMI calls), the load balancer should attempt to keep components which are connected through a link on the same computer.

The tight integration of ADE with Sun’s JAVA source development kit allows for the definition and compilation of new components within ADE, which are then immediately available for use in the architecture. The implementation of an embedded editor for ADE will also be pursued in summer 2004.

ADE based-applications that are in the works include an implementation of a rule-based system. While a theoretical, high-level translation of rule-based systems exists, an implementation would implement a system such as SOAR at a much finer granularity (e.g., implementing each rule as an ADE component) thus allowing for a much more thorough use of the computational parallelism potential present in our system.

A second ADE project investigates possibilities to improve the performance of the autonomous robotic system described in Section 9.3 by adding optic flow methods, although there seem to be intrinsic computational barriers involved. We are also trying to additional low-cost methods of motion estimation in an effort to add predictive camera movements, which should help in cases where the robot now loses track of an object, because the camera is too slow.

Finally, the robotic experiments presented in this paper (robot navigation and reference resolution) together with related work done in our lab form the foundation for the
robowaiter project. Work is currently in progress towards that the creation of a robotic waiter with a first working version being anticipated by July 2004.
BIBLIOGRAPHY


