

Technische Universität München
Fakultät für Informatik

Report compiled from two Diplom theses

The Barrier
A Prototype for a New HCI Framework

Roland C. Aydin, Robert West

Supervision: Prof. Dr. Alois Knoll, Prof. Dr. Şahin Albayrak
February 2007

Abstract

The Barrier is both the specification of a new HCI framework, and the name of the prototype that has been built adhering to that specification. It separates the user ('environment') from the applications, handling all I/O with the user. Applications receive their commands through the Barrier and also send it any data that is to be output. Moreover, they can communicate among themselves by means of the Barrier. The user, on the other side, can use any input media that support the Barrier's simple API to input requests. Interaction with a computer is less application-centered, but with the Barrier instead becomes request-centered. The Barrier directs user requests to the appropriate application, and outputs data to the user by means of the output media best suited for him/her and for the data displayed. A prototype of the core system has been developed alongside example applications as well as input and output modules, resulting in a running proof-of-concept system. Tests of the prototype have demonstrated this to be a potentially intuitive new way of interacting with computers. The Barrier is implemented in Java, also using XML, and is therefore platform independent.

Acknowledgements

It is difficult to overstate the role of thesis advisor Prof. Alois Knoll. Not only did he significantly contribute to many ideas and technologies used in the Barrier, but he also provided encouragement and, when necessary, enough creative leeway to delve into new endeavors. As a focal point he helped tremendously in sifting ideas throughout the design process, keeping this project on track.

This thesis would not have come to pass without him.

Hence, this project is dedicated to his unit at the Technical University of Munich, ‘Informatik VI: Robotics and Embedded Systems’.

We also thank SomaFM’s Secret Agent radio station for keeping spirits high with a steady stream of music, “for our stylish, mysterious, dangerous life”.¹

¹description at <http://www.somafm.com>

Contents

I. The Barrier	1
1. Introduction	3
1.0.1. Design goals	4
1.0.2. Motivation	5
1.0.3. Secretary metaphor	6
1.0.4. Related disciplines	8
1.0.5. Structure of this document	10
2. Literature survey	13
3. General setup	19
3.0.6. Overview	19
3.0.7. Scenario	21
3.0.8. Technical description	22
II. Left Automaton	25
4. Introduction	27
5. Multimodal input processing	29
5.0.9. Overview	29
5.0.10. Research survey	31
5.0.11. Description	31
5.0.12. Technical description	37
5.1. Grouper	39
5.1.1. Overview	39
5.1.2. Description	40
5.1.3. Development history, alternatives, and venues of improvement	40
5.2. Merger	41
5.2.1. Overview	41
5.2.2. Description	42
5.2.3. Development history, alternatives, and venues of improvement	43
6. Natural-language processing	45
6.0.4. Overview	45
6.0.5. Ambiguity	46

6.0.6.	Description	47
6.0.7.	Development history, alternatives, and venues of improvement	49
6.1.	Grammar	52
6.1.1.	Overview	52
6.1.2.	Research survey	53
6.1.3.	CCG and OpenCCG	54
6.1.4.	Description	58
6.1.5.	Parser	65
6.1.6.	Technical description	65
6.1.7.	Development history, alternatives, and venues of improvement	68
6.2.	Imperatives	69
6.2.1.	Overview	69
6.2.2.	Description	70
6.2.3.	Technical description	72
6.2.4.	Development history, alternatives, and venues of improvement	74
6.3.	Goal Test and Catalog	74
6.3.1.	Overview	74
6.3.2.	Description	74
6.3.3.	Technical description	76
6.3.4.	Development history, alternatives, and venues of improvement	77
6.4.	Matcher	78
6.4.1.	Overview	78
6.4.2.	Description	78
6.4.3.	Scenarios	83
6.4.4.	Technical description	84
6.4.5.	Development history, alternatives, and venues of improvement	86
6.5.	Rulebook	88
6.5.1.	Overview	88
6.5.2.	Description	90
6.5.3.	Scenarios	92
6.5.4.	Development history, alternatives, and venues of improvement	93
7.	Data mode and command mode	105
7.0.5.	Overview	105
7.0.6.	Description	105
7.0.7.	Scenarios	106
7.0.8.	Technical description	107
7.0.9.	Development history, alternatives, and venues of improvement	107
8.	Module management	111
8.0.10.	Overview	111
8.0.11.	Description	111
8.0.12.	Technical description	114
III.	Modules	115
9.	Introduction	117

9.0.13. Overview	117
9.0.14. Technical description	117
10. Example input modules	119
10.1. Speech input	119
10.1.1. Overview	119
10.1.2. Description	120
10.1.3. Technical description	120
10.1.4. Development history, alternatives, and venues of improvement	120
10.2. Clumsyboard	121
10.2.1. Overview	121
10.2.2. Description	122
10.2.3. Scenario	125
10.2.4. Technical description	125
10.3. Keyboard	126
10.3.1. Overview	126
10.3.2. Description	126
10.4. Vision	127
10.4.1. Overview	127
10.4.2. Description	127
10.4.3. Technical description	128
10.4.4. Development history, alternatives, and venues of improvement	130
11. Example output modules	131
11.1. Screen	131
11.1.1. Overview	131
11.1.2. Description	131
11.1.3. Technical description	133
11.1.4. Development history, alternatives, and venues of improvement	133
11.2. Speech output	135
11.2.1. Overview	135
11.2.2. Description	136
11.2.3. Technical description	136
11.2.4. Development history, alternatives, and venues of improvement	136
IV. BSF	137
12. Introduction to BSFs	139
12.0.5. Overview	139
12.0.6. Scenarios	143
12.0.7. Technical description	143
12.0.8. Development history, alternatives, and venues of improvement	144
12.1. Barrier-BSF	144
12.1.1. Overview	144
12.1.2. Description	145
12.1.3. Scenario	147
12.1.4. Technical description	148

12.1.5. Development history, alternatives, and venues of improvement	149
V. Middle Automaton	151
13. Introduction	153
13.0.6. Overview	153
13.0.7. Description	153
14. Request management	157
14.0.8. Overview	157
14.0.9. Description	157
14.0.10. Development history, alternatives, and venues of improvement	158
15. BSF storage	161
15.0.11. Overview	161
15.0.12. Description	161
15.0.13. Development history, alternatives, and venues of improvement	162
16. Unification of BSFs	165
16.0.14. Overview	165
16.0.15. Description	165
16.0.16. Technical description	167
16.0.17. Development history, alternatives, and venues of improvement	168
17. Thresholds	169
17.0.18. Overview	169
17.0.19. Description	169
17.0.20. Supertypes	170
17.0.21. Scenarios	171
17.0.22. Development history, alternatives, and venues of improvement	171
18. Threshold management	173
18.0.23. Overview	173
18.0.24. Description	173
18.0.25. Development history, alternatives, and venues of improvement	175
VI. Right Automaton	177
19. Introduction	179
19.0.26. Overview	179
19.0.27. Description	179
19.0.28. Technical description	181
20. Right Catalog	183
20.0.29. Overview	183
20.0.30. Description	183
20.0.31. Development history, alternatives, and venues of improvement	184

21. Application management	187
21.0.32. Overview	187
21.0.33. Description	187
21.0.34. Technical description	190
21.0.35. Development history, alternatives, and venues of improvement	190
22. Arbitration	193
22.0.36. Overview	193
22.0.37. Description	193
22.0.38. Development history, alternatives, and venues of improvement	193
23. Communication with the applications	197
23.0.39. Overview	197
23.0.40. Description	197
23.0.41. Technical description	198
23.0.42. Development history, alternatives, and venues of improvement	200
VII. Applications	203
24. Introduction	205
24.0.43. Overview	205
24.0.44. Description	205
24.0.45. Technical description	206
25. Example applications	209
25.1. Bobertino	209
25.1.1. Overview	209
25.1.2. Description	209
25.1.3. Technical description	211
25.2. Bmail	211
25.2.1. Overview	211
25.2.2. Description	211
25.2.3. Technical description	213
VIII Miscellaneous and advanced topics	215
26. System configuration and performance	217
26.0.4. Overview	217
26.0.5. Description	217
27. Communication setup	221
27.0.6. Overview	221
27.0.7. Description	222
27.0.8. Scenarios	224
27.0.9. Development history, alternatives, and venues of improvement	225
28. Serialized data	227

28.0.10.Overview	227
28.0.11.Description	227
28.0.12.Technical description	229
28.0.13.Development history, alternatives, and venues of improvement	229
29. Automatic GUI generation	231
29.0.14.Overview	231
29.0.15.Description	231
29.0.16.Scenario	233
29.0.17.Technical description	234
30. Periodic commands	235
30.0.18.Overview	235
30.0.19.Description	235
30.0.20.Technical description	235
30.0.21.Development history, alternatives, and venues of improvement	238
31. Communication among applications	241
31.0.22.Overview	241
31.0.23.Description	241
31.0.24.Scenario	242
32. Multi-Barrier setup	245
32.0.25.Overview	245
32.0.26.Description	246
32.0.27.Scenario	250
32.0.28.Technical description	251
33. Outlook and conclusion	253
User guide	257
Bibliography	263
Glossary	267
Index	271

List of Figures

1.1.	The Barrier is designed to be more permeable than its historic counterpart.	5
1.2.	A scenario without the Barrier in the secretary metaphor.	8
1.3.	A scenario with the Barrier in the secretary metaphor.	9
2.1.	Screenshot of Project Looking Glass.	14
2.2.	Screenshot of a spreadsheet in the Croquet Project.	15
3.1.	Simple view of the Barrier’s general setup.	19
3.2.	Major (not all) data paths through the Barrier.	20
5.1.	One command spread over several modalities.	30
5.2.	Multimodal input processing in the Left Automaton.	32
5.3.	The <code>acceptGroupedWords</code> algorithm.	38
5.4.	The algorithm for computing permutations.	43
5.5.	An example run of the Merger for input $\{1, 2, 3, 4\}$.	44
6.1.	Screenshot of <code>LinguaStream</code> .	54
6.2.	Screenshot of <code>Antelope</code> depicting anaphora resolution.	55
6.3.	The Barrier’s type hierarchy (its ontology).	61
6.4.	An example parse showing how variables are implemented.	65
6.5.	The lexical family of nouns in <code>OpenCCG</code> .	66
6.6.	The lexical family of transitive verbs in <code>OpenCCG</code> .	96
6.7.	The logical form resulting from parsing the sentence “make $\$x1$:virt-thing for $\$x2$:person”.	97
6.8.	The logical form resulting from parsing the sentence “make $\$x1$:virt-thing for $\$x2$:person” without applying the XSL transformation.	98
6.9.	The <code>goalTest</code> algorithm.	99
6.10.	A possible scenario if we use a tree-based instead of a graph-based search algorithm.	100
6.11.	The scenario of figure 6.10 when tackled with a graph-based algorithm.	100
6.12.	The <code>Matcher</code> algorithm.	101
6.13.	An example run of the <code>Matcher</code> .	102

List of Figures

6.14. Logical form of the noun “e-mail”. 102

6.15. The `apply` algorithm for applying a rule to an imperative. 103

6.16. The recursive part `applyRec` of rule application. 103

7.1. The `acceptGroupedWords` algorithm. 108

7.2. The *old* `acceptGroupedWords` algorithm’s synchronization part. 109

9.1. UML class diagram of modules, only major relationships. 118

10.1. The Bayes model used for Clumsyboard. 123

10.2. The Clumsyboard module’s input prompt. 125

10.3. A screenshot of the keyboard module’s input prompt. 127

10.4. Finite state machine describing the vision input module. 128

10.5. A typical input scene for the camera module. 129

11.1. A screenshot of the screen output module. 132

11.2. UML class diagram of the functional parts of the screen output module. . . 134

12.1. Example of a typical BSF header. 140

12.2. Example of the data part of a BSF. 142

16.1. XML structure of composite BSFs (minus the header). 167

18.1. Before/after picture of the “accept output” command. 176

20.1. Part of the old ontology initially planned for use in the Barrier. 185

21.1. Overview of the application management in shape of a UML diagram. . . . 188

21.2. Example of the data part of a registration BSF. 190

23.1. Example of the data part of an imperative BSF. 198

23.2. Logical form representing the word “e-mail”. 199

24.1. UML class diagram for the two applications’ Barrier infrastructure. 207

25.1. A Robertino robot, as controlled by the Bobertino application. 210

25.2. A finite state machine representing Bmail’s behavior. 212

27.1. UML diagram of the communication setup. 223

29.1. Juxtaposition between a normal desktop and one using semantic grouping
enabled by the Barrier. 233

30.1. Periodic commands data flow. 236

31.1. Example of a Barrier-BSF of type <code>imperativeForApplication</code>	242
32.1. A single Barrier.	246
32.2. Two Barriers, showing how a user's command is forwarded to an application of another Barrier.	247
32.3. Many Barriers.	248
33.1. If the gap is too wide for a bridge, build a Barrier as a stepping stone. . . .	254
33.2. A jar.	258
33.3. An example of a configuration file.	259

List of Figures

List of Tables

6.1. The lexical families specified by the Barrier’s grammar.	62
12.1. Mandatory and optional parts of the BSF header.	140
12.2. Exhaustive list of all Barrier-BSFs.	145
16.1. Examples of the two kinds of unification rules.	165
18.1. Exhaustive list of ‘threshold imperatives’ that explicitly influence thresholds.	174
28.1. Serialization in the Barrier: classes, files, and usage.	228

List of Tables

Part I.

The Barrier

1. Introduction

Many authors have insisted that man is divided by an insuperable barrier from all the lower animals [...].

—Charles Darwin, *The Descent of Man, and Selection in Relation to Sex*

The Barrier is a system whose use it is, simply put, to separate the user from the applications. This can solve a lot of potential misunderstandings in both directions, supposedly resulting in a more intuitive manner of interacting with a computer.

The user, in our system, is free to use whatever input media available to express her desires to the Barrier. The Barrier, then, will communicate these desires to the appropriate applications. Hence, applications can automatically make use of all available input media, getting input from them without necessarily even knowing they exist. Application programmers only have to satisfy the simple API the Barrier mandates, and can then entirely forget about the intricacies of user interaction, GUIs, speech recognition, vision recognition etc. Instead, they are now free to focus entirely on the actual functionality of their program, leaving user interaction to the Barrier. Even more so, given that many programs were to subscribe to the Barrier, all could be assured of a certain quality, standard and ‘look and feel’. Quality of Service agreements and quality control worries could be substantially alleviated. In addition, the user would get used to a homogeneous interface experience, regardless of how many or which programs are controlled by it. This can be a boon also for the applications, as they would spend less resources in tutoring or training the user in the use of their program.

The user can benefit just the same. Ideally, a user of the Barrier will never have to worry about what programs are processing her requests. Instead, she simply uses functionalities taken from across many applications, not having to learn by rote nor memorize what each application can do individually. All the user perceives in our system are the combined capabilities of all programs, displayed in a uniform way.

When a new program is installed, then, the user experience would not change at all. All that would be discernible for the user would be the appearance of new commands offered

1. Introduction

by the Barrier. Finally, the user would keep all configurations, which would be applied to the new application as well (where applicable). For instance, if the user had trained the Barrier to display pictures in a certain way on a certain screen, these settings would automatically be functioning for pictures coming from the new application as well, without that application even knowing.

It is conceded that, normally, introducing additional layers in a model create additional complexities. Consequently, each layer must be well justified. With the Barrier, it can be argued that the additional layer in fact *reduces* complexity. For with the Barrier in between, the user and the application layers are completely decoupled, technical details are shielded from the user, while user peculiarities and juggling different media (in short, the whole I/O) are shielded from the application programmer.

Particular highlights of this paradigm are that

- it enables applications to exchange data without knowing each other,
- GUI generation is entirely taken off the application programmer's burden, including significant portions of dialog management, and that
- all available input and output media are implicitly used by all applications, without them even knowing.

1.0.1. Design goals

These are the Barrier's general design goals:

1. Allow for a *natural way of communication* between user and computing system.
2. Make the computer system *appear* to be designed fully adhering to *user-centered design*.
3. Nonetheless enable the programmer *not to adhere to user-centered design*.
4. Support a dynamically *extensible* set of numerous *input and output media*.
5. With a *robust specification*, develop a *functioning prototype* using state-of-the-art technologies and design principles.
6. Develop *proof-of-concept applications and input/output modules* to demonstrate the essential working principles.

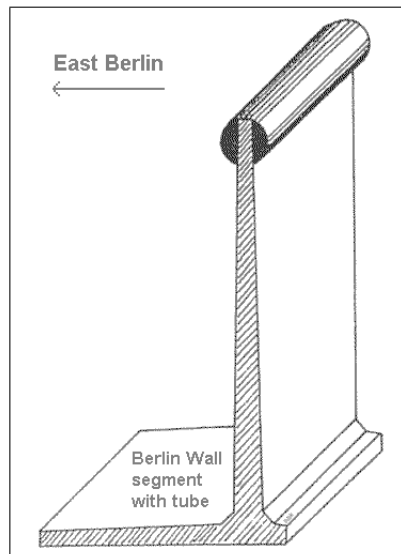


Figure 1.1.: The Barrier is designed to be more permeable than its historic counterpart.

Source: <http://www.berlinermaueronline.de>

1.0.2. Motivation

The first idea for a Barrier-like system arose when work was done on yet another GUI for yet another application. Why, the question arose, was it necessary to continue developing windows with sufficient eye-candy for each and every application *ad nauseam*? Why, even then, was the user usually malcontent with how data was presented to her? And, most of all, why could not somebody else do the work and deal with such problems for the John Doe application programmer?

It turned out somebody else just might. From the requirements stated above and the faint notion that techno-savvy application programmers who do not want to care for not-so-savvy users need to be *separated* from each other came the structure of a *barrier*. Its programmers would eventually do all the direct communication with the user, leaving programs free to focus on their actual functionality, i.e. the services they provide—instead of fancy window-dressing.

Over time it was realized that this put other benefits in reach as well—such as programs which could communicate amongst themselves by natural language phrases, without even knowing each other.

Also, not all users want to have all data displayed the same way: e.g. some might prefer their e-mails read out aloud, some might be more comfortable with reading them on-screen. Such individualities, so the idea, could be coped with in one central instance as well, by learning a user's preferences.

1. Introduction

Most importantly, this de-coupling of application and user enabled users to work and think *task-oriented* instead of *program-oriented*. That means that users do not have to memorize what each program can do, but instead simply hand out commands to the Barrier, which then distributes it to the system.

This decoupling seems to make particular sense since the era of ubiquitous computing [Gre06], i.e. the integration of computation into the environment. It is not evident why such a multitude of computers would each need to have its own sensible user interface, or why, with such pervasive computing, the user should be forced to get to know the user interface of every last gadget hiding in the room. Instead, once ubiquitous computing pervades home computing, and “you walk into a room, and something happens in response” [Gre06, page 38], with the Barrier the user would be able to centrally control all these gadgets that are networked and compatible with the Barrier while enjoying the same user interface experience she always does, even with applications/gadgets having been changed. In that case, the available commands would change, while the user experience would not.

There is another, maybe one of the most vital aspects of what the Barrier might enable: Handicapped users, who were previously largely excluded from the IT revolution, could be seamlessly included. They were mostly excluded because application programmers did not have the budget to cater to their demands. For example, few applications can still be handled by a blind user.

With the Barrier, however, applications are fully disconnected from the media they use. If a handicapped user is able to communicate and receive data over any one media (module) supported by the Barrier, he will be able to control and receive data from *all* applications using the Barrier. Just imagine a module parsing sign language; the applications would not notice any difference. The social and ethic advantages of such an inclusive system, i.e. including users with disabilities, are obvious.

Realizing these considerations above was the main motivation for creating the Barrier.

1.0.3. Secretary metaphor

To illustrate what exactly the Barrier aims to accomplish, let us consider the current and the envisioned situation by means of a metaphor. For that purpose, we introduce what we call the ‘secretary metaphor’.

Figure 1.2 depicts, in a pointed manner, the current situation. The user, in this case the businessman in the middle, wants to interact with his computer system. That system consists of applications, which each supply certain services. Those are represented by the clerks surrounding him.

The problems as we perceive them to be are obvious. For one, the businessman does not know whom to talk to for a certain service. He himself must pick one of the clerks. If he does not know who to talk to (i.e. if he does not know which application provides that service), he is essentially left at a loss.

The applications, since they have to be in direct contact with the user, are forced to ‘dress up’, in the metaphor indicated by their ties. Their attention is necessarily focused on the user, and hence the energy they can spend on their core tasks is reduced. Given that the clerks only speak their technical lingo (i.e. applications are inherently technical artifacts), whereas the customer is only fluent in business language (i.e. the user normally only speaks natural language), one of the two has to adapt to the other: either the client has to learn the technical slang or the clerks have to be ‘socialized’ in their behavior, which is inconveniencing in both cases.¹

Also, since they are not organized in any central manner, should they want to talk to each other, chaos could ensue, since they, too, would not know who to talk to, and what services their fellow colleagues each supply.

In short, question marks prevail. Such an organizational pattern—and this is how we perceive the current situation in desktop computing to be—would be akin to a large company building without a receptionist, where clients just wander around the building, hoping to find someone to help them on their errands.

In contrast, figure 1.3 depicts how our solution, the Barrier, aims to tackle these shortcomings by installing such a receptionist. The client, then, has a central contact point for all interactions with that company: the receptionist, who symbolizes the Barrier. To stay in the metaphor, she is not a ‘techie’, but instead a customer relations professional. She is specialized both in handling the client’s requests and in communicating them to the appropriate clerk inside the company. Only that clerk is then notified of the request, while his colleagues (read: other applications) are free to continue their tasks. They do not have to expect a visit from a client at all times. This is visualized by them dressing casually and looking at their desks: customer relations is fully factored out away from them.

Vice versa, when the clerk has finished a technical memo for the client (who is certain not to understand it), the secretary will relay that information to the client in a way the client can cope with. She will memorize whether that client wants to have his memos presented in a certain way and whether he is interested in that memo at all.

Finally, a clerk will also use the secretary’s services himself to communicate with his fellow colleagues. If he has a request, just as if he were a client, the secretary will forward it to a fitting colleague, even be those two unbeknownst to each other. For example, in one

¹At least when remembering that clerks are in fact just metaphors for applications.

1. Introduction

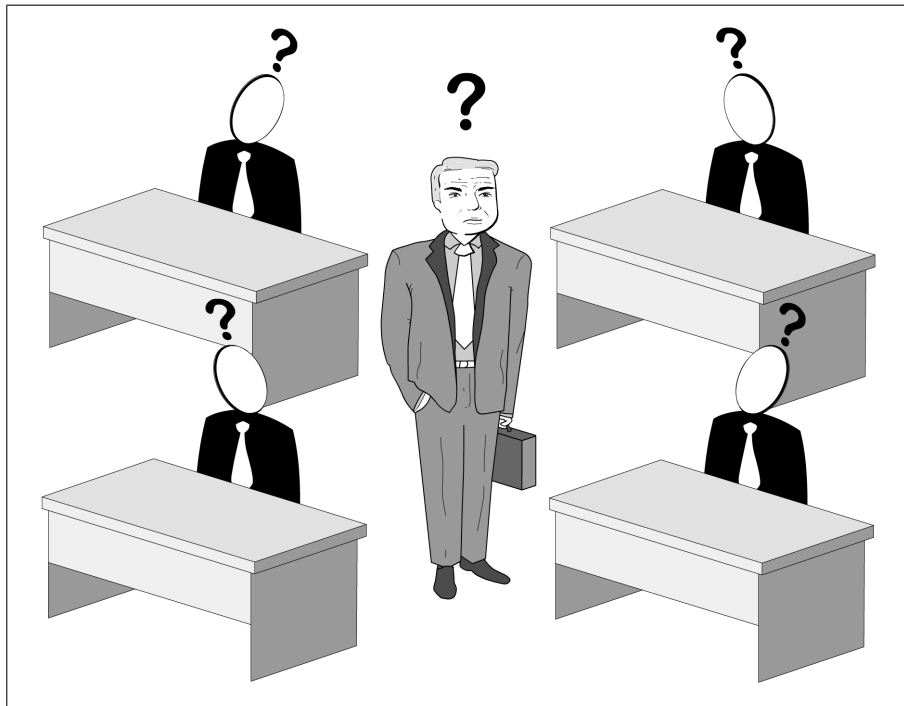


Figure 1.2.: A scenario without the Barrier in the secretary metaphor.

Illustration: Megan Jack, 2007

clerk's office the heating is broken; the clerk will now just tell the secretary, who in turn will notify building maintenance. The clerk with the broken heater, then, will neither have to talk to the janitor himself, nor to even have his phone number.

There are other aspects which will only be explained further on in this paper, since they would have overloaded the metaphor. Hence, while the metaphor should be kept in mind, it is in no way feature complete.

1.0.4. Related disciplines

Because of the scope of the Barrier, and the requirement of providing a working prototype, technologies from a big many disciplines and areas of computer science had to be used. These include, but are not limited to:

1. Platform-independent data structures (\rightarrow XML),
2. keeping the Barrier's state persistent over many sessions (\rightarrow serializing data and storing it on the hard-drive),
3. allowing for arbitrary manipulations on data structures in a well-defined way (\rightarrow Grammars, XML Schema, XSLT),

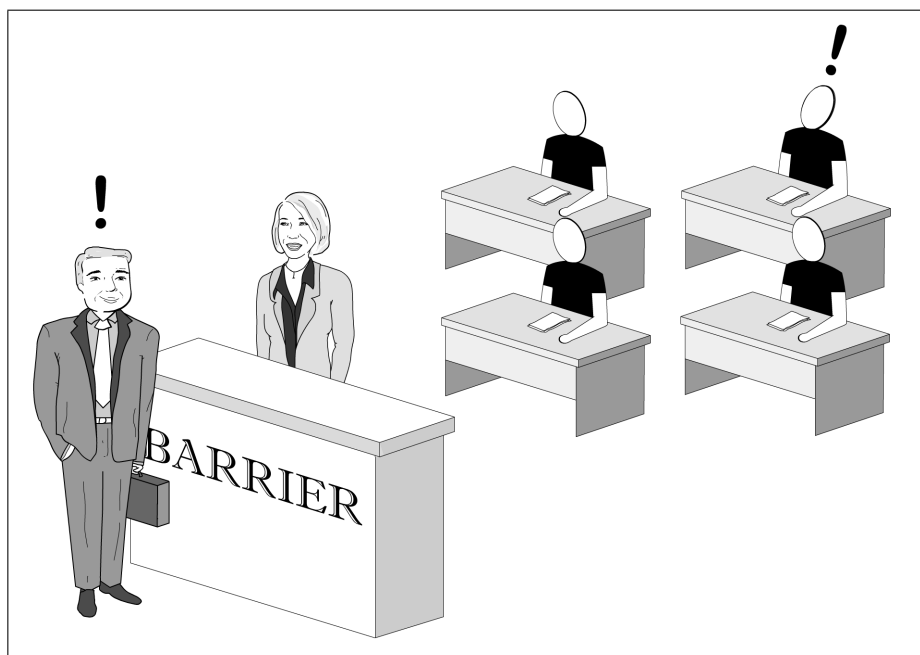


Figure 1.3.: A scenario with the Barrier in the secretary metaphor.

Illustration: Megan Jack, 2007

4. keeping the Barrier's code as platform-independent as possible (\rightarrow Java),
5. providing graphical user interfaces (GUIs) for the screen output module (\rightarrow Swing),
6. incorporating image processing techniques for the vision input module (\rightarrow Java Advanced Imaging, histograms, color spaces),
7. parsing natural language expressions (\rightarrow OpenCCG),
8. matching parsed expressions against a Catalog, including variables, macros and translations (\rightarrow graph theory),
9. translating audio signals into strings for the speech input module (\rightarrow Dragon Naturally-Speaking),
10. reading aloud text for the speech output module (\rightarrow FreeTTS),
11. programming example applications for a robot and an e-mail client (\rightarrow Robotino API, Java Mail),
12. providing a distributed client/server architecture (\rightarrow socket programming),
13. providing as much parallelism for efficiency and scalability as possible (\rightarrow thread programming), and
14. using AI methods for learning parameters for the 'Clumsyboard' input module (\rightarrow

1. Introduction

Bayes nets).

This list lacks most of the actual problems and tasks in specifying the Barrier's behavior and the rather big three Automata it consists of. The code base of the entire Barrier project hovers at around 11,000 lines of code, excluding XSDs and some peripheral elements.

1.0.5. Structure of this document

Mostly, each section will describe either one part of the Barrier, such as the Right Automaton, or one particularly important algorithm, such as unification of BSFs, which can well be referred to in multiple parts. Some of the information may appear to be reiterated when necessary for better understanding. Most sections conform to the following structure,

1. Overview: this subsection will give an overview of what is about to be explained.
2. Description: an in-depth description which focuses on the *structural* and/or *algorithmic* aspects, and does not go into technical implementation details.
3. Scenarios: provides one or more example scenarios that demonstrate either
 - a) the normal case, mostly if the description is too complex to understand without providing a factual example, or
 - b) the extreme case, to demonstrate how this part/algorithm handles border cases.
4. Technical description: this subsection will *not* reiterate the description, but rather go into technical implementation details, such as what and how a technology was used to implement and realize the description. This will *only* be present if there were intriguing or noteworthy details arising in the actual coding. Otherwise, it can be assumed that the Java/XML coding does not warrant a description apart from the comments that are embedded in the source code itself, which, of course, does not mean there were no intricacies.
5. Development history, alternatives, and venues of improvement: this subsection will look back on how the part/algorithm evolved, what design choices were made, and how it could be improved. Due to the modular nature of the Barrier, this could very well include a proposal to exchange the part/algorithm in question with a more sophisticated one, state that performance may be poor in some special cases or list disadvantages to another solution which was also pondered. Those notwithstanding, *all* parts fill their designed role at least adequately.

In the above taxonomy, the first two subsections will most certainly be present, while the other three may be missing if deemed unnecessary or if their content proved to be rather

uneventful.

Important note: The ‘technical description’ subsection will most certainly be *specific to the prototype implementation*, not to the Barrier *specification per se*. In other terms, other software prototypes in other programming languages could just as well implement the Barrier’s structure, adhering to the algorithms and principles laid out in this document. They could, however, very conceivably use wholly different technologies to achieve that goal, and still be valid ‘Barriers’ all the same. The technical description, therefore, while being closer to the actual software coding process, should not be taken to explain the only, or even best technological solution to implement the task as outlined in each chapter.

1. *Introduction*

2. Literature survey

The Barrier is not *middleware*, as defined by [Ber96], as its purpose is not mainly defined as solving interapplicability and connectivity issues.

It is also unaffiliated with CORBA, whose purpose it is to “allow software objects to talk to each other across a network in a well defined way” [LS02, page 2]. The Barrier’s realm is on a much higher level.

Other approaches trying to solve problem sets more similar to the Barrier’s include ‘Archy’. Formerly called ‘The Humane Environment’, it is designed chiefly by Jef Raskin, who held considerable influence at Apple Computer, Inc. It follows closely the user interface paradigm he developed in [Ras00]. While he advocates several interesting aspects, such as that there should only be one way to accomplish an action (instead of redundancy), or that modal features of current graphical user interfaces like windows and separate software applications are to be removed, the emphasis again is neither on the actual application programmer, whose task should be facilitated as much as possible, nor on using multiple input and output media (at the present state).

The Barrier’s approach of letting the programmer ‘skip’ the user interface part is further justified by [CR01], who states that “by considering system functionality first [instead of user interface design], designers can make progress more quickly—they can focus on *what* a system will do, and not worry about how at the same time.” [CR01, page 81]

Handling modern systems is often subject to a large ‘gulf of evaluation’, which is the cognitive distance, i.e. the amount of thinking required, to understand system output. Dan Olsen writes that it will increase significantly when “items may be ineffectively grouped so that the user does not see an important relationship. The critical information may be incorrectly placed so that a quick scan of the display completely misses what is needed” [Ols98, page 13]. Jakob Nielsen goes so far as to advocate ‘natural dialog’, stating that “user interfaces should be simplified as much as possible, since every additional feature or item of information on a screen is one more thing to learn, one more thing to possibly misunderstand” [Nie93, page 115]. The Barrier’s design follows such a doctrine, getting rid of the superfluous association of ‘functionality/program name’, and predominantly exposing the user only to system data which she has previously accepted.

2. Literature survey

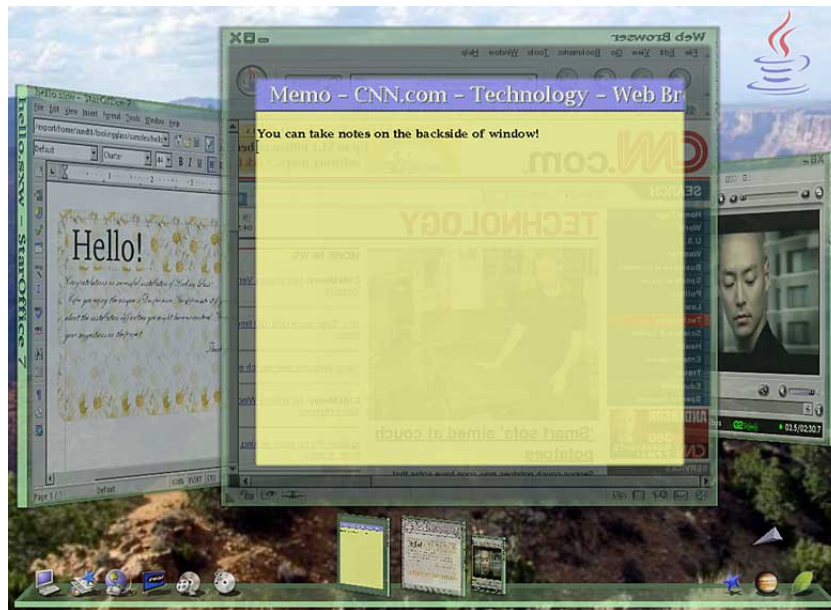


Figure 2.1.: Screenshot of Project Looking Glass.

Source: http://www.sun.com/software/looking_glass/details.html

What the Barrier does aim to provide to the user can be considered the affordance of goal-directed multimodal dialog as described in [KG01], in effect *upgrading* applications which hitherto did not support more sophisticated input media such as gesture or facial expressions.¹

While the Barrier shares certain similarities with the ‘task-oriented functional design’ methodology outlined in the same book, a strong emphasis is on the way applications and their programmers are also offered more conveniences, by having part of their duties taken away.

A platform that “supports running unmodified existing applications in a 3D space, as well as APIs for 3D window manager and application development”² is ‘Project Looking Glass’, developed under the auspices of Sun Microsystems, Inc. Already usable, it uses the GUI that existing applications already come with, and improves on them by adding a better look, and more affordances such as the ability to write on the ‘back’ of a window, as shown in picture 2.1.

The Barrier, while not able to run unmodified applications, does not need a pre-existing user interface to ‘boost’ artificially. Rather, its benefits can be seen that when designing for the Barrier in the first place, the whole of I/O takes less effort, and can be redelegated to a much more technological level (i.e. not adapted to the actual user).

¹The latter is not included in the example prototype, but would integrate seamlessly.

²http://www.sun.com/software/looking_glass/

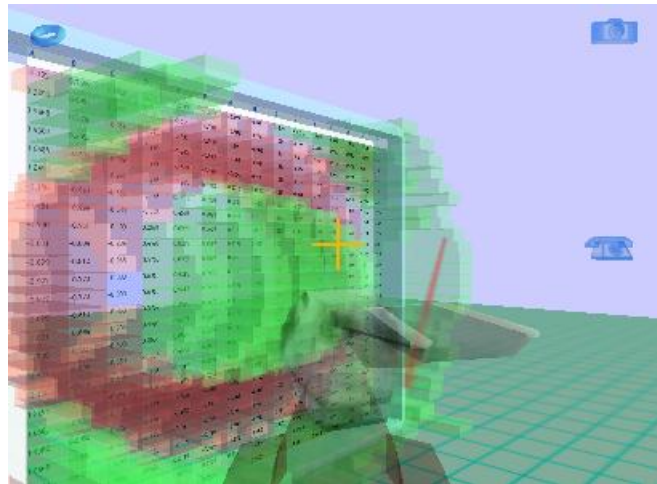


Figure 2.2.: Screenshot of a spreadsheet in the Croquet Project.

Source: http://upload.wikimedia.org/wikipedia/en/7/75/Croquet_Spreadsheet.jpeg

Another such platform was developed principally by Seymour Papert, Alan Key, David Smith, and David Reed: The ‘Croquet Project’ also aims to provide a new HCI framework, and is more than a simple new GUI. Instead, it also changes the application development process by including collaborative elements taken from the educational technology branch of computer science. Mainly, these are PBL (project-based learning) and Constructivism. Both are said to increase productivity. While not being in widespread use, the Croquet Project is advocated mainly by an alliance of academic institutions [KRSR03].

While the Barrier has more in common with the goals of the Croquet Project, it does not emphasize the collaboration of users as strongly as that project. In fact, the prototype is meant to be used only by a single user, and multi-user viability remains to be tested in the future. Picture 2.2 shows a screenshot of a spreadsheet displayed with Croquet. As cannot be inferred from that screenshot, Croquet aims to let the user construct a landscape, and to fill it with objects such as that spreadsheet. This uses a human’s predisposition to memorize information along *topological* lines.

‘Rules’ and Benchmarks in HCI

In the field of usability engineering, several important benchmarks have arisen:

Hick’s law

Psychologist William Hick developed an important measure of the rate at which additional choices hinder a user. In [Hic52], the average reaction time T that a user needs in order

2. Literature survey

to make one of n decisions is given as

$$T = b \log_2(n + 1),$$

where b is a constant that can be determined empirically.

Note that this logarithmic dependency only holds if the choices are somehow, e.g. alphabetically or semantically, *ordered*. If this is not the case the time grows linearly with the number of choices.

This rather important discovery in experimental psychology still has a severe impact on usability engineering, as it quantifies a relationship which already seemed obvious: additional choices slow down the user, and only in a best-case scenario (see above) does the time for making a decision increase merely logarithmically. It follows yet again that the user should only be given the data she needs, with an option to request additional data.

As previously described, this further justifies the approach undertaken by the Barrier, as when the number n of choices decreases (as choices are displayed potentially purely³ by semantic selection, not based on what a specific application can do⁴), the user needs less time T to make a decision.

GOMS

According Allen Newell in [CMN83, page 140], a GOMS Model consists of

1. a set of **G**oals,
2. a set of **O**perators (used to reach a goal),
3. a set of **M**ethods for achieving the goals (sequence of operators), and
4. a set of **S**election rules for choosing among competing methods for goals.

Interactions that a user has with a computer system are decomposed into these categories.

Although it has some clear drawbacks, such that typical GOMS analyses only consider expert users, it has evolved (with its variants) into an important taxonomy that helps evaluate user interfaces.

For the applicability of using GOMS to evaluate the Barrier it is important to realize that it is *not* to provide über-sophisticated input/output modules, but rather to provide a framework with a fully working prototype that has proof-of-concept modules that can

³although not realized in the prototype, but the infrastructure fully exists

⁴this is realized in the prototype, as such superfluous information are not even available to an output module

be improved arbitrarily from a pure usability perspective, *without* endangering any of the Barrier's workings. It can, however, be stated with confidence, that the Barrier's purpose is exactly that *once* these modules are capable of satisfactory I/O based on a GOMS evaluation, that advantage will apply to *all* applications supporting the Barrier, without the applications even noticing a change.

The Wizard of Oz experiment

First conducted by John Kelley for his dissertation at Johns Hopkins University in 1980, the Oz paradigm still is used for usability testing, and when designing new user interfaces [Ake06].

As described in [Kel83], its premise is to have a human interact with a supposedly autonomous computer system, which is in fact secretly controlled by another human.⁵ Through this simulation, the developer of a user interface is actively forced to simulate step-by-step his proposed system. Repeating this experiment with a wide range of users can help validate the supposed user behavior with the behavior of actual users, and accordingly adapt the system *before* too many resources were expended into the actual implementation.

⁵Note how this experiment is something like the *inverse of Turing's test*, in which a computer must trick the test subject into believing it is in fact a human.

2. Literature survey

3. General setup

3.0.6. Overview

As shown in the extremely cursory figure 3.1, the *Barrier* is itself subdivided into *partes tres*:

- the ‘*left*’ sphere, which is closest to the user, and handles most of the *interaction with the user*,
- the ‘*middle*’ sphere, which stores most of the state representation and does most *data operations*, and
- the ‘*right*’ sphere, which is closest to the application and handles all *connections to the system*, i.e. the applications.

Each of these three spheres is realized as a large module, which is called an *Automaton*. A view which is more detailed than figure 3.1 is figure 3.2.

The Barrier is surrounded as follows.

- **Input modules** pass the user’s intent on to the Barrier, i.e. to its left sphere (Left Automaton) and **output modules** forward application data to the user.
- Then comes the **Barrier** itself, whose structure was described in the previous enumeration.

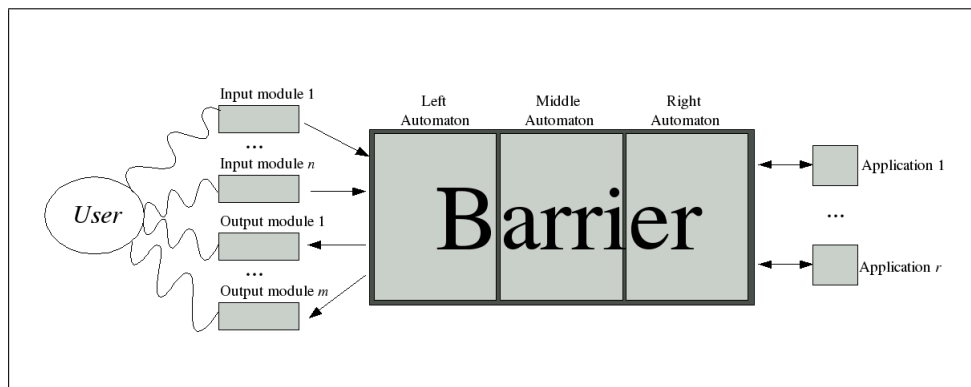


Figure 3.1.: Simple view of the Barrier’s general setup.

3. General setup

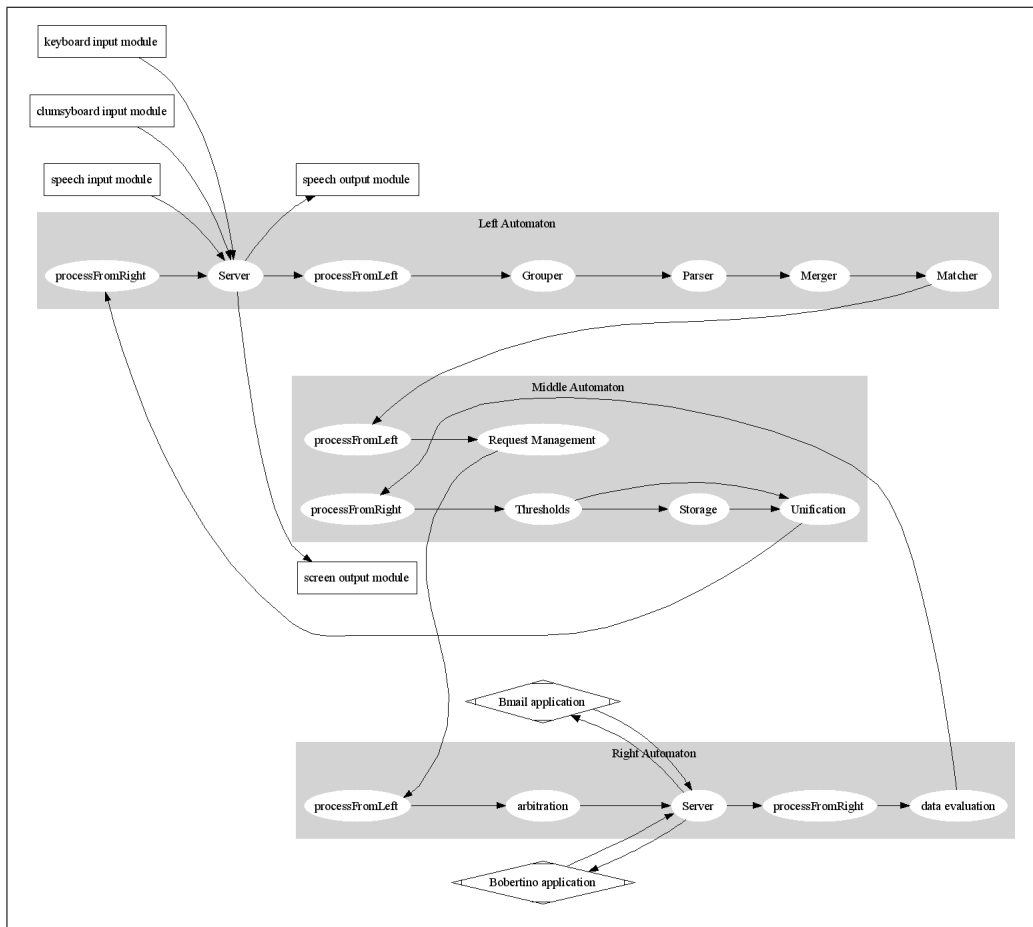


Figure 3.2.: Major (not all) data paths through the Barrier.

- Then come the *applications*, i.e. programs that support and *connect to the Barrier*, i.e. to its right sphere (Right Automaton).

An input module usually controls one input media, e.g. camera, keyboard, etc., while an output module usually controls one output media, e.g. speech, vision etc.

The communication between the Barrier and these surrounding parts is done in a special data format we call *Barrier Structured Format* (BSF; specified as dedicated XML documents): e.g. applications wrap their output into BSFs before sending it to the Barrier.

3.0.7. Scenario

The following scenario shows merely one interaction among many, and is meant just to give an initial impression of how the Barrier processes its input. Other major interactions would be communication among the applications or applications sending data without prompt. Many minor parts are left out.

The Barrier prototype is set up as pictured figures 3.1, or slightly less opaque in figure 3.2. When the user makes an ‘utterance’, i.e. issues a command, an input module (or several of them) will convert that utterance into a string, and pass it to the Left Automaton’s server.

Once arrived in the Barrier, the string(s) will be given to the `processFromLeft` method of the Left Automaton (chapter 4), which will initiate processing of the string(s). In a first step, the Grouper (section 5.1) constructs groups of strings that are thought to belong together, and feeds them to the Parser (section 6.1), which will convert the string into a parse tree (if possible). Those will then be attempted to be matched against a Catalog (section 6.3) of known services by the Matcher (section 6.4), after which the Barrier will have arrived at one or more executable commands, based on the initial user input. The commands will then be passed on to the Middle Automaton (chapter 13).

There, as part of the request management (chapter 14), the Middle Automaton will memorize the request as to be able to correctly process any future answer from an application. Control will then pass to the Right Automaton (chapter 19).

The Right Automaton, given the command, now makes a decision as to which application to ‘entrust’ with the user command. That happens in a process called arbitration (chapter 22). If necessary, i.e. if no apt application is running yet, the Right Automaton can also start an application that is able to process the command. Once that decision has been made, the command is given to the Right Automaton’s server, which then communicates it to the application (which might reside on another computer, cf. chapter 23).

3. General setup

Still in the same scenario, some minutes later the application finally answers the command that it was passed, and aims to send data to the user. That data arrives in the Right Automaton's server, since the Right Automaton is 'closest' to the applications. It will conduct a first 'data evaluation', which mostly means tagging the data correctly (chapter 19), plus updating the status of open requests, and then pass it to the Middle Automaton.

There, a choice is made on whether the data is let through to the user. In this case, it is, since it is in reply to a user command. Also, based on the previous history with the specific user, it will be decided by what media the data will be output, i.e. to which output module it is to be send. This is done in the thresholds algorithm (chapters 17 and 18), which also implicitly adapts the system to user preferences. If the data had been blocked, it would have ended up in the storage (chapter 15). Unification is bypassed in this scenario, for details on what unification is, confer chapter 16. The data is then passed to the Left Automaton.

In the Left Automaton, finally, the data is given to the output module previously determined, leaving the Barrier through a socket in the Left Automaton.

Once in the output module, the data is output to the user dependent on how exactly that output module is implemented.

This concludes the example scenario of the user given a command, an application processing it and answering back with data.

Note that the Barrier only consists of the three Automata, all the modules and applications are peripherals.

3.0.8. Technical description

Each Automaton provides two major methods as its API:

1. `processFromLeft`, which is called when something enters that Automaton from the 'left' side to be processed, and
2. `processFromRight`, which is called when something enters that Automaton from the 'right' side to be processed.

To trace the Barrier working, a short taxonomy of what threads are actually at work is needed:

The `processFromLeft` and `processFromRight` methods cannot be called *sequentially* by an olympic instance because they are called from within different threads after arriving over the socket. It is not the Barrier's main method that calls those methods, it happens

spontaneously whenever something arrives at the Left Automaton's server: Hence, one method must call the equivalent one of its neighboring Automaton (e.g. the Left Automaton's `processFromLeft` calls the Middle Automaton's `processFromLeft`, which in turn calls the Right Automaton's `processFromLeft`). The methods do not return values; in case something is wrong they throw exceptions instead.

If a method does *not* want to pass anything further in 'its own' direction it may just 'turn around': e.g., if the Middle Automaton's `processFromLeft` has found out that a command can be answered from the BSF buffer without passing it to the application first, it calls the Left Automaton's `processFromRight` instead of the Right Automaton's `processFromLeft`.

Thread architecture

As to what threads are active *inside* the Barrier, cf. as an example the direction from 'left' to 'right':

1. Input is accepted by the Left Automaton's server from different input modules.
2. For every input from a module a thread is spawned; each thread simply puts the input into the Left Automaton's input buffer; the input buffer is there to join words that have arrived from different modules within a short interval into one single user command.
3. That buffer runs in a thread of its own; this is the one thread that will carry the input rightwards to the Right Automaton's sender, where it will be flushed over the socket.
4. So the general rule is: "One user command, one thread."

The converse direction, which is from 'right' to 'left' functions analogously (thread-wise, not functionality-wise).

3. *General setup*

Part II.

Left Automaton

4. Introduction

Die Geschichte mit dem Automat hatte tief in ihrer Seele Wurzel gefaßt, [...]

—E. T. A. Hoffmann, *Der Sandmann*

The Left Automaton functions as the conduit to the user. More precisely, it handles the *incoming commands* from the user, as well as the *outgoing data*.

Concerning the incoming commands, it receives strings from input modules that represent events in the environment those modules have recognized, and has to recombine and then make sense of them, in order to eventually pass them deeper into the Barrier (to the Middle Automaton).

With the outgoing data, the task is simpler: Data that is arriving at the Left Automaton has to be forwarded to the appropriate output module.

In addition to these tasks, the Left Automaton has to handle the management of input modules, pass on Barrier-specific output to the user (such as error BSFs) and/or control the output modules (i.e. by commanding them to stop data output).

4. *Introduction*

5. Multimodal input processing

5.0.9. Overview

The primary goal of the Barrier is to provide computer users with a natural way of expressing wishes to the computing system. What do we mean when we say ‘natural way’? We mean a way that is similar to the manner in which the user would communicate with a fellow human; a manner in which her primary thought is not *how* to formulate the wish in order to make it understandable to the computer, but rather *what* to want in the first place.

Most of the time people exchange their thoughts by talking, but there are dozens of other *communication ‘channels’*; to name but a few:

- *Speech*, as mentioned;
- *gestures* are often more compact than spoken words;
- *mimics* can sometimes say more than a thousand words;
- and from time to time, even *writing* might be a natural way of informing others about our ideas, e.g. when the material is complicated, such as formulæ, or hard to understand in spoken words, such as foreign vocabulary.

The word ‘body *language*’ encapsulates all this: we do not speak with our voice only, let alone with our typing fingers, but with all of our body (often subconsciously, without realizing it). Additionally, humans do not make exclusive use of these modalities; they are not restricted to using only one at a time, but rather continuously employ a whole band of communicative techniques simultaneously. Some ideas are best expressed by one specific out of a host of techniques, and some ideas can best be formulated by making use of several modalities at once. It can be seen how important this diversity is when we consider the difficulties some people have talking on the phone; phone calls simply lack the variety of modalities we have at our disposition when talking to a physically near person.¹

¹Minsky, when discussing a similar topic in his book ‘The Emotion Machine’, makes the additional note that face-to-face contact, as opposed to speaking on the phone, not only opens more communication channels, but also is a potential source of confusion: “What if the stranger you have just met should resemble (in manner or facial appearance) some trusted friend or some enemy?” [Min06, page 172]

5. Multimodal input processing



Figure 5.1.: One command spread over several modalities.

Source: A. Franquin. *Idées noires. L'intégrale*. Fluide Glacial, 2001; page 24.

When we use multiple modalities in parallel there are several reasons for doing so:

- a. Sometimes we spread the semantics over several modalities; this is what a sergeant does when he is telling his platoon to “charge!” while he is pointing into the intended direction with his arm (cf. figure 5.1). In that case, his intended will can only be understood by merging both modalities.
- b. Sometimes, on the other hand, we use redundancies to make sure the listener understands us, to make our words more interesting so others attend to what we say, and to rule out ambiguities; e.g., a good speaker is expected to emphasize his points with body language.

Given such considerations, it was indispensable to build into the system ways of merging input signals from different modalities. Otherwise, the user would not be enabled to communicate with the computer system in a *natural way*, i.e. using multiple modalities. While humans have in fact become used to being ‘restricted’ in how they interact with a computer (using only body language, i.e. their hands), that restriction was imposed by long gone hardware inadequacy, and should be lifted. This is why we introduced the concept of input modules: we want to allow for any variety of input sensors as long as it preprocesses its data to produce a form that can be further ‘digested’ by the Barrier.

It is worth noting that speaking of a natural way of communication implies that both partners should make use of natural—including diverse—ways of uttering their thoughts. Certain data types are better suited for certain ways of being communicated. This is why the Barrier allows not only for a multitude of input but also for an extendible set of output modules. This will be discussed in detail in the chapter about output modules.

Then you might perceive different things in what the other one says than what he actually intends to say, because you have been primed by that physically similar person.

5.0.10. Research survey

Intensive research is currently done in the field of multimodal user interfaces: “Today an increasing effort is made towards multimodal communication; this is an extension of natural-language communication by means of, e.g., pictures all the way to gestures and scents.”² [DG02, page 984]

An exhaustive overview can, e.g., be found in the collection “Readings in Intelligent User Interfaces” [MW98]. In a more concrete article, the same authors propose an abstract system architecture and a road map [BKMW05]; [BKMW05, figure 4, page 328] conveys a good idea of that project’s modular structure. Certainly, some valuable ideas might be taken from there in the future to improve on the Barrier’s multimodality in the versions to come.

When tackling this problem in more depth, one must not only look for solutions in the realm of computer science; a solution is not viable “without a deeper understanding of linguistics and human psychology” [KG01, page 1]. In the same paper [KG01], Knoll and Glöckner explain two fundamental approaches to communicating a command to a computing system: the ‘*front-end*’ vs. the ‘*incremental*’ approach. In the former, a task is specified once in a single modality, whereas in the latter, ‘sub-instructions’ may be added step-by-step in increasing granularity, possibly over a multitude of modalities.

The Barrier follows the incremental approach in that it allows for multimodal input, yet it could vastly benefit from a *general* framework that would enable the user to incrementally refine her wishes to the computing system. A proof-of-concept for the *specific* domain of assembling toy airplanes and the like with “Baufix” building blocks has been achieved in the context of the JAST³ project [FBRK06].

5.0.11. Description

In the pages to come we will dwell upon how the Left Automaton deals with data originating from several input modules.

Here is a rough overview of the process (see also the diagram of figure 5.2):

1. Different *input* ‘snippets’ (w_1, w_2, w_3 in the picture) arrive from different input modules (of course there will also be the case of only one input module being involved);

²Original German quotation: “Heute wird verstärkt in Richtung multimodaler Kommunikation gearbeitet; das ist eine Ergänzung der natürlichsprachigen Kommunikation mit z.B. Bildern bis hin zu Gesten und Gerüchen.”

³JAST stands for Joint-Action Science and Technology; for details, cf. the project website at <http://atknoll11.informatik.tu-muenchen.de:8080/tum6/research/JAST>

5. Multimodal input processing

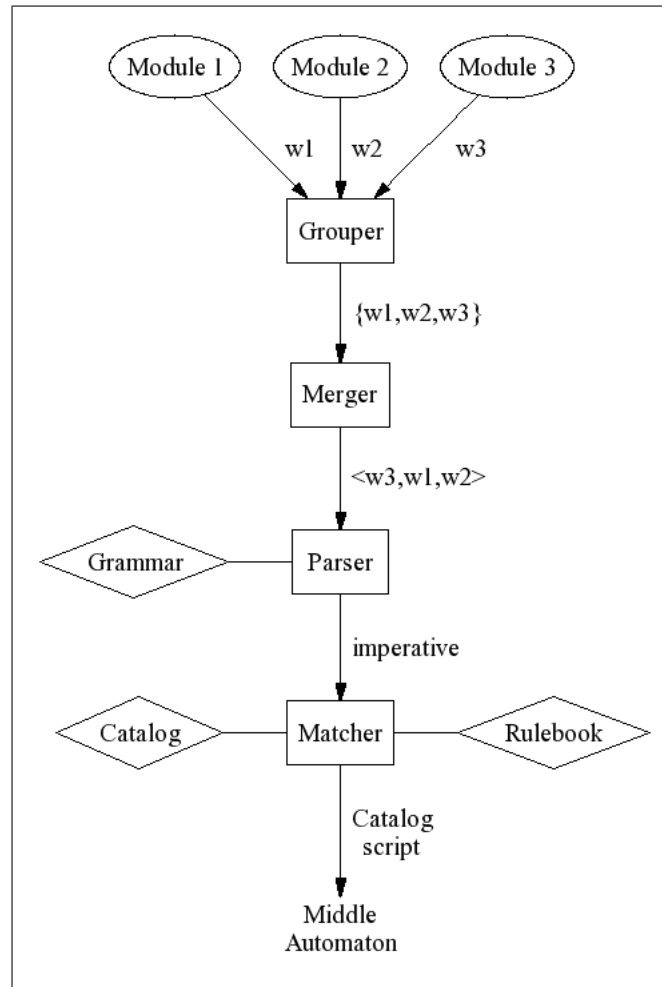


Figure 5.2.: Multimodal input processing in the Left Automaton.

e.g., there could be “drive” from the speech recognition module and “left” from the camera module.

2. The Barrier decides on which words belong together (i.e., which words are likely to express one single ‘user will’); we call this process *grouping*. In the above example, {“left”, “drive”} will be grouped. This step is necessary because there are words before and after the two words “drive” and “left” in the example, too; they do not, however, belong to the same user wish as “drive” and “left”.
3. A decision is made about the order in which these words have to be put to express the user will; this process will be referred to as *merging*. In the example, ⟨“drive”, “left”⟩ and ⟨“left”, “drive”⟩ are possible.
4. The merged string will be handed over to a natural-language *parser*. As a result, the purely symbolic input string will be transformed into a syntactically and semantically meaningful logical form. The example would then contain the information that “drive” is an imperative verb and that “left” is an adverb indicating a direction.
5. Finally, the parsed sentence will be *matched* against a Catalog containing all the commands that can be executed by the set of applications that have registered with the Barrier. To illustrate this at the above example, “drive left” could be matched by the Catalog entry “move left”.

We shall now explain the single points in greater detail:

Input from modules

Each of the active input modules ‘listens’ to the environment, and whenever it records something interesting, it converts it into a word (of class **String**) which is then passed to the Barrier.

To determine what exactly is considered to be interesting, and hence forwarded to the Barrier, is completely up to the input modules themselves; this can be a very easy but also quite demanding a task. For further illustration, two examples of that:

A keyboard module will have detected ‘something interesting’ whenever the ‘enter’ key is hit; it will then simply pass all the text that has been entered before the final ‘enter’ stroke to the Barrier.

The case is different with a camera module, which has got a subtler job to do: It has to filter the interesting movements out of a continuous image input stream and transform them into words with semantic content before handing them over to the Barrier. Such more sophisticated input modules will have to be endowed with the capacity to, at least

5. Multimodal input processing

in a restricted way, understand certain events in the environment. Later we will show how we built a simple camera input module that can distinguish two basic gestures: It can spot the user's head moving to the left and to the right, and it will transform these into the words "left" and "right", respectively.

To be sure regarding nomenclature, when we speak of a 'word' we need not necessarily mean 'sequence of letters without blank symbols'. An input module may pass several such entities to the Barrier at once, and we will still call this a 'word'. For example, a speech recognition module may pass whole sentences like "get me some beer" into the Barrier, and we will refer to it as a word.

Grouping

Depending on how many things are happening in the environment, there will be a more or less continuous stream of words produced by input modules and entering the Barrier, and it has to be decided which of them belong together (part of the input fusion).

How this is achieved will be discussed in detail in the section about the Grouper (chapter 5.1). At this point, we just anticipate that temporally close words will be grouped.

Merging

After words have been grouped they just form an unordered set. Our goal is, however, to regard them as constituting a sentence, and sentences always consist of an *ordered* sequence of words. Thus, the goal of the algorithm we call Merger is to put the elements of the set produced by the Grouper into sequential order.

As a guideline, the Merger will try to 'minimize entropy', i.e. try those combinations first that do not deviate much from the temporal order given by the words' time of arrival in the Barrier.

Again, this will be discussed in a separate section. Here it shall suffice to note that with merging, another method of processing is employed: that of trial and error. Whilst we consider only one single grouping of words, we are, from the Merger onwards, trying several possibilities of building sentences of the grouped words, until we succeed or until we have failed in all trials. The structure of the loop we use to accomplish such will be laid out in the technical section of this chapter.

Parsing

At that point, the input command is still only a string of letters, lacking any syntactic and semantic content. The only thing we know is that letters between a pair of spaces belong together and form single-word units. While grouping those words, there still was some guess work and an assumption (about temporal bindings indicating ‘relatedness’) involved. There could be no verification of this assumption for the specific input, since all that was done was just a series of symbolic transformations.

In the parsing step, however, we for the first time test whether our string of letters actually makes sense—or rather, makes sense *to the Barrier* according to the grammar we supplied. In case the sentence is found not to be nonsensical in the first place, we also learn the actual meaning of those character strings. The natural-language framework we use, OpenCCG, yields two kinds of information for every word:

- **Syntactic** information: Which part of speech is taken by which word(s) of the sentence? For example, in “eat a banana”, “eat”, an imperative verbal phrase, is the predicate, while “a banana”, a noun phrase, is the belonging accusative object; it is in turn formed by the indefinite article “a” in combination with the noun “banana”.
- **Semantic** information: Which is the meaning of each word (or sequence of words)? This goes well beyond the syntactic structure of a sentence. In the example, OpenCCG would provide us with these semantics: “eat” is an *action*; the second person (either singular or plural) is the *actor* (for he is meant to eat); “a banana” is the *patient* of the action, for it is the banana that is affected by it (and rather strongly so).

We shall explain why we used OpenCCG, how it works, and what kind grammar we used with it in on page 54, in chapter (6.1.3).

For now, it is worthwhile noting that parsing a sentence may produce more than one parse; the structure and meaning of a sentence may not be unequivocal. For example, “killing bears” might denote bears that kill, or the act of killing bears. In one case, the bears are perpetrators, in the other they are victims.

We call the elements of such a result set of the parsing process *imperatives*.

Matching

The final step in finding out whether a sentence is meaningful to the Barrier consists in comparing it with the *Catalog* of possible commands, the question being: “Is there an

5. Multimodal input processing

application that has registered with the Barrier and that has indicated that it can cope with the input sentence?”

Note that in this process, *no* information about the application itself is used at all, that information is not even available in the Left Automaton, where all of this happens.

We call this process *matching*. The Matcher takes a parsed sentence (a so-called imperative) as input and sifts through the Catalog for sentences that ‘match’ it. An imperative may be matched by one single Catalog entry, but generally it might be matched by a sequence of Catalog entries. We call such a sequence of imperatives a *script*.

Here is what exactly we mean when we speak of an imperative I , parsed from input uttered by the user, being *matched* by a script S of Catalog entries:

- I is trivially matched by itself, i.e. if S is a script consisting of one single imperative that equals I .
- I is matched by S if I can be transformed into S by applying a sequence of transformation rules that preserve the meaning of I . The set of these rules is stored in the Rulebook. Intuitively, if I is matched by script S , this means that the command I can be decomposed into the subcommands forming S ; e.g. one could imagine “buy a pretzel” matching ⟨“take a pretzel”, “put 45 cents onto the counter”⟩.
- I is matched by S if it is matched (according to the two above rules) by a script S' , with S being a *generalization* of S' , i.e. S contains variables that can be replaced by corresponding parts of S' . Here is an example in intuitive notation: “buy a pretzel” is matched by the Catalog script ⟨“take a T ”, “put 45 cents onto the counter”⟩, where T is a variable placeholder for all kinds of things.

We will use ‘*matches*’ as a symmetric relation: Whenever I matches S , we will also say that S matches I ; so no attention need be paid to the order in which the two appear.

Note that a sentence being parseable does not automatically imply that it is matchable. There will be lots of commands that can be ‘understood’ by the Barrier but which cannot be executed by any of the applications that have registered. “Buy the professor” is a conspicuous example.⁴

A great deal of time and work has been spent on a sophisticated Matcher algorithm and, as mentioned, the details will be illuminated in a chapter of its own.

⁴Or is it?

5.0.12. Technical description

In this section we will describe certain interesting aspects of our implementation of what has been said about multimodal input processing so far. We will do so in the same temporal order in which words ‘percolate’ through the Left Automaton.

Words from possibly multiple input modules enter the Left Automaton through its `processFromLeft` method—the canonical point of entry for all of our three Automata. Words are, however, not processed directly by this method, they are merely buffered in the Left Automaton’s Grouper. `processFromLeft` is called in a separate thread for each input token, but as there is only one Grouper object in the Left Automaton, one might think of all these threads as being joined by it.

When the Grouper has decided to group a set of words together, it will flush this group back into the Left Automaton for further processing. For this, it calls the Left Automaton’s `acceptGroupedWords` method.

As can be seen from the code in figure 5.3, we loop over all the permutations that are returned by the merger (`mergeLoop`) and try to parse each of them. If `$words = {"left", "go"}`, then we would try to parse “left go” and “go left”. As “left go” is not parseable we would immediately continue (line 9) with the next permutation “go left”, which would indeed be parseable. The result of parsing the sentence would be a set of parses (also called *parse trees*). Again, we loop over all parses (`parseLoop`), trying to match them against the Catalog. When we fail for one parse, we try the next one (line 15); when we succeed, we can quit the whole mechanism (line 17), since we have then found what we have been looking for: a Catalog entry that matches the user input. If no parse can be found, the algorithm will try to parse the next permutation, and so forth.

The result of matching is a script, i.e. a list of imperatives (might also only consist of one entry), each of which could be found in the Catalog; in this example one could conceive of a matching script (“turn left”, “start walking”), or simply (“go left”).

In case we succeeded with finding a parse, i.e. a matching script, we will process each of the imperatives forming the matching script in turn (line 20); in most cases this will be done by passing the imperative on to the Middle Automaton. Should an error occur while processing an imperative inside the Barrier, the user will be informed of this and subsequently the thread will return from the procedure (line 25) without trying to execute any of the remaining imperatives: Whenever one part of a more complicated command (i.e. of a script) fails for whatever reason, we consider the whole command to have failed. In the example, it would not make sense to start walking if we have encountered an error while turning left, since that would mean the initial command “go left” would not be fulfilled in a satisfying manner. We must assume this worst-case scenario for any command.

5. Multimodal input processing

Algorithm: `acceptGroupedWords($words)`

Input: `$words`: a set of words that have been determined to belong to one single user command by the Grouper

Output: none

```
1 $success = false ;
2 $permutations = get list of permutations of $words from the Merger;
3 mergeLoop:
4   for (each $permutation in $permutations)
5     $sentence = concatenate all the words in $permutation;
6     try
7       $parses = parse $sentence;
8     catch (ParseException)
9       continue mergeLoop;
10  parseLoop:
11    for (each $parse in $parses)
12      try
13        $matchingScript = match $parse against the Catalog;
14      catch (NoMatchException)
15        continue parseLoop;
16      $success = true;
17      break mergeLoop;
18
19 if ($success)
20   for (each $imperative in $matchingScript)
21     try
22       process $imperative;
23     catch (Exception)
24       report the error to the user;
25   return;
```

Figure 5.3.: The `acceptGroupedWords` algorithm. This is a simple form, not distinguishing between the command and data modes.

5.1. Grouper

5.1.1. Overview

The Grouper’s behavior is specified as follows: From the continuous stream of words arriving in the Barrier from input modules, it must form groups, i.e. sets, of words it deems likely to belong to the same ‘user will’. Fancy the situation that the user says the word “drive” and moves her head to the left; so “drive” will be catered to the Barrier from the speech recognition module, while the word “left” is produced by the camera module. The Grouper behaves correctly if it joins “drive” and “left” into the unordered set {“left”, “drive”}. Note that the correct *order* will be decided at a later point, the Grouper is merely concerned with finding correct *associations*.⁵

However, what if she then said another word a few seconds later? Would it also have to belong in that set?

The question is, then, which is the best way of making a decision as to what belongs together? How do humans do it?

A strong heuristic humans use is to group words that arrive at their ‘sensors’ with only small intervals (of time) in between: This is probably one reason why we make longer pauses between sentences than between single words, for sentences usually form one meaningful entity.⁶

We are also more likely to group input that reaches us via different senses: It seems easier to distinguish an auditory stimulus from a simultaneous or rapidly following visual stimulus than from another auditory one. Note, however, that such a separation of senses is not always called for and is indeed often not executed by humans—on the contrary, it is *merging* different senses rather than separating them that makes much of our perception as reliable as it is (cf. page 30).

Finally, humans can even distinguish two inputs that are perceived by one single sense simultaneously—if it originates from different sources. For example, we can listen to a conversation on TV and nonetheless understand our name or the phrase “The meal is hot!” being called by our mother; not only do we understand this separate utterance, we also recognize that it is indeed a *separate* utterance. Most psychological theories attribute this ability to the presence of different thresholds connected to different words in our

⁵Although the set is unordered, we do not discard temporal information about the order in which the tokens arrived. We will use it in the Merger.

⁶This, of course, is only true when we neglect inter-sentence dependencies like anaphora that are necessary to understand the meaning. But surely, a sentence forms more of a self-standing unit than a single word.

5. Multimodal input processing

minds: Our name has such a high priority, i.e. such a low threshold, that it can easily interrupt another input stream.⁷

This list is progressing from simple to more complicated, and when implementing a grouping mechanism for the Barrier, we would rather stick to the first alternative than try to emulate mechanisms that are not completely understood yet.⁸ In the next paragraph we will lay out how our grouping procedure works.

5.1.2. Description

Whenever a word arrives from some input module, it is stored in a buffer that is common to the whole Left Automaton, i.e. input from all the different modules is cached in the same buffer. If there has not been any new word for a fixed amount of t milliseconds, the current content of the buffer is considered to be belonging together, and the buffer is flushed. Whenever a new word arrives before t milliseconds have elapsed, the clock is reset and another t milliseconds have to pass before the buffer is flushed back to the Left Automaton to be further processed.

Briefly, all words that arrive in the Barrier with at most t milliseconds in between each pair of subsequent words are considered to form a semantically coherent group. In the Barrier prototype, the value of t is set to equal one second.

5.1.3. Development history, alternatives, and venues of improvement

In future versions of the Barrier, one could consider extending the mechanism, which is currently only taking temporal features into account, towards the more complicated methods employed by humans sketched above.

In an even more advanced version one could endow these mechanisms with more flexibility by making them adaptive. One could, e.g., try learning a variety of aspects from experience:

- One could learn for each input *modality* which other modalities they are likely to form word groups with.

⁷Palmer describes the scenario thus: “If you are at a party talking with one person, for example, and someone nearby says your name, you are very likely to notice it and shift your attention to find out why your name was mentioned.” [Pal99, page 534] There, he summarizes Treisman’s ‘attenuator theory’, which comprises the threshold idea mentioned, as the most likely explanation.

⁸Palmer, e.g., says Treisman’s ‘attenuator theory’ is the “the most widely accepted theory”, yet it is “in contrast to Broadbent’s filter theory” and other possible explanations. [Pal99, page 534] The dispute on which is the best model has not been settled yet.

- One might also try learning which particular *words* are likely to end up in the same group.

Such an approach would be viable as a supervised learning process: The feedback would be ‘true’ or ‘false’, depending on whether the combination of words that have been joined by the Grouper in a specific training example can be parsed or even matched (cf. lines 7 and 13 in the algorithm of figure 5.3). Since a successful match is a stronger condition than a successful parse,⁹ one could introduce three instead of just two feedback values, meaning ‘unparseable’, ‘parseable yet unmatchable’, and ‘matchable’, respectively.

A similar supervised learning process should also be conducted when a new user uses the system. The value to be learned would be t as defined in 5.1.2. The reason for that is that the ‘speed’ with which commands are input, and make groups, is probably dependent on each individual user’s characteristics. Hence, such a background learning of the value for t would serve to even better customize the Barrier to its user. It would also make sense as the available hardware resources certainly differ from system to system, e.g. the input modules might run on different hardware, further increasing (decreasing) the ‘lag’ in between words on a slower (faster) machine.

5.2. Merger

5.2.1. Overview

We have seen in the previous section that the Grouper’s job is to find out which user utterances are likely to belong together to form one single ‘user wish’. The Merger’s job is to further process the Grouper’s output by bringing the words that belong together into order.

A major difference between the ways the Grouper and the Merger work is that in the output of the former, each word appears in only one single grouping, while the Merger will return all possible orderings of a grouping, meaning that each word may appear in several orderings. They will later all be tried when we try to find matching Catalog entries (cf. the algorithm of figure 5.3).

The reason why we consider on several orderings is the multimodal nature of our input processing: when a user spreads her wish over several modes (i.e. ‘communication channels’), it need not necessarily be the case that she utters the ‘snippets’ in the order she would use when vocally pronouncing the demand. For example, she moves her head to the

⁹For an easy ‘proof’ of that, only successfully parsed sequences are attempted to be matched, and that match in itself is not always successful, implying that matching is indeed the stronger condition.

5. Multimodal input processing

left and says “drive” some moments later. Then the original ordering (“left drive”) would not be the one she would use in a spoken sentence (“drive left”).

Also, she might have done these actions *simultaneously*, and they would still arrive in the Left Automaton in some sequence (in the discrete environment of a computer, there is always an order), which depends simply on which input module takes longer in processing the user’s action, and to forward it to the Barrier.

So, to be able to fulfill the user’s wish nonetheless, we must try several orders.

5.2.2. Description

As is known from basic combinatorics, n words can be arranged in $n!$ different sequences. If the Merger gets a set of n words that have been grouped it will produce all $n!$ permutations. What matters is the order in which we put these permutations, for later, when matching against the Catalog, we want to try the most likely permutations first. For example, although it is possible that the user puts her utterances *not* into the order she would use in an all-spoken sentence (see above), we assume that she is *most likely* to use the same order in both cases, and for that order to possibly be preserved on their way through the input modules.

This is why the principle we followed in ordering the permutations is to proceed *in ascending entropy*, meaning that the permutations that resemble the order of utterance most come first. More concretely:

Permutation π comes before permutation ρ if and only if $\text{trans}(\pi) \leq \text{trans}(\rho)$, where $\text{trans}(\cdot)$ stands for the number of *transpositions* needed to produce a permutation from the original ordering. A transposition is the exchange of two *neighboring* elements.¹⁰ For example, 1324 has one transposition.

Instead of using transpositions (exchanges of two neighboring elements) in our definition of entropy we could also have used the number of exchanges we need when we may swap arbitrary elements. But this is much less precise, e.g. it would assign the same amount of entropy to 4231 as to 2134, although most probably a user is more likely to just swap two neighboring elements (as in the latter case) than the first and the last one (the former case).

Figure 5.4 displays the pseudocode of the algorithm we are using for computing all permutations of an initial sequence. A call to the Merger with a list `$words` of grouped words will simply return `permute(λ , $words)`, where λ is the empty list, serving as the prefix

¹⁰In the rest of this section we shall use the numbers from 1 to 4 as an example, 1234 being the original, i.e. transposition-less, order.

Algorithm: `permute($prefix, $words)`

Input: `$prefix`: an auxiliary list of words, use the empty list in the initial call

`$words`: a list of words, their order reflecting the temporal order of utterance

Output: the list of permutations of `$words`, ordered in ascending entropy

```

1     $N = number of elements of list $words;
2     if ($N == 0)
3         return a list with $prefix as its single element;
4     else
5         $result = empty list;
6         for ($i from 0 to $N)
7             $p = $prefix with $i-th element of $words appended;
8             $w = $words with its $i-th element removed;
9             result.add(permute($p, $w));
10    return $result ;

```

Figure 5.4.: The algorithm for computing permutations.

for the first call of `permute`. In the resulting list, permutations with lower entropy (as defined above) will come first, so our algorithm exactly fulfills our ‘entropy order’ requirement. Figure 5.5 illustrates an example run for the original sequence 1234. Permutations (the leaves of the tree) are produced by the algorithm in depth-first order, starting with the edge labeled ‘1’ and stopping with the edge labeled ‘4’. One can see that 1234 (no transpositions) comes first, and the inverse, 4321 (six transpositions) comes last. Edges are labeled with the prefixes all of their subsequent nodes share.

5.2.3. Development history, alternatives, and venues of improvement

Similarly to what has been stated in the section about the Grouper, one could try *learning* again: in the case of the Merger, this would involve learning the entropy function from examples rather than hard-coding it the way we did. For example, if we used neural nets, the number of transpositions (which is our complete entropy measure) would be only one of several input features.

Neural nets would provide a method for generalization. Also one could learn explicitly from the past by making those orderings more likely that have been successfully parsed or even matched previously.

In general, we are currently using a merging mechanism of type a in the list of page 30:

If a command is spread over several modalities, the system can reasonably well cope with that. There is, however, still a large leeway for extending the functionality towards type b of that list: dealing with redundancies (e.g. a word being said and merely *emphasized* by a

5. Multimodal input processing

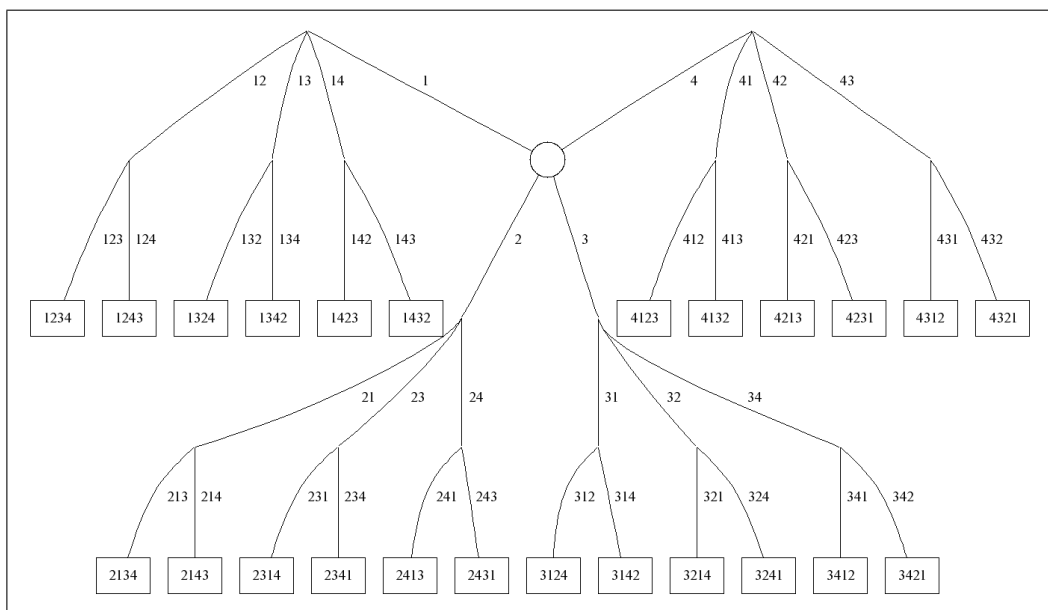


Figure 5.5.: An example run of the Merger for input $\{1, 2, 3, 4\}$, 1234 being the original order. Nodes are produced in depth-first order. Edges are labeled with the prefixes all of their subsequent nodes share.

gesture) would be sure to significantly boost the accuracy of intention detection. But this is not a simple add-on, it would rather involve the implementation of a large framework in an area of science that is currently scrutinized by a big many researchers in a big many fields ranging from computer science through linguistics to psychology (cf. 5.0.10). How such is being achieved for a limited domain can be seen in the scope of the JAST project [KG01] that has been mentioned in section 5.0.10.

6. Natural-language processing

Dave Bowman: *Open the pod bay doors, HAL.*

HAL: *I'm sorry, Dave. I'm afraid I can't do that.*

—Stanley Kubrick and Arthur C. Clarke, screenplay of *2001: A Space Odyssey*

6.0.4. Overview

Natural-language processing is an integral part of the Barrier. It provides a high degree of flexibility, since working with natural instead of manually ‘compiled’ sentences abstracts from the implementation level, which is exactly what the Barrier is supposed to do.—We do not want the user to have to remember a host of syntactically very restricted shell commands to communicate her wishes to the computing system, we want her to be able to simply ‘speak her mind’.¹

An important idea that is realized in the system presented is to apply natural-language techniques not only to sentences that are spoken or typed as complete sentences by a user, but also to sentences that are produced by the Barrier from user input that reached the system through a *multitude* of input modalities.

Grouping and merging, the two processes that have been described in the previous chapters, produce ordered sequences of words. After this we will operate in the realm of natural-language processing: The sentences we work with from this point onwards are regarded as though they were spoken or typed by the user *as such*, even if they were *in fact* recorded by different input modalities. So, when “compile” and “program” were perceived by two different modules and we are currently considering the permutation “program compile” produced by the Merger, we will consider it the same way as if the user had typed “program compile” into the keyboard input module, although she would not actually do this—unless her English is bad.

¹We believe that “list the contents of my home folder” is more intuitive than `cd ~; ls -al`.

6. Natural-language processing

We used OpenCCG,² a state-of-the-art tool for natural-language processing written in the Java language. It offers a parser to convert natural sentences to syntactically as well as semantically labeled data structures and a realizer for the opposite direction.

Details of the OpenCCG tool and the underlying theory will be discussed in a later section. In the subsequent paragraphs we will first look at one of the major problems in natural-language processing and then describe in a rather high-level manner which data structures we maintain for dealing with natural language and how we make use of them.

6.0.5. Ambiguity

One of the larger hurdles to be overcome is the ambiguity inherent in language. A particularly impressive example is the simple sentence given in [JMK⁺99, page 4]:

I made her duck.

That supposedly simple sentence alone allows for at least *five* interpretations:

1. I cooked waterfowl for her.
2. I cooked waterfowl belonging to her.
3. I created a duck she owns.
4. I caused her to quickly lower her head or body.
5. I waved my magic wand and turned her into undifferentiated waterfowl.

Such ambiguities can be categorized as follows:³

- Sentence boundary ambiguity, i.e. it is unclear where a sentence starts or ends (as the sign “.” is not accurate enough, given e.g. an Internet address within a sentence, containing a dot).
- Lexical ambiguity, i.e. one word has several meanings (“duck” in the example); resolving implies selecting the meaning of a word that is the most meaningful in the given context.
- Syntactic ambiguity, i.e. one sentence has multiple parse trees, with different meanings; resolving usually requires semantic and contextual information.

²CCG stands for ‘Combinatory Categorical Grammar’.

³cf. <http://www.proxem.com/WhatIsNLP/tabid/59/Default.aspx>

- Semantic ambiguity, e.g. “her” in the example; it might not be clear who is meant. The solution is resolving such *anaphoras* (cf. figure 6.2).

Different ways of curbing these ambiguities are, among others:

1. Reduce the set of words in the grammar to one which allows for fewer ambiguities,
2. restrict the domain to a simpler one, e.g. instead of all-purpose sentences only allow those about calendars, or
3. only allow certain kinds of sentences, e.g. only questions, or only commands.

The Barrier uses the first venue to some degree (restricted vocabulary), and the third one fully: as we will see, we only admit imperatives, i.e. commands.

A thorough introduction to computational linguistics is given e.g. in [Hau01] and [All01].

6.0.6. Description

Each of the following will be described in greater detail in its own appropriate section, whereas this is only meant to be a short presentation of the language-related inventory we deploy.

Grammar

The grammar encapsulates the language model used. The parser will be able to recognize those and only those sentences that are derivable from the grammar. Conversely, the realizer will make use of the grammar to build a natural sentence from a previously parsed one.

Parser

The OpenCCG parser must be provided with a grammar in a specific format and with a concrete sentence; it is then able to parse the sentence, i.e. to recognize whether it is in accordance with the grammar, and if so, which syntactic and semantic function each word has.

Our parser wraps an OpenCCG parser, adding some further constraints on what is a legal utterance.

Imperative

OpenCCG returns its parse results as objects of a specialized class. Obviously we do not want to manipulate this plain data structure but rather adapt it to our particular needs. This is why we wrap what OpenCCG gives us in objects of our own.

We call these concrete objects *imperatives*. Also, we apply the term in a more abstract way when referring to any legal user utterance as an imperative. We may do so because, as we shall see later, our grammar admits only sentences that form demands, i.e. imperatives.

Script

As we have already seen when we first talked about the Matcher, it will be of use to have a dedicated word with the meaning ‘ordered sequence of imperatives’. We chose the term ‘*script*’.

As with imperatives, we will use the word both for the concrete data structure and for the high-level notion of a command list.

Catalog

When some input from the user has been successfully parsed, we must check whether there is any application program that can cater to the user’s wish. To inform the Barrier about what they can do, applications register for imperatives representing those actions. The Catalog is the data structure in which all these imperatives are stored.

Goal Test

The easiest way to check whether some user input matches an imperative offered by an application would be to trivially check whether the two are equal. This, however, would be an enormous restriction: Applications would have to register for each separate action they can perform, which would be cumbersome or even unfeasible in many cases. This is why we introduced the concept of variables within imperatives, and an imperative I_1 does not match another imperative I_2 in case of equality only but also when I_2 is a generalization of I_1 , i.e. when I_2 contains variables and I_1 is of a form so that it fills all the variable slots of I_2 correctly.

Testing this is a task that will occur in several places, and we call the mechanism Goal Test.⁴

⁴The reason for this nomenclature will become evident when we discuss the Goal Test’s role in the Matcher

Rulebook

Even with the possibility of using variables within an imperative that is to be registered in the Catalog, testing whether a user’s utterance conforms to a given imperative would still be a one-step process: it would consist of one single call to Goal Test.

We, however, strive for more flexibility. So the Barrier also provides a mechanism for defining rules that can be applied to imperatives *without changing their semantics*. One type of such a rule could be a simple translation between synonyms: e.g. “delete” has the selfsame meaning as “discard”. Alongside these will be other types of rules as well.

The ensemble of all such rules will be called Rulebook.

Matcher

By introducing rules for modifying imperatives, matching is no more a one-step process but rather changes its character towards a complex search problem. It will be solved by the Matcher algorithm, which we have already mentioned in section 5.0.11.

Realizer

Finally, we are also employing the realizer that is provided by OpenCCG, thus exhausting the whole spectrum of services the tool has to offer. We will, however, use the realizer in a far less extensive manner than the parser.

6.0.7. Development history, alternatives, and venues of improvement

Naive alternatives to a full-scale grammar

In the beginning we envisioned to define a format of our own for specifying imperatives. Such was one of our first sketches that what is now simply passed to the parser as the natural-language sentence “eat a banana” would have been:

Action: eat
Patient: banana
Beneficiary: —
Modifier: —

algorithm.

6. Natural-language processing

Clearly this is a naive and very inelegant way of dealing with the problem. Here are some reasons why:

- Input ‘snippets’ arriving from different input modules would have to be stuck into the correct slot; e.g. it must be decided whether “eat” is the action or the patient etc. of the resulting structure. It is not clear how this could be accomplished.
- Application programmers who want to register their application for an imperative would have to specify the imperative in this format, which is unrealistic because it assumes programmers spend time learning the intricacies of the format. In our current solution they simply use natural language.
- These two points could be simplified if we had written a parser to build such structures from natural-language input. Which is even more unrealistic.
- The structure is very rigid; once specified, it would be difficult to change it because many parts of the code would depend on it. In OpenCCG, on the other hand, one merely has to change the grammar, while all the surrounding code stays the same.
- It is unclear how such a structure could allow for more complex entries such as “ripe banana”, which is actually hierarchical, consisting of a noun with an adjective depending on it.
- There would have to be *one* pattern (such as the one shown above) containing *all* the slots that could possibly be filled, even if in fact there is nothing to fill them (indicated by the “—” in the example). A tool like OpenCCG is much more flexible in this respect, since it allows to specify complex grammars, and anything conforming to the grammar is considered legal.⁵

We stop continuing this list, as these items should have made obvious that this would not have been a smart way of tackling the input language problem.

It was only thanks to a valuable hint by Alois Knoll that we learnt about OpenCCG and chose it as the way to go. It has since proven to be an indispensable advantage to exploit the services of a natural-language tool for our tasks, which clearly exceed what is normally done with such software. We were also able to write grammars supporting variables in it, which to us was a significant advancement of its capabilities.

⁵In terms of theoretical computer science, a simple scheme as sketched would also define a grammar; it would just be so extremely simple that one could hardly call it a natural-language grammar.

Context sensitivity

If the user is to interact in an absolutely natural manner with the computing system, then the latter would have to cope with the problem that it is “a ubiquitous feature of natural languages that utterances are interpretable only when the interpreter takes account of the contexts in which they are made.” [KvGR, page 2] As an example, the user might utter the request

“Harvest a banana. Peel it.”

It will then be necessary to somehow link the pronoun “it” to the indefinite noun phrase “a banana”. The same will be true if the user does not say “it” but the definite “the banana”. In general, *indefinite* expressions such as “a banana” have no point of reference, whereas *definite* expressions such as “the banana” or “it” always refer to something that has previously been mentioned—or *that is implicitly evident from the context*, as though it had in fact been mentioned. This phenomenon of back-referral is called *anaphora*.

To cope with such a scenario in the Barrier, one would have to wedge something like a *contextualization* step between parsing and matching whose task it would be to deal with such complications.

One can easily imagine that this is, in the general case, not a trivial problem. But there do exist a variety of linguistic theories about how to deal with this and similar intricacies of human speech. “The central concern of these theories is to account for the context dependence of meaning.” [KvGR, page 2] One example is Hans Kamp’s *Discourse Representation Theory* (DRT; cf. [KvGR]). It provides a framework that allows for labeling the above sentence in such a way:

“Harvest a banana^{*i*}. Peel it_{*i*}.”

There, a superscript (*i*) labels a phrase, while the corresponding subscript indicates which phrase the subscripted phrase (the anaphora) refers back to.

These labels would have to be incorporated into the imperative object that is to be passed to the application chosen to do the job. The application itself would then have a clear representation of which phrases within the imperative refer to the same entities of the ‘universe’. This information might then be exploited to better meet the user’s demands. In our above example, the ‘banana peeling application’—if it is programmed to be intelligent enough—could tell from the labels that it is meant to peel the banana it most recently harvested, not just any one from its banana pail.

Out of all the existing theory frameworks, DRT would be among the foremost to consider

6. Natural-language processing

because there already are solutions for coupling it with the major natural-language framework used in the Barrier, CCG: “At present there are versions of DRT building on many of the leading syntactic frameworks—in particular [...] forms of Categorical Grammar [...]” [KvGR, page 3]

Note that such a tool, however useful it might be, cannot be a philosopher’s stone either: even humans have sometimes problems resolving ambiguities such as anaphoras (cf. also section 6.0.5). For a computer to know that

“Get my kid a banana^{*i*}. Peel it_{*i*}.”

is a correct labeling, whereas

“Get my kid^{*i*} a banana. Peel it_{*i*}.”

has horrible consequences, it would have to have a command of an immense body of commonsense knowledge, which is, for the time being, still a utopy.⁶

6.1. Grammar

Semantics is the theory concerning the fundamental relations between words and things.

—G. Bealer in [Bea82, page 157]

6.1.1. Overview

Theoretically, one may think of the OpenCCG parser as a Turing machine which we feed both a grammar and a natural-language input sentence that is to be derived according to that grammar. The realizer computes the inverse function, i.e. it takes a grammar and a parse tree as input and yields a natural-language sentence as its result. No matter what the direction, the point to make is that OpenCCG can only provide the framework for natural-language processing (NLP), the grammatical description of that natural language still has to be supplied by him who uses that framework.

⁶Compare e.g. Minsky in his 2006 book ‘The Emotion Machine’: “No present-day programs have Commonsense Knowledge. Each present-day program is equipped with no more knowledge than it needs for solving some particular problem.” [Min06, page 163]

The Barrier’s operating language is English, so we developed an English grammar following two criteria:

1. The grammar should be simple enough to clearly demonstrate the working principles of our proof-of-concept implementation without entangling readers and developers in a jungle of excessive details.
2. The grammar should be sophisticated enough to cover a large spectrum of possible use-case scenarios. If the grammar were too simple it would simply be useless.

One big requirement was for the grammar to allow for a *flexible* mechanism for matching *concrete* user input against services offered by applications in a possibly more *abstract* way. This is why we built in the possibility to make use of *variables*.

In this section we will first take a brief look at some frameworks for dealing with natural-language processing, before going into more detail about OpenCCG, our tool of choice. Finally we will lay out all the details of the grammar we conceived.

6.1.2. Research survey

Here is a list of some alternatives to OpenCCG:

“Grok is a library of natural language processing components, including support for parsing with categorial grammars and various preprocessing tasks such as part-of-speech tagging, sentence detection, and tokenization”.⁷ It also makes use of the OpenNLP Maximum Entropy Package, which is “a powerful method for constructing statistical models of classification tasks, such as part-of-speech tagging in Natural Language Processing”.⁸ Grok is the precursor of OpenCCG.

WordNet is a semantic lexical database, which was developed by Princeton University’s Cognitive Science lab under the direction of George Miller.⁹ It is in usage as a database for NLP solutions, and considered one of the biggest repositories on information about the English language. [Fel98]

“WordFreak is a Java-based linguistic annotation tool designed to support human, and automatic annotation of linguistic data as well as employ active-learning for human correction of automatically annotated data”.¹⁰

Self-styled the “Eclipse of Natural Language Engineering”,¹¹ GATE is a framework for

⁷<https://sourceforge.net/projects/grok/>

⁸<https://sourceforge.net/projects/maxent/>

⁹<http://wordnet.princeton.edu>

¹⁰<https://sourceforge.net/projects/wordfreak/>

¹¹<http://gate.ac.uk>

6. Natural-language processing

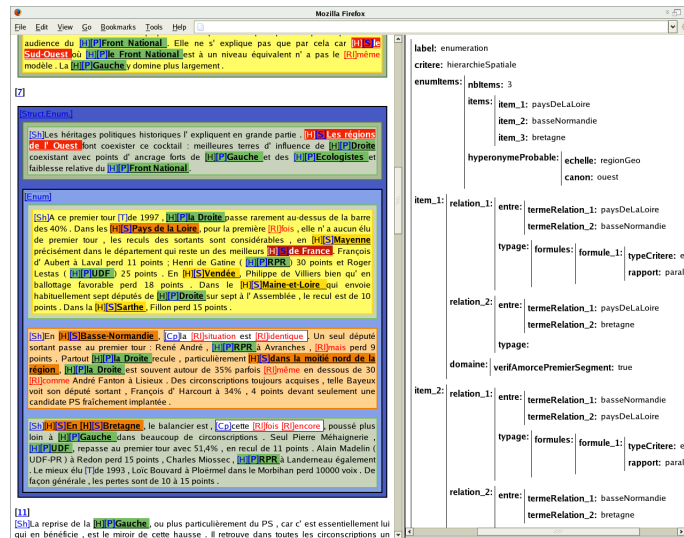


Figure 6.1.: Screenshot of LinguaStream.

Source: <http://www.linguastream.org/images/screens/output.png>

robust NLP tools, and in use at a number of institutions, chiefly the University of Sheffield, where it is developed by H. Cunningham [CMBT02]. While in itself it offers no truly new approaches, it has a large number of other solutions and applications included within its framework.

In the same niche is LinguaStream.¹² Developed by the University of Caen, France, it offers “a variety of standard operators, like word or sentence splitting, part-of-speech -tagging, projection of lexical resources and statistical analyses”.

Antelope¹³ is yet another framework for developing NLP applications. It comes with implementations of a part-of-speech tagger and an anaphora resolver, as depicted in picture 6.2. It leans on the previously described WordNet as a semantic lexicon.

OpenCCG’s properties are described in the next subsection. While not necessarily showing better performance than the previously mentioned NLP software, it uses state-of-the-art semantic structures and is slim enough to package with the Barrier without dominating it.

6.1.3. CCG and OpenCCG

A good introduction to Combinatory Categorical Grammars (CCG) is [Ste96] by Mark Steedman, who was the first to introduce the concept. For a more detailed account, cf.

¹²<http://www.linguastream.org/home.html>

¹³An acronym for **A**dvanced **N**atural Language **O**bject-oriented **P**rocessing **E**nvironment.



Figure 6.2.: Screenshot of Antelope depicting anaphora resolution.

Source:

<http://www.proxem.com/Antelope/AnaphoraResolution/tabid/65/Default.aspx>

6. Natural-language processing

[Ste87] by the same author. An excellent hub with links to many further documents is the University of Edinburgh CCG site.¹⁴

Here we shall just briefly introduce the concept of CCG and its implementation, OpenCCG, so that the reader has all the tools at hand to understand how we make use of the package.

CCG

Combinatory Categorical Grammars, like any formal grammar, are a formalism for describing formal languages. In particular, CCG was developed for the formal description of natural languages. In terms of the Chomsky hierarchy, its generative power is *mildly context-sensitive* [Bal02, page 47], i.e. a CCG can describe more languages than a context-free grammar but fewer than a fully context-sensitive one. This is still a long way from Turing completeness, i.e. the generative power of *any* formal grammar, but the step from context-free grammars to CCGs seems to be a ‘linguistically significant’ [Bal02, page 46] one. Whether this gain be the crucial one allowing for the formalization of any natural language or not, the power of CCG is definitely sufficient for our purposes. Also, despite this expressiveness, CCGs are still ‘efficiently parseable’.¹⁵

The approach of CCG is very much alike to the *lambda calculus*: there, a function (*abstraction*) of one argument is written as

$$\lambda x. fx$$

and the *application* to an argument a yields

$$(\lambda x. fx)a = fa.$$

In OpenCCG every word of the lexicon is of a function type (called *category*) that is defined similarly: a word category is e.g.

$$C_2/C_1,$$

meaning: an argument of category C_1 is expected on the right-hand side and the result of its application will be of category C_2 ; schematically:

$$(C_2/C_1)C_1 \longrightarrow C_2.$$

To see the similarity to the lambda calculus, imagine a notation like this for the last

¹⁴<http://groups.inf.ed.ac.uk/ccg/>

¹⁵cf. <http://groups.inf.ed.ac.uk/ccg/index.html>

example: $(fC_1/C_1)C_1 \longrightarrow fC_1$, with $fC_1 = C_2$. As one can see, a difference to the simple lambda calculus exemplified above is that CCG is typed (categorized); another one is that in CCG an argument can also be expected from the left- instead of the right-hand side, indicated by a mirrored slash:

$$C_1(C_2 \setminus C_1) \longrightarrow C_2.$$

C_2/C_1 is called a ‘*complex category*’, while both C_1 and C_2 are ‘*atomic categories*’ within the complex one.

Here is an example: intransitive verbs (i.e. verbs without object) have category

$$s \setminus np,$$

meaning that they expect a noun phrase (np) to their left and result in a sentence (s), e.g.

$$np_{\text{“Agnetha”}}(s \setminus np)_{\text{“comes”}} \longrightarrow s_{\text{“Agnetha comes”}}.$$

If the subject were “I” instead of “Agnetha”, the verb form would have to be “come” instead of “comes”. CCG offers techniques to define such restrictions. In our example, one would have to specify that the person and number of the verb and the noun phrase must match (this would not be the case with “I comes”: “I” is first person singular, whilst “comes” is third person singular).

Since we will later see a whole list of all the categories used in the Barrier, the example we have given shall suffice here.

OpenCCG

OpenCCG¹⁶ is essentially an implementation of the abstract CCG formalism and forms part of the comprehensive open-source NLP software project OpenNLP.¹⁷

It does include some extensions, notably by Baldrige (cf. [Bal02]), but as we do not exploit any of them, we will not further dwell upon them.

The ‘technical description’ section of this chapter will feature several examples of the actual XML representation prescribed by OpenCCG, while in this section we will explain how the tool is used on a slightly more abstract level. More technical details can be looked up in the manual issued by OpenCCG’s creators [BKW06].

First of all, the set of words is specified in the vocabulary file `morph.xml`. Every word

¹⁶<http://openccg.sourceforge.net>

¹⁷<http://opennlp.sourceforge.net>

6. Natural-language processing

form belongs to a *stem* (e.g. both “come” and “comes” are derived from the stem “come”) and has a *part of speech*; each stem represents exactly one part of speech. For example, “professor” is a noun, and it is *nothing but* a noun. The part of speech may be looked at as a “pre-syntactic” property: it does not tell us anything about the word’s behavior in a sentence yet, it just tells us of what kind a word is.¹⁸

Through its part of speech, a word is related to the set of *lexical families* in `lexicon.xml`.¹⁹ A lexical family can be thought of as a group of one or several related categories. Each lexical family can contain words of exactly one part of speech. If the family is declared *open*, any word with this part of speech is a legal member (given it is of one of the family’s categories, of course), otherwise (if it is declared *closed*) all member stems must be explicitly listed.

Each category can contain *feature structures*, which are used to implement limitations such as the one mentioned above, stating that the number and person must match for intransitive verbs —so that “I comes” is not successfully parsed.

When specifying a family, one not only lists its categories but also defines the *logical form* that will result when one of the categories is applied to a concrete word. One may conceive of the logical form as the parse tree, enhanced by semantic information. An example for such additional information is the semantic *type* of an expression; e.g. the logical form of “Agnetha comes” not only states it is a sentence (syntactic information) but also that it is an action (semantic information; there might be non-action kinds of sentences such as “Yes.”). The type hierarchy is stored in the file `types.xml`.

All the `.xml` configuration files mentioned must be listed in `grammar.xml`, which thus contains all the information that defines a grammar.

6.1.4. Description

When designing the grammar, we must ask ourselves, “What is the most sensible form for specifying imperatives that are legal input from the environment and that serve as commands to applications?” The challenge is to find the straight between being too general (in the most extreme case admitting just any English sentence) and being too restrictive (e.g. admitting just intransitive verbs). Note that the OpenCCG developers themselves state that “at present, OpenCCG is best suited to limited domains, where

¹⁸As an example consider the words “us” and “we”: they share the same part of speech (pronoun first person plural), but they are syntactically different: “us” can only appear as an object, while “we” will always be subject.

¹⁹The nomenclature is slightly irritating: the lexicon is not the set of words, as in everyday speech, but rather the set of lexical families.

coverage is not as large an issue.”²⁰

Imperatives only

The most obvious restriction we impose is that user input should have the shape of an *imperative* sentence in order to be considered legal. The Barrier does not accept floating sentence tokens such as single nouns, nor sentences that are complete but not in demand mode, such as questions or propositions. The rationale behind this is that the Barrier’s purpose is to filter user *commands* (i.e. imperatives) and pass them to an apt application.

One could indeed conceive of questions serving as legal input, too, for questions—unless they are merely rhetorical—are something like implicit commands: they call for an answer. Restricting the scope to imperatives might at a first glance rule out the possibility of questions, but we can actually cope with the situation in an elegant and consistent way if we explicitly mark the question as what it has implicitly been all the time: an imperative. If we do not ask “Where is Godot?” but rather command “Answer my question: Where is Godot?”, the input meets the criterion of being an imperative.

The actual question part “Where is Godot?” would then be considered raw (i.e. unparsed) data input that will be appended to the parsed and matched imperative “Answer my question”, for which some question answering application might be registered. How we handle such raw data appendices is discussed in chapter 7.

This solution complies completely with the Barrier’s philosophy: The Barrier can only specify the structure of commands *directed at itself*, it can know nothing about the ways any other applications expect their natural-language input to be; this is clearly those applications’ prerogative and duty. For example, such a question answering application (or ‘oracle’) could be written for questions in any language. So it would be inconsistent to cater questions in a pre-parsed form to an English ‘oracle’ while leaving the job undone for a French one.²¹

Variables

Beyond this useful constraint to imperative sentences there are no major limitations. On the contrary, there is one major advantage built in: The concept of a variable within a natural-language sentence. This way one cannot only represent regular English sentences like “find me”, but also more abstract expressions like “find *P*”, *P* being a placeholder

²⁰Quoted from file `SAMPLE_GRAMMARS` in the OpenCCG installation directory.

²¹Apart from being inconsistent, it would be unfair and—at least in Canada—considered politically incorrect.

6. Natural-language processing

for *any* person noun phrase. A specific advantage our system offers is that such general sentences need not be ‘pre-compiled’ either; they can simply be formulated in natural language and be parsed the usual way.

“Find *P*” was just used as a convenient way for displaying what would actually have to be written “find \$x0 :person”. The technical reasons for this particular scheme will be discussed below, when we list our lexical families. In a higher-level view, it is merely important to know what the single parts mean.

- We call all of the expression “\$x0 :person” a *variable*.
- “\$x0” is called a variable *label*; it represents the name of the variable. In our specific implementation, variable labels range from “\$x0” to “\$x7”. Although it is hard to imagine a sentence with more than eight variables, one could of course easily extend the set of possible variable labels by simply adding further names to the grammar’s morphological registry.
- “:person” is the *semantic type* of the variable. A variable being of a specific type means that it can be filled only by a concrete sentence part of that type. In the example, we could not substitute *things*, say, a chair, for the variable but only semantic objects that are labeled as *persons* (or subtypes of type ‘person’) in the morphological lexicon of the grammar.

Note that, although we refer to “\$x0 :person” as a single variable as though it formed one entity, it is indeed, from the grammatical point of view, composed of two separate words. This ‘factorization’ enables us to combine any label with any type, and we have to define each of them only once. The alternative would have been to ‘hard-code’ eight variables for each type, e.g. “\$person0”, “\$person1”, ..., “\$direction0”, “\$direction1”, ..., etc. In spite of looking fancier, this would be a less elegant solution because it would require a multiplication of lexical entries.

Types

We have seen that the notion of a semantic type is essential when dealing with variables. Fortunately, OpenCCG comes with a mechanism for defining a customized *ontology* of types, so there is no need to implement this anew.

It turns out that it is actually only in the context of variables that a type hierarchy is necessary. But in this particular place they are an indispensable ingredient. The whole concept of a variable would not make a lot of sense if variables were not typed.—Just imagine an untyped scenario in which an application that controls an ‘eating robot’²²

²²At this point the user is supposed to simply make a leap of faith and believe such a robot exists.

```

conjunction (e.g. "and")
sem-obj (semantic object)
  phys-obj (physical object)
    animate-being
      person
      phys-thing (physical thing, e.g. "flower")
  virt-obj (virtual object)
    virt-thing (virtual thing)
      module
        output-module
        input-module
situation
  change
    action (e.g. "compose a message")
    direction (e.g. "left")
  state
    boolean (e.g. "yes")
    time (e.g. "minute")
    speed (e.g. "fast")
    property
      number (e.g. "five")

```

Figure 6.3.: The Barrier’s type hierarchy (its ontology).

registers for “eat \$x0” and a user, in the mood for quips, requests “eat me”. Then “me” would indeed match the typeless variable “\$x0”. We do not want to further paint this scenario, but rather remark that it cannot possibly occur in a typed environment.

Figure 6.3 displays in a hierarchical tree-like list the ontology we used. If a type is indented, it is a subtype of the next type that is above it and left to it in the schema. ‘Semantic object’ (sem-obj) is the root of our type hierarchy;²³ virtually all entities are semantic objects, just like every object in the Java language is of class `Object`. We will not further discuss the type tree, it should be rather self-explaining.

Lexical families

To document what our grammar is capable of doing, we listed all the lexical families the Barrier knows in table 6.1.

If the family is closed the ‘Examples’ column will contain all the member stems of the family, if it is open it will show just a few examples.

²³In our case, only ‘conjunction’ stands out, for they do not really have a semantic value but are rather of syntactic nature only.

6. Natural-language processing

Family	Part of speech	Open/closed	Category	Examples
Nouns	N	open	n	e.g. message, professor
Pronouns	PRON	open	np	e.g. he, she, it
Adjectives	ADJ	open	n/n	e.g. green, two, every
Adverbs	ADV	closed	pp, s\s	away, left, right, forward, backward, west, east, north, south, circle, clockwise, wave, fast, slowly, now, always
Definite determiners	DET	closed	np/n	def (stem of “a”, “an”, “some”)
Indefinite determiners	DET	closed	np/n	indef (stem of “the”)
Prepositions	P	open	pp/np _{acc} , s\s/np _{acc}	e.g. in, for, from
Casemark prepositions	P	closed	ppCasemark/np _{acc}	to, for
Intransitive verbs	V	closed	s, s/pp, s/pp/pp	go, stop, drive, dance, exit
Transitive verbs	V	closed	s/np _{acc}	buy, rent, write, compose, send, enter, download, fetch, make, reject, accept, redirect, clear, set, exit, take
Ditransitive verbs	V	closed	s/np _{acc} /np _{acc}	buy, rent, give, write, compose, send, download, fetch, make
Ditransitive verbs with “for”	V	closed	s/ppCasemark _{“for”} /np _{acc}	buy, rent, download, fetch, make, dance
Ditransitive verbs with “to”	V	closed	s/ppCasemark _{“to”} /np _{acc}	give, write, compose, send
Conjunctions	CON	closed	s/s\s	and
Particles	PART	closed	s	part-negative (stem of “no”, etc.), part-positive (stem of “yes”, “affirmative”, etc.)
Periodic interval	ADJ	closed	s\s _{action} /np _{time}	every
Variable labels	VAR	closed	var	\$x0, ..., \$x7
Variable types	V	closed	s\var	:action
Variable types	N	closed	n\var	:sem-obj, :phys-obj, :animate-being, :person, :phys-thing, :virt-obj, :virt-thing, :situation, :change, :state, :time, :output-module
Variable types	ADV	closed	pp\var	:direction, :speed

Table 6.1.: The lexical families specified by the Barrier’s grammar.

In addition to displaying the grammar in this compact tabular format, it is worthwhile pointing out and exemplifying some details:

Part of speech (POS) The following parts of speech exist: N (nouns), PRON (pronouns), ADJ (adjectives), DET (determiners), P (prepositions), V (verbs), ADV (adverbs), CON (conjunctions), PART (particles), VAR (variable labels).

Note there is a difference between, e.g., N and n: capital letters will always denote parts of speech, whereas small letters stand for atomic categories.

Category Syntactic information comes in with the *category* of a word. Also, as mentioned in 6.1.3, we can restrict the category by means of *feature structures*. For example, the requirement that the object of a transitive verb be in accusative case (np_{acc}) is encoded within a feature structure. Other constraints concern the word stem admissible for an atomic category within a complex one (e.g. $pp_{Case\text{mark}_{\text{to}}}$) and the semantic type of an atomic category ($s \setminus s_{action} / np_{time}$). Figure 6.4 shows an example of a parse of an example of the latter case.

Type conversion In addition to the lexical families listed in figure 6.1, there is an external rule allowing for the conversion of a noun (category n) into a noun phrase (category np).²⁴ This is because we do want ‘mutilated’ Pidgin-style sentences like “get banana” to be considered correct sentences, which they would not without such a rule: transitive verbs (of which “get” is one) require an accusative noun *phrase*, whereas “banana” is just a simple noun. (One would have to say “get a banana” or “get some banana”.) Through our additional rule, however, any noun becomes a valid noun phrase as well. See figure 6.4 again for an example of how this rule is applied: there the noun “\$x2 :time” is converted into a noun phrase.

We even go a step further and ignore any indefinite determiners: “get a banana” will educe the same logical form as “get banana”. So, from the implementational point of view, “get banana” is not actually short-hand for “get a banana”, instead, it would be more correct to speak of the “a” in “get a banana” as a redundancy.

Logical form To show where the *semantic* information is added, one has to look at the example of *ditransitive verbs*: these are verbs that take two objects (which is a *syntactic* requirement): a direct and an indirect one; there are two options: the objects can be plain accusative nouns appearing in the order *indirect–direct* (e.g. “buy me candy”) or the

²⁴Such rules are defined in the file `rules.xml`.

6. Natural-language processing

direct object can be followed by a prepositional phrase beginning with “to” or “for” (we call these pronouns *casemarks*; e.g. “buy candy for me”). Still, we have been operating in the syntactic realm; we get semantic only when we specify in the family’s *logical form* that the sentence that is the result of the ditransitive-verb category is an *action* and the direct object is that action’s *patient*,²⁵ whilst the indirect object is its *beneficiary*.²⁶

To illustrate the new level that is afforded by this additional information, note that the semantic level might as well be completely detached from the syntax: One could define another family of transitive verbs whose direct object is not a patient (as in our case) but a beneficiary. This would completely make sense in many cases. Here is an example: the verb “to persuade someone” (with a *direct* object) is derived from Latin “persuadere alicui” (with a dative, i.e. *indirect*, object). So, while the syntax is different in the two languages, the semantics has stayed the same, and in any case we would go astray from a fixed syntax–semantics mapping: we would either have to make the English direct object a beneficiary or the Latin indirect object a patient.

Note that, while one family may have multiple categories—each as one entry—, it has only one single resulting logical form; e.g. “buy me candy” (category s/np/np) and “buy candy for me” (category s/ppCasemark/np) will result in the selfsame logical form.

The semantics packed into a logical form can later be used when we use a category as attributes for another category. Consider the lexical family ‘Periodic interval’: it has only one member, the adjective “every”. Its categorial definition demands that there be a sentence to its left and a noun phrase to its right, plus that the sentence be of type *action* and that the noun phrase be of type *time*. *To make sense, natural-language sentences must be defined in both syntactic and semantic terms!* “Yes every flower” would from the syntactic point of view be a correct periodic interval according to our definition, it is only the semantic add-ons that constrain the family to phrases like “download e-mails every two minutes”.

Variables The use of variables has been touted several times by now; here we shall inspect how we implemented them within the OpenCCG framework: The goal is to make the sequence *label–blank–type* result in the category implied by the given type specifier; e.g. “\$x5 :person” should be of category n (noun) because all persons are nouns.

This situation calls for defining a complex category. We introduce one family for each part of speech that can possibly be taken by a *variable type* (e.g. “:action” has part of speech V, i.e. verb). This family’s category is complex; it requires a word of atomic category var (e.g. “\$x5”) to the left and yields the category we want the type to be of (in the case of

²⁵i.e. the one with whom the action is performed

²⁶i.e. the one on whose behalf the action is performed

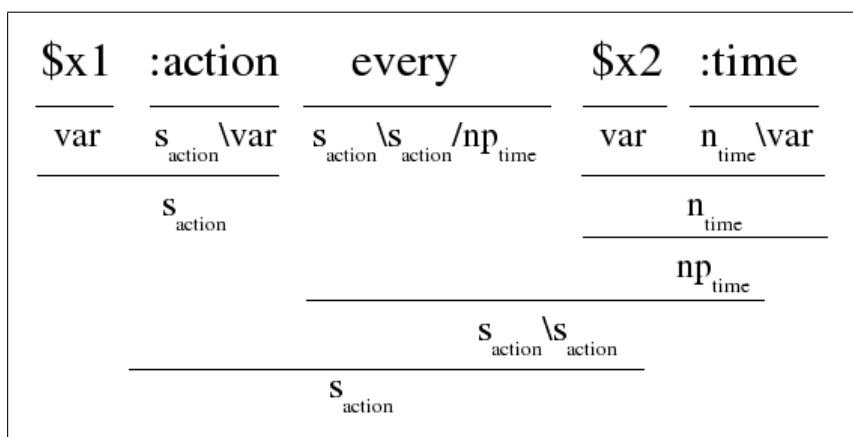


Figure 6.4.: An example parse showing how variables are implemented.

“:action”, this would be *s* for sentence). Figure 6.4 shows an example of how variables are implemented.

6.1.5. Parser

The parser we use basically consists of an OpenCCG parser (initialized with the grammar described in this chapter) with one additional filtering method: our `parse` method does not simply return all the parses retrieved by the wrapped OpenCCG parser, but only the subset containing those that are of result category *s*, i.e. that represent a sentence. As has already been said, we accept only imperatives as sentences,²⁷ which is reflected in the name of our parser class, `ImperativeParser`. The parser returns its results as a list of imperatives.

Let us briefly look at an example: The noun “message” could be parsed by the OpenCCG parser contained in our `ImperativeParser`, but the result set returned by the latter would nonetheless be empty, since there is no result with category *s* but only one with category *n* and one with *np*.

6.1.6. Technical description

All that has been said was on a rather abstract level. For reasons of completeness and to give an idea how these concepts are actually coded, we will give some examples of OpenCCG sources in their actual XML notation.²⁸

²⁷Plus short particles, like “yes” or “no”.

²⁸We deem it not worthwhile to mention every detail, such as whether an XML attribute is optional. Such information can be easily retrieved from the OpenCCG guide [BKW06].

6. Natural-language processing

```
1 <family name="noun" pos="N" closed="false">
2   <entry name="noun">
3     <atomcat type="n">
4       <fs id="2">
5         <feat attr="num"><featvar name="NUM"/></feat>
6         <feat attr="index">
7           <lf>
8             <nomvar name="X"/>
9           </lf>
10        </feat>
11      </fs>
12      <lf>
13        <satop nomvar="X">
14          <prop name="[*DEFAULT*]"/>
15          <diamond mode="noun">
16            <prop name="[*DEFAULT*]"/>
17          </diamond>
18        </satop>
19      </lf>
20    </atomcat>
21  </entry>
22 </family>
```

Figure 6.5.: The lexical family of nouns in OpenCCG.

First, let us have a look the morphological registry that contains all the vocabulary that is known to the Barrier. It contains each morphological form in a separate XML element:

```
1 <entry word="minute" pos="N" stem="minute" class="time" macros=" 2
   @numSingular_@numSingular-X" />
2 <entry word="minutes" pos="N" stem="minute" class="time" macros="@numPlural 2
   _@numPlural-X" />
```

Each form (attribute `word`) is derived from the word stem indicated by the respective attribute `stem` and takes a certain part of speech (attribute `pos`). The `class` attribute names the semantic type (one of those listed in figure 6.3). The macros referenced from within the `macros` attribute further constrain the word for parsing and constructing the logical form; e.g. “minute” can only be used as a singular, “minutes” only as a plural. This is obvious to a human, but still it has to be manifested in the code as well.

Second, for the sake of exemplifying how lexical families—the core of an OpenCCG grammar—are defined, let us first pick out an atomic category, then a complex one. Figure 6.5 shows how the noun family (category `n`) is defined:

As one would expect from an XML implementation, it requires a category that is notated

with the single letter `n` in abstract notation to become a lot more verbose. The `family` tag specifies the name of the family as well as the part of speech words must have to be qualified as members (note that the correct part of speech is a necessary, not a sufficient condition for membership). If the `closed` attribute is `true`, all the members have to be listed within the family tag, too. With nouns this is not the case, however, for anything with part of speech ‘noun’ shall be of the lexical family `noun`, too.

Each `entry` tag specifies a category that members of the family can have. As the case is very simple with nouns, we have only one such category, the atomic `n`, which is coded in the `atomcat` (for ‘atomic category’) tag. This tag contains a feature structure (`fs`) and a logical form (`lf`).

The *feature structure* can be thought of as specifying properties the word that is of the atomic category must have. Such properties may also be variables, as in the example: we specify that each noun has a feature ‘number’ (`num`), but we do not hard-code its value. The second feature, `index`, is there to introduce an identifier (`X`) that we can use in the logical form to refer back to this word.

In the *logical form* after the feature structure, we define, as it were, this word’s contribution to the resulting parse tree. The attribute value `[*DEFAULT*]` will be replaced by the word itself, e.g. “flower”. The logical forms of the single words are joined into the parse tree by the OpenCCG parser.

Now that we have seen some details in this simple example, the next step to take is to have a look at the more intricate family of transitive verbs,²⁹ whose only entry is of a complex category; the code is listed in figure 6.6.

Here the abstract categorial notation is `s/np`: a transitive verb requires a noun phrase to its right and yields a sentence. Again, this is a bit more lengthy in XML: we have to specify both `s` and `np` as atomic categories within a `complexcat` (for ‘complex category’) tag with a `slash` tag (denoting the slash’s direction—/ or \—and its mode³⁰) in between. We can see that the noun phrase has to be in accusative case, ruling out phrases like “search I”.

All member stems must be listed in `member` tags, as verb families are closed. This is because we do not just want all words with part of speech `V` (verbs) to be eligible as *transitive* verbs—some verbs will be intransitive, some ditransitive, etc. So we will have to discriminate as to what verbs are of what syntactic type. Also, family membership is not exclusive: a word can be a member of several families; e.g. the imperative verb “buy” can be both simply transitive (“buy some flowers”) and ditransitive (“buy me some flowers”).

²⁹It should be remembered that all our verbs are imperatives!

³⁰For our purposes it was not necessary to make use of the different modes OpenCCG offers. A mode encodes for associativity, commutativity, etc. of the slash operator. See [Bal02, chapter 5] for details.

6.1.7. Development history, alternatives, and venues of improvement

Alternative considerations

Although the variable concept seems rather smooth and clear as it is, the decision to solve the problem this way was not obvious. During the design of the system we had to face the question, ‘What shall the nature of variables be?’ There were basically three options:

1. Syntactic: the variable type specifies the word’s syntactic role in the sentence; e.g. “buy \$x3 :accusative”.
2. Semantic: the variable type specifies the word’s semantic role in the sentence; e.g. “buy \$x3 :patient”.
3. Semantic: the variable type specifies the word’s ontological type (as in figure 6.3); e.g. “buy \$x3 :phys-obj”.

Alternative 1 would not permit to limit the range of concrete accusative objects to a particular domain of objects, and the horrific scenario sketched on page 60 would loom again.

Alternative 2 would require application programmers, i.e. those people who eventually have to define variables, to know quite a lot about the implementation of the Barrier’s grammar, e.g. that with the word “buy”, the entity that is being bought is called a ‘patient’. To assume as much knowledge in someone who is merely making use of the services the Barrier offers him would clearly be unrealistic.

So we opted for alternative 3. It provides an intuitive and easily understandable view for the application programmer developing a Barrier-compatible solution: he just has to know the Barrier’s ontology—the simple tree of figure 6.3—instead of grammatical details, as in alternative 2. Also, it allows for restrictions with regard to the domain of objects that can possibly fill the variable slot, a conspicuous advantage over alternative 1.

Venues of improvement

The current vocabulary is rather small and mainly constrained to meet the demands posed by the prototype implementation. This is, however, just a matter of economy; adding a huge number of words that are known to never be used would afford nothing but to make the grammar less transparent. On the other side, enlarging the set of words would be possible without any conceivable drawbacks as to the system’s performance.³¹

³¹Speaking of performance, we mean ‘quality of service’. The performance as measured in processing time would obviously decrease as the size of the vocabulary is increasing.

For future, ‘real-world’ versions of the Barrier, one could think of two ways to make the vocabulary more universal:

1. Deliver the system with a complete morphological registry from the start.
2. Develop a mechanism for its dynamic extension by application programmers.

Alternative 2 would probably be the more powerful solution, for in every real-word use case there would be words that certain applications would require and that are not included in the standard thesaurus. But *vice versa*, the Barrier would have to ship with a strong vocabulary to begin with, since it would be quite cumbersome if application programmers would have to make use of the dynamic registration feature all the time.

Another point that could be improved is the nature of verbs: Currently we use imperatives only as verbs. One could, however, imagine other modes, too, that could make sense even if we allow nothing but imperatives *as top-level sentences*; as an example consider the non-imperative verb “is” in the relative clause of this imperative sentence: “give me the bun that *is* in the oven”. For the current proof-of-concept implementation we deemed it not necessary to build in support for this case.

6.2. Imperatives

6.2.1. Overview

The words ‘imperative’ and ‘script’ have been used several times by now, and it should have become clear how these concepts are used within the Barrier. Here is a short summary:

- A user utterance will be considered valid only if it is an imperative sentence.³² This sentence is represented by a data structure called *imperative*. It is basically an XML document encoding the parse tree of the sentence. Therefore, an imperative packages a user command.
- A *script* is an ordered list of imperatives.

In this chapter, we will describe in detail how these concepts have been implemented.

³²We again neglect the minor exception of particles like “yes” and “no”.

6.2.2. Description

Imperatives

A recursive structure like the parse tree that is encapsulated by an imperative is conveniently represented in XML. It is easily processable in the computer while still maintaining a certain level of readability for humans. When we talk of a parse tree we actually mean the logical form as introduced on page 63. An example of such an XML document will be given in the technical part of this section.

Most of the time, imperatives will result from parsing user-uttered, natural-language commands. Thankfully, OpenCCG provides methods for converting the parser's results into XML documents, so we can use these methods in our own wrapping parser and need not implement such a conversion ourselves.

In one particular place it will be necessary to manually edit imperatives: this is in the Rulebook, for our current implementation does not yet provide a mechanism for adding rules in natural language.³³ This is where the human readability comes in handy.³⁴

Apart from the logical form, which constitutes an imperative's core, the following data are contained in its object representation:

- As described in chapter 7, some imperatives require additional, *raw data*. For example, “set recipient” (as registered by an e-mail application) needs the recipient's e-mail address. Data of that kind is stored in a field within the imperative object. The data is called ‘raw’ because it is unparsed. It can be considered an additional *parameter* for the application.
- It will be explained in chapter 31 how applications can act just like regular users by passing imperatives to the Barrier. The mechanism requires that the *application issuing a command* be indicated in the imperative. If the imperative has been input by a user, not by an application, this field will hold the default value `de.tum.in.barrier`.
- The Matcher algorithm needs to associate information with imperative objects at runtime, e.g. whether the imperative has already been encountered during the search. Such data will also be stored in fields. Details on which fields exist, and what they are there for, will be given when we examine the Matcher algorithm.

In the Matcher algorithm we will also have to compare two imperatives for equality. To this

³³Note that Catalog entries can already now be added in natural language.

³⁴In fact, when an imperative is to be produced for the Rulebook, one will rather feed a natural sentence into a stand-alone OpenCCG parser and post-edit (if at all) the output instead of assembling a logical-form XML document ‘from scratch’. It will not be necessary for application programmers to interact with OpenCCG at all, entering new rules is strictly optional, without any loss of generality.

purpose, we simply compare the string representations of the XML documents representing their logical forms; all other information stored in an imperative will be neglected. Again, we will lay out why this makes sense in the section about the Matcher.

Scripts

‘Script’ is just a name for an ordered list of imperatives, and this is also how it is implemented: an object of class `Script` simply wraps a list of `Imperative` objects and additionally offers methods for inspecting and manipulating this list.

Barrier imperatives

Barrier imperatives are imperatives that are meant not for an application, but for the Barrier itself. This is quite similar in concept to Barrier-BSFs (see chapter 12.1), which are also BSFs, but destined for the Barrier, not for the user.

As an example, consider a command such as “exit Barrier” (the command that shuts down the Barrier). Clearly, it is not meant for an application. Obviously, it is a valid command, and as such appears in the Catalog. The difference is that when a Barrier imperative is created in the Left Automaton, it is not passed through the usual channels, but instead taken care of in a special function in each Automaton.

Since all Automata might have to process a Barrier imperative (cf. “exit Barrier”, which makes all Automata ‘clean up’ and terminate), after an Automaton is finished, it will pass the Barrier imperative to its neighboring right Automaton (or, in the case of the Right Automaton, terminate processing, since by then all Automata had the chance to process the imperative).

The vast majority of imperatives are *not* Barrier imperatives, those are a minority among their brethren.

Still, apart from “exit Barrier”, there are other Barrier imperatives, such as periodic commands (cf. chapter 30) and commands concerning the (graphical or other) user interface (UI). Both warrant a short explanation.

Periodic commands This class of commands is fully contained in that of Barrier imperatives. While they are described in more detail in their chapter (30), suffice to say that they contain a *normal* imperative, along with an interval time which denotes after what time interval that normal imperative should be executed, periodically. Since periodic imperatives are processed in the Barrier, it follows that they are also Barrier imperatives.

6. Natural-language processing

Nevertheless, the imperatives themselves that will be generated based on a periodic commands will *not* be Barrier imperatives necessarily. They, in all probability, will be destined for a normal application. Figure 6.4 shows how periodic commands are specified in the grammar.

UI commands UI (user interface) commands are commands such as “clear screen”, “redirect speech to screen” or “always reject current output”. For a list of these, confer table 18.1. Note that these are just the *stock UI commands* of the *Barrier*. For further information on them, refer to section 8.0.11. Apart from these commands, output modules can also normally register arbitrary other commands, and when invoked by the user, process them correctly. This is because while the modules (in their function as modules) are connected to the Left Automaton, their ‘control entity’, i.e. the module in its capacity as an *application*, is connected to the Right Automaton just as any other application. As further explained in 8, modules can hence register for new commands just like any other application could.

6.2.3. Technical description

OpenCCG uses as its Java API to XML a package called JDOM³⁵ (package name `org.jdom`), which provides a light-weight interface for accessing and manipulating XML documents with convenient ways of transformation to and from the standard interfaces known as Simple API for XML (SAX) and Document Object Model (DOM). We used the same software for dealing with XML data in imperatives.

Figure 6.7 shows a typical imperative’s XML representation. Note that this is the document content in ASCII, which is independent of the XML interface (such as JDOM or SAX) used.

While figure 6.7 displays an imperative’s XML data as it is later used within the Barrier, this is not the actual output of the OpenCCG parser. In order to make it look that way we first have to transform it. The most convenient way to perform such transformations on XML documents in an automated manner is applying an Extensible Stylesheet Language (XSL) transformation (XSLT for short). XSL stylesheets are themselves XML documents of a specific format, and there exist convenient Java methods for applying them to XML documents. OpenCCG even offers the possibility to specify stylesheets that are to be applied whenever an XML document is produced from a logical form and stylesheets that are to be applied whenever an XML document is to be transformed back into a logical

³⁵Available at <http://www.jdom.org>.

form.³⁶

If we do not make use of stylesheets the parser’s plain output would be as shown in figure 6.8. Our LF-from-XML stylesheet is exactly inverse to the LF-to-XML one: Whilst `make-var-nodes.xml` transforms figure 6.8 into figure 6.7, `unmake-var-nodes.xml` transforms figure 6.7 back into figure 6.8. (We need this latter direction for realization, for, of course, the realizer can only cope with the structure it knows as produced by the parser, not our customized one.)

We will now explain which changes are made by our LF-to-XML stylesheet, and why. As the LF-from-XML version does just the opposite, there is no reason to dwell on it separately.

- Remove the `target` node containing the parsed sentence; it is not needed.
- ‘Peel off’ the enclosing `xml` and `lf` tags; they are redundant since they do not carry any information.
- Pull each variable label and variable type into the parent node, so we can tell straight away that a node represents a variable without having to descend in the tree; this makes lines 11–20 of figure 6.8 collapse into line 10 of figure 6.7.
- Remove all the labels that are attributed by OpenCCG (e.g. `m1` in the `satop` node); we do not need them and they are possibly cumbersome when comparing two imperatives.
- Make the single `satop` node a `diamond` node. `satop` stands for “satisfaction operator” and is always top in the hierarchy after removing the `xml` and `lf` tags. Let us consider an example to see why we make this change: A sentence like “do *A* and do *B*” would have a ‘conjunction’ node (“and”) as root, and the two ‘action’ nodes “do *A*” and “do *B*” would be its children. So “and” would be a `satop` node, while the two actions would be `diamond` nodes. But if we parse the simpler sentence “do *A*” the ‘action’ node would be a `satop` node, meaning that it would not match the left child of the first sentence, which is a `diamond`. We do, however, require that the two match, and by making root nodes `diamonds`, too, we enforce this.
- Group sibling `property` nodes into one enclosing `properties` node. `property` nodes encode for adjectives, and this transformation will help us to define a rule (in the Rulebook) for switching the order of two adjectives.

³⁶This is done in the LF-to-XML and LF-from-XML tags of the grammar’s `grammar.xml` file.

6.2.4. Development history, alternatives, and venues of improvement

At first it was planned to exploit the labels that are attached to nodes by OpenCCG itself when implementing variables (e.g. the `n1` in line 12 of figure 6.8). But this would have involved too much tinkering. It is much cleaner to operate on a higher level, so we now just ignore the OpenCCG labels by introducing our own labeling system as described above.

6.3. Goal Test and Catalog

6.3.1. Overview

Now that we have seen how imperatives are implemented, in this section it will be explained how they are compared. Since one of the Barrier's main tasks is to forward user commands to applications (that have indicated in the Catalog that they are able to cater to these commands), comparing imperatives is a major task of the Barrier's.

The basic comparison step performed is called *Goal Test*. The key concept for a Goal Test is *derivability*. We say that an imperative I is *derivable* from an imperative J , if and only if

- I equals J , or
- J is a generalization of I , i.e. J contains variables that can be replaced by corresponding parts of I within the type constraints laid out in section 6.1.4; e.g. “eat a banana” is derivable from “eat \$x0 :phys-thing”.

The *Catalog* stores all imperatives for which applications have registered. Also, it provides a method to check whether a concrete imperative uttered by the user is contained in it. This check will employ the Goal Test just introduced as a sub-method.

6.3.2. Description

Goal Test

The Goal Test algorithm for comparing two imperatives I and J works by traversing the two parse trees (i.e. the XML representations of the logical forms) simultaneously, trying to match node for node. If all nodes match, then I is derivable from J . Breaking the above definition down to nodes rather than imperatives, we say that a node i of I matches a node j of J if

- i equals j , or
- j is a generalization of i , i.e. j is a variable of the same ontological type as i or of a supertype of i 's type. For example, i could encode “banana” and j the variable “\$x0 :phys-thing” (a banana is a physical thing).

If this node-for-node comparison is performed in a recursive manner, it implements exactly the notion of derivability presented in the overview section above. Pseudocode for the Goal Test algorithm is printed in figure 6.9. The Goal Test will always be performed to compare a concrete imperative uttered by the user to an imperative that is a Catalog entry. This is why the algorithm's two parameters are called `$inputElem` and `$catalogElem`. Note that a *user input imperative will never contain variables*. This simply would not make sense: applications that register imperatives in the Catalog may well offer a whole set of services by using variables in imperatives, but a user can only make one concrete request at a time. So when the first of the two node matching criteria listed above (“ i equals j ”) holds, both i and j will always be “all-terminal”, i.e. variable-free.

To clarify the procedure we make some annotations to single lines in the algorithm of figure 6.9:

- Line 1: e.g. a `diamond` element does not match a `prop` element.
- Line 4: e.g. a “phys-thing” variable does not match a concrete “person”, but an “animate-being” variable does match a concrete “person”; the type hierarchy referred to is the one of figure 6.3.
- Line 6: the semantic types match, so the variable matches the concrete tag.
- Line 9ff.: all the following cases look at non-variable tags only.
- Line 9: e.g. `mode="verb"` is an attribute of `<diamond mode="verb">`.
- Line 14: the tags themselves are equal, so check their children recursively.
- Line 16: all the children match, so the two elements match.

Note that a concrete element that is of a specific type does not only match variables that are of strictly the same type, but rather all variables that are of any supertype. This makes the variable mechanism even more flexible.

The `$labelTable` variable that is filled in line 7 is later used to retrieve information about which variable has been matched by which concrete element. When we look at the `Matcher` algorithm we will see how this data is being exploited. The variable is declared as

‘static’ in the pseudocode to indicate that it is maintained beyond single recursive calls.³⁷

Catalog

The Catalog’s core content is the set of imperatives that can currently be processed by the computing system. We call the constituting imperatives *Catalog entries*. Entries are added and removed in the Right Automaton, since these actions are initiated by applications, whose management is the Right Automaton’s, not the Left Automaton’s, task; the methods involved will accordingly be discussed in chapter 20.

The reason for keeping the Catalog free of any direct information on applications is so that the Barrier’s metaphor be kept clean, and the distinction between the different spheres/Automata not be blurred. Mixing such information all over the Barrier would only result in less modularity, and thus a less powerful—in the sense of harder to understand or modify—system.

Here it will suffice to delineate the function that is crucial to the Catalog’s usage in the Left Automaton: finding an entry that matches a given imperative. This is simply done by sequentially comparing the imperative with each Catalog entry by means of the Goal Test algorithm just presented: as soon as the imperative is found to be derivable from a Catalog entry, we may return this entry.

6.3.3. Technical description

The algorithm called `goalTest` here is in fact the `isDerivable` method of class `GoalTest`. The method to find out whether an imperative is matched by the Catalog is the `goalTest` method of class `Catalog`.

We have already mentioned that it is the Right Automaton that manipulates the Catalog by adding and removing entries. To clearly distribute the liabilities, there is a separate class called `RightCatalog` that is extending the base class `Catalog` with the manipulation methods; also, any information as to which application has registered for which imperatives is encapsulated in the subclass. Now, there is in fact only one single Catalog in all of the Barrier. It is shared by the Left Automaton, doing the matching of concrete user input against it, and the Right Automaton, in charge of manipulation and application management; it is, however, exclusively the Right Automaton that owns the object as a `RightCatalog`; the Left Automaton knows it merely as an object of the superclass `Catalog`. This way the Left Automaton has no way of tinkering with information that

³⁷In the actual code this is accomplished by making the variable an instance field rather than a local variable of the method.

is reserved for the Right Automaton. In particular, any information about the actual applications is strictly shielded from the Left Automaton, upholding the clear distribution of responsibilities and knowledge.

When we talk of Catalog entries as imperatives, this is in fact a simplified version of the actual implementation: As we shall later see, in the part about the Right Automaton, it is in fact possible to register imperatives in natural language. But as we have already noticed in other contexts, a single natural-language sentence may have several parses, and the parser cannot possibly know which of them is the correct or at least the best one. (Remember the “killing bears” example from 5.0.11.) Thus a Catalog entry encompasses not only a single, but possibly a whole set of imperatives, each representing one parse of the original natural-language imperative. A given imperative is then derivable from a Catalog entry if it is derivable from any of the imperatives the entry contains.

6.3.4. Development history, alternatives, and venues of improvement

If one closely examines the Goal Test algorithm, one can notice that it always immediately returns when the current node of the Catalog entry represents a variable (line 3); there are no further recursive calls. This means we assume that variable nodes have no children.— At least we simply neglect them. This way, variables can only be placeholders for atomic elements. In the overwhelming majority of cases this suffices. There are, however, cases that would result in variable nodes having children, such as “find tasty \$x0 :phys-thing”: the adjective “tasty” would be a child of the variable representing a physical-thing noun. Such a case would currently not be completely supported by the Barrier; instead, imperatives like “find apple” (without “tasty”) would also be found to be derivable from the above node. In future versions the code could be modified to also cover these special cases.

In all of our code we quite often have had to recursively traverse parse trees. As of now, each such algorithm implements the traversal mechanism anew. This is definitely not wrong, but from the software-engineering point of view it might have been more elegant to deploy the visitor design pattern (as, e.g., described in [GHJV95, page 331]): this way the recursive traversal (e.g. depth-first) would have to be implemented only once, and the algorithms would only differ in the code that is to be executed when a node is visited. In our current implementation it was more convenient to proceed the way we did it, since we used the JDOM classes to represent parse trees, which makes it more difficult to augment the code with a visitor scheme than if we had used classes implemented by ourselves. But clearly, using the robust JDOM XML classes was of greater importance than having a nice visitor pattern in place.

6.4. Matcher

6.4.1. Overview

All the tools that have been documented up to this point provide us with the ability to verify whether in the Catalog there is an entry that is a generalization of a given, concrete user-uttered imperative. For example, we could find out that the user command “download favorite song” is matched by Catalog entry “download \$x0 :virt-thing” (of course, both imperatives would at this point be represented by their parse trees, or logical forms, rather than the natural-language versions).

While this makes our mechanism already significantly more powerful than it would be without the generalization concept introduced by variables, this is not where we want to stop. When we command a human servant in natural speech we assume that she is more flexible in understanding us: we assume that she will download our favorite song even if we do not pronounce the imperative “download favorite song” *verbatim*; we expect her to understand when we use an altered formula, such as “download *my* favorite song”, “*get me my* favorite song”, or “download favorite *track*”.

This enhanced flexibility is achieved by introducing rules that modify the outward appearance of an imperative without changing its meaning. We could also put it this way: an imperative may be modified syntactically as long as it stays the same semantically. The set of such rules we call the *Rulebook*.

It turns out that rules transform the single-step comparison scenario we face when just using the Goal Test into a complex search problem. During the development of this system a big effort has been made to make this search tractable and efficient.

In this section we will describe the matching algorithm that was conceived and that is called Matcher. In the next one we will inspect the Rulebook and its algorithms.

6.4.2. Description

Although this has not been obvious from the start, it turned out that matching is a classical *search problem* of the type that is frequently encountered in Artificial Intelligence.³⁸ The definition of the problem is quite straightforward, while the actual search algorithm has some intricacies to offer. We will first give a formal definition of the search problem:³⁹

States: The set of scripts, i.e. lists of imperatives

³⁸For an exhaustive discussion of the topic, cf. for example [RN03, chapter 3].

³⁹We follow the notational convention delineated in [RN03, page 62].

Initial state:	A script containing only the imperative that is to be matched against the Catalog
Goal states:	All scripts that contain only imperatives that can be derived from Catalog entries with the Goal Test described in section 6.3
Actions:	Application of rules found in the Rulebook
Path cost:	The number of rule applications

We naturally start out with the imperative we want to match against the Catalog. We continue transforming its syntactic ‘appearance’ by applying Rulebook entries until it conforms to a Catalog entry. Conformance to a Catalog entry does not, however, mean strict equality but rather derivability as described in the section about the Goal Test algorithm.

But what if the imperative simply cannot be transformed into anything matching a Catalog entry? This will often be the case, e.g. if the user commanded “detonate an atomic bomb”, but there is—thank goodness!—no sequence of rules transforming this sentence into a script whose imperatives are part of the Catalog. Will the search then go on forever or will it be able to terminate with the result that no match is possible?

Also, even if there is indeed a matching entry in the Catalog, we might not be able to find it if we do not design the algorithm carefully: we might get stuck in a cycle before retrieving the matching entry. For example, there might be a rule translating the word “get” into the word “download” and another rule translating “download” into “get” (cf. above example in the ‘Overview’ section)—such a pair of rules absolutely makes sense if we consider the words to be synonyms of each other. Similarly, we assume there is such a pair of translation rules for “song” and “track”. Now imagine the imperative “get track” being a Catalog entry and the user saying “download song”. Clearly, the two should match.

But what might a naive search algorithm do? It might first apply the rule (“download” \mapsto “get”),⁴⁰ producing the imperative “get song”, which is not to be found in the Catalog. To complete the match, the algorithm would have to apply the rule (“song” \mapsto “track”). If it applies, however, the rule (“get” \mapsto “download”) instead, it is in the starting state “download song” again. We are stuck in a cycle because a deterministic algorithm would now chose the rule (“download” \mapsto “get”) again, then the rule (“get” \mapsto “download”), etc. Figure 6.10 illustrates this: We keep alternating between the imperatives “download song” and “get song” but we will never make the crucial step (the edge marked “!!!”) into the goal imperative that is in the Catalog (highlighted as a rectangle-shaped node). Rules that are applicable but never get applied by the algorithm are drawn in gray.

⁴⁰In fact, rules map an imperative to a *script*; this notation suggesting that imperatives are mapped to imperatives is just for the sake of simplicity.

6. Natural-language processing

Another possible cause for cycles might be rules that are ‘recursive’, e.g. (“walk” \mapsto ⟨“push left foot”, “push right foot”, “walk”⟩). In this case we could keep applying the same rule over and over again in an infinite loop. It is one question whether such rules are indeed sensible and necessary, but they clearly can easily be defined within the rule framework we will later present. So rather than forbidding the use of such rules (which would also severely hamper the generality we aim to achieve), we should make sure in the matching algorithm that they cannot do any harm.

The solution is to find a way of dealing with repeated states. This can be done by not using a tree as the underlying data structure but rather a graph. While the tree can possibly grow infinitely, the graph will always stay finite. Instead of making a fresh node for every rule application we rather first check if an equivalent node (i.e. one that represents the same imperative) has already been generated by the algorithm. If so, we simply draw an edge back into the existing node, thus detecting the cycle. The current path can then be discarded, and we can explore further. This is graphically demonstrated in figure 6.11: The cycle is detected and can consequently be avoided; the rectangle-shaped goal imperative is found. Rules that are never applied and imperatives that are never generated are drawn in gray again.

After these preliminary notes we will now present the pseudocode for the Matcher algorithm. Figure 6.12 contains a listing of it.

In the subsequent ‘Scenario’ section we will explain the algorithm’s working principle in detail at the concrete example of figure 6.13, whereas, in the following paragraphs, we will give a rather abstract description (also with the help of figure 6.13).

The algorithm implements a *depth-first graph search*. The nodes of the graph, i.e. the states of the search problem, are scripts. This is indicated by their record-like shape: each node may contain several imperatives. Edges do not, however, link a whole script to a whole script, they rather lead from an imperative (i.e. a ‘sub-script’) into a script; this is due to the fact that rules (which is what edges stand for) map from imperatives to scripts.

It is important to note that, while the picture shows a tree structure, we in fact deal with a graph: each of the nodes imp_1 , imp_3 , and imp_4 appears twice, and we consequently do not maintain duplicate copies but rather point to the same object. In the image we had to display the nodes twice to clearly show which of them belong to the same script. In reality, edge r_1 leaving imp_4 leads back into imp_3 one level higher, thus closing a cycle.

- When does an imperative match the Catalog?—When the starting script (imp_0 in the picture), consisting of just that imperative, is matched. So the algorithm must be started with a script that contains just the imperative to be tested.
- When is a script matched?—When *all* its imperatives are matched. All parts of

a command must be executable for the whole command to be executable. This is why line 1 iterates over all imperatives; if and only if the loop is not terminated by returning `false` do we return `true` after the loop (line 30).

- When is an imperative matched?—When *there exists* a rule such that the script that results from applying the rule to the imperative is matched. This recursive definition is manifested in the code in line 17: the algorithm calls itself with the successor script. In line 13 we loop over all rules that are applicable; as soon as a recursive call returns `true` we know that the imperative is matched (line 18).⁴¹

In summary, the loop of line 1 implements an all-quantifier (\forall), while that of line 13 implements an existential quantifier (\exists). The outer loop can be terminated *unsuccessfully* as soon as one iteration is unsuccessful (as $\exists x \neg P(x)$ is equivalent to $\neg \forall x P(x)$); the inner loop can be terminated *successfully* as soon as one iteration is successful. Thus the pattern of a ‘quantifier alternation’ arises:⁴² *all* imperatives must be matchable by *any* rule that produces a script whose imperatives must in turn *all* be matchable by *any* rule, etc.

This being the basic functionality of the Matcher, attention must be paid to several details:

- By simply switching from a tree- to a graph-based approach we have not yet tackled the problem of cycles; we have merely laid the basis, but the actual loop detection must still be built in. This is done by endowing imperative objects with a Boolean field `onCurrentPath` (initially `false`).

The idea is as follows: a cycle is closed when an edge leads into a node that is on the path from the start node to the current node. If we followed such an edge, we would go back into a state which we have already visited but whom we have not yet fully inspected (since we are still in a recursive call that was started at that very node); in figure 6.13, e.g., *imp*₄ would lead back into *imp*₃. As the algorithm is deterministic this would mean that we would again get into the current state, which would in turn take us back to that earlier state, etc. In the picture, we would alternate between *imp*₃ and *imp*₄. So we can detect cycles by flagging exactly those nodes that are on the current search path. This is achieved by setting the node’s `onCurrentPath` attribute to `true` before the node is expanded (line 12) and by resetting it to `false` when we have backed up from all the recursive calls on that node (line 24).

We may think of `onCurrentPath` as a lock variable: before expansion, a node is locked, afterwards it is unlocked again. Now, if we find out that *any* of the current

⁴¹In such a case of success we could actually break from the loop immediately. When we keep looping over the remaining rules this is just to be able to find the rule that yields the *best* resulting script. In our current implementation we, however, stop as soon as we have found one succeeding rule.

⁴²This quantifier alternation was inspired by the concept of an *alternating Turing machine*; cf. [Pap93, pages 399–401].

6. Natural-language processing

node's imperatives lies on the current search path (line 15), we may consider the rule that produced the script as having failed (and try the next rule, line 16), for we would have to match *all* imperatives of the script, and if just *any* of them gets us stuck in a loop we would be doomed to never terminate.

- Before we actually start expanding a node into its successor scripts by applying rules from the Rulebook,⁴³ we can exploit information that we have gathered in earlier stages of the algorithm. That is why we store, with each imperative, the field `matched`; it can have three values: `true`, `false`, and `unknown`, indicating whether this particular imperative has already been found to be matchable; the default is, of course, `unknown`.

In lines 26 and 29, respectively, the values are set depending on the results of the recursive calls. The 'grounding case' is in line 7: if an imperative can be directly (i.e. without any further rule applications) derived from the Catalog by means of the Goal Test method described in section 6.3.2,⁴⁴ we may also set `matched` to `true`. Whenever we find `matched` to be `true` for the imperative to be tested (lines 3 and 7), we may, without applying any rules, continue and consider the next imperative of the current script. This is either because we can directly match a Catalog entry or because we have previously found the imperative to be matchable. If we, however, find `matched` to be `false`, we may return `false` for the current call to `Matcher` since, if one imperative is unmatchable, so is the whole script.

Note that everything from line 10 on is executed only if `matched` is `unknown`: if we know the status of a node already there is no need to further examine it.

- The return value of the `Matcher` is of Boolean type: it only tells us whether the initial imperative can be matched against the Catalog. What we are, however, actually most interested in is the *matching script*, i.e. the script that is the result of applying rules to the initial imperative and of which all imperatives are directly derivable from Catalog entries. All imperatives that are part of the matching script can be passed to the applications that registered for them, since they are now in the form that was used by the applications themselves for the very process of registration.⁴⁵

Each imperative has its matching script stored in the field `matchingScript` (initially `null`), and the matching script of the imperative in the starting script is what is of interest to us. The attribute is manipulated in two places: line 8 is the terminating case: the imperative is found in the Catalog, so the matching script may be set to a

⁴³The call to `apply` in line 14 refers to the algorithm of figure 6.15.

⁴⁴Here we refer to the *Catalog's* Goal Test, which calls the actual Goal Test repeatedly for all Catalog entries.

⁴⁵Of course, an application may have registered for a more general form by using variables.

singleton script consisting of only the imperative itself. The recursive case is in line 23: if an imperative was matched after being transformed into a script by applying a rule then the matching script is the concatenation of the matching scripts of the resulting script's single imperatives.

This may be described as a *divide-and-conquer* approach: an imperative is divided into several imperatives (a script), for each of these subproblems we find the matching script recursively, and finally the solutions to the subproblems are concatenated to 'conquer' the whole problem.

6.4.3. Scenarios

To make the way the Matcher works more conspicuous we will now go through the example of figure 6.13. Some preliminary notational issues: as already mentioned, nodes (scr_m) represent scripts; subnodes named imp_n represent the imperatives that form a script. Edges, labeled with rule names r_i , represent rules. The special circular nodes labeled “/” stand for “failure”, i.e. the respective rule is not applicable. A “+” sign within an imperative means the imperative has been successfully matched, a “—” sign means it could not be matched. If there is neither of those two signs the imperative has not even been visited in the course of the algorithm.

We aim to find out whether the imperative imp_0 can be matched against the Catalog. The Goal Test of line 6 fails for imp_0 , meaning that there is no such Catalog entry. We assume there are three rules in the Rulebook, r_1 , r_2 , and r_3 . For each of them (line 13), we test whether its applications yields a script that is matchable, thus making imp_0 itself matchable (this will not be the case). We start out with rule r_1 , which cannot be applied. Rule r_2 , on the contrary, can be successfully applied and produces a successor script $scr_1 = \langle imp_1, imp_2, imp_3 \rangle$.

We now skip several steps, assuming that we recursively find out that imp_1 can be matched. This is why it is marked with “+”. Since all imperatives of the script must be matched for the script to be matched, we next check imp_2 , assuming the Goal Test for it in line 6 is positive (there is a corresponding entry in the Catalog, possibly more general due to variables); hence the “+” sign.

So line 9 causes us to immediately look at the next imperative imp_3 . It turns out that rules r_1 and r_2 are not applicable, but that r_3 results in script $scr_2 = \langle imp_1, imp_4, imp_5 \rangle$, which is to be examined next (remember that the order of traversal is depth-first). Imperative imp_1 need not be checked recursively, as line 2 tells us that it has previously been matched successfully.

6. Natural-language processing

We can proceed to imp_4 . Only r_3 applies, resulting in script $scr_3 = \langle imp_3 \rangle$; as imp_3 is on the current search path (in scr_1), we are caught in a cycle and lines 15/16 make us escape from it. So imp_4 can be marked as unmatchable, whilst imp_5 need not even be considered: if one imperative cannot be matched the whole script cannot be matched (analogous to the logical operator \wedge).

We back up, labeling imp_3 in script scr_1 as unmatchable.⁴⁶ Thus, scr_1 is not matchable and the only chance for imp_0 to be matched is for r_3 to yield a matchable script; r_3 , however, results in scr_4 , whose only imperative imp_4 has been previously found to be unmatchable; so line 5 makes the algorithm return **false** for scr_4 , resulting in the final top-level result **false** (line 27): scr_0 and thereby imp_0 cannot be matched against the Catalog.

6.4.4. Technical description

In the Java code, the Matcher is implemented by the method `matchRec` in class `Matcher`.

As is often the case, lots of details had to be considered at implementation time. Here we shall just outline the most notable ones:

- For further processing, the matching script alone does not suffice: what we need is the exact Catalog entry that corresponds to each imperative in the matching script. This is why `Imperative` objects have a field named `matchingCatalogEntry` storing this piece of information.
- It has been pointed out several times by now that, by containing variables, a Catalog entry may be more general than the concrete imperative it matches. So just storing the `matchingCatalogEntry` in an imperative is not enough: we also need information about which variable is to be replaced by which concrete element. For this, imperatives also feature a field `labelTable`, a mapping from variable labels to the elements that match them in the user utterance.

For example, if the user said “eat banana” and the Catalog contains, among others, $e_1 = \text{“peel } \$x1 \text{:phys-thing”}$ and $e_2 = \text{“bite into } \$x0 \text{:phys-thing”}$ then the matching script would be $\langle \text{“peel banana”, “bite into banana”} \rangle$; those two imperatives would store pointers to e_1 and e_2 , respectively. The label table would be $\{ \$x0 \mapsto \text{banana}, \$x1 \mapsto \text{banana} \}$.

- It has already been outlined how we store pointers to the same imperative objects to keep track of repeated states. Technically this is not straightforward feasible, for, in order to find out whether an imperative has already been generated by the algorithm, we must first make sure that imperatives are comparable in a sound manner.

⁴⁶ imp_3 becomes labeled simultaneously in scr_3 , since the two actually are the same object.

Obviously it does not suffice to just compare the objects themselves, for rule application produces a fresh object every time, and when simply comparing objects, Java will just compare the objects' memory addresses, which will always result in the objects being unequal. So one must override the `equals` method `Imperative` inherits from `Object`. We can then pool all the imperatives that are generated by one run of the Matcher in a set G , which rules out duplicate items. After generating the successor script in line 14 we call a method `reuseGeneratedImperatives` that checks for each member imperative whether an equivalent is already contained in G , and, if so, replaces it by the previously generated one.

When do we consider two imperatives equal, i.e. how do we implement `equals`? Surely we should not consider the kind of additional information that was described in the chapter about imperatives (e.g. the label table). The relevant information should be the one that is already contained in its natural-language form, i.e. the parse tree (the logical form). It turns out that to compare the two logical forms we need not even traverse the parse trees recursively, it suffices to compare the string representations of the corresponding XML documents for equality.

- Look at the logical form of the noun “e-mail”, displayed in figure 6.14:

The element `<prop name="e-mail" />` appears twice. It is generated like this by the OpenCCG parser, and we cannot just remove the redundancy in the XSL transformation described in section 6.2.3, because this structure is necessary for realization of the element (OpenCCG requires this).

Without paying minute attention there is the possibility of inconsistencies that could arise when applying rules to such an imperative: we could incur a problem if there is a translation rule for “e-mail” (say, to “message”) and if only one of the two nodes were translated into “message” before all of the code in figure 6.14 is successfully matched against a variable. This would lead to inconsistencies because such a ‘chimeric’ node just does not make sense. Here is an example:⁴⁷ “compose e-mail/e-mail” \mapsto “compose e-mail/message” \mapsto “make e-mail/message”, which could be matched against a Catalog entry “make \$x0 :virt-thing”.

The solution is to remember within the Matcher algorithm (in a persistent variable) which was the last rule applied and apply it first in the next recursive call; this way we will translate all the “e-mail” nodes into “message” nodes in a row without applying any other rules in between and will thus leave such ‘chimeric’ states immediately after entering them.

⁴⁷The notation “ X/Y ” means that one of the two nodes is `<prop name="X" />` and the other one `<prop name="Y" />`.

6.4.5. Development history, alternatives, and venues of improvement

Venues of improvement

When looking for a matching script in our current prototype implementation we simply use the first one we can find during the search. Note that the algorithm in figure 6.12 is more sophisticated in that respect: it finds all matches and returns the best one of them. There may, however, be better ways to do this: our approach is depth-first and it is a well-known fact that depth-first search is not an optimal algorithm,⁴⁸ i.e. we might not find the matching script that requires the fewest rule applications. Probably the most popular optimal search algorithm is A*, but for it to work we need an admissible heuristic, i.e. a path cost function that never overestimates the number of rules that must be applied to reach a goal state. It is not obvious how this can be achieved.

Even if one could come up with such an admissible heuristic and thus always find the matching script that requires the fewest rule applications, this need not necessarily mean the ‘intuitively optimal’ script is found: some rules might transform the imperative in a stronger way than others. Consider e.g. two rules applicable to an imperative, one mapping it to one single imperative, one decomposing it into a script of three imperatives. Then one would probably think of the second rule as making a bigger difference than the first rule.⁴⁹ So one would have to adapt the path cost function: instead of attributing uniform cost to each rule, the cost of applying a rule would have to reflect the ‘amount of change’ the rule affords.

So if one wanted to go this way one could either hardcode the path cost connected to each rule—which is definitely not a wise approach—or one could try learning it. Using neural nets would be the first approach to be tried: the input neurons would represent different features of the rule (and possibly of the imperative to which it is to be applied), and in an off-line training process the net weights would be learned so that the output neuron returns a number that reflects the cost of applying the rule. Learning could even be continued on-line, but this would be a major change to the current system, since we would have to integrate a way for the user to provide feedback about the quality of the Barrier’s reaction to her command input. In addition, the quality will depend on vastly more mechanisms than just which rules were applied.

With a modified path cost function one could either implement A* search or stick to the idea incorporated in the pseudocode of figure 6.12, i.e. traversing the complete graph and

⁴⁸cf. [RN03, page 81].

⁴⁹It has been noted that rules are considered to imply only structural as opposed to semantic transformations. But even as the meaning of a command stays the same, there might be more or less diligent ways to execute it. This may also depend on user preferences.

then sift through all the matches to find the best one. The latter, although computationally more expensive,⁵⁰ allows for an even broader spectrum of approaches: one would not only be able to consider the amount of changes made by single rule applications but also to take into account the ‘overall image’ of change (e.g. the number of *different* rules applied in the whole match). That is why this more general idea was put into the pseudocode.

Development history

A lot of development time has been spent towards a sound mechanism for matching user input against the Catalog of possible commands. On this way many ideas appeared, of which some proved to be good and helpful while some turned out to be fallacious and had to be abandoned. We shall put forth just two examples for this, the first an intricacy in the Matcher’s implementation, the second a general design issue:

- We have explained how we ‘lock’ and ‘unlock’ imperative objects in the search by labeling them as lying on the current search path. *A posteriori* this may seem a straightforward approach, but in fact we first tried something different—something that seemed canny to begin with and that worked for the first few dozen test examples but that all of a sudden turned out to be flawed: the idea was to detect cycles without even adding a field `onCurrentPath` to the `Imperative` class.

Instead of recognizing a cycle by checking whether `imp.onCurrentPath == true` (as is done now), we checked for the condition `imp.matched == unknown`, thinking this would implicitly ‘simulate’ the explicit `onCurrentPath` variable; the condition seemed sensible since it implies that `imp` has already been generated but has neither been matched nor been found to be unmatchable yet. It would correctly find the cycle in the example of figure 6.13; `imp3` is generated for the second time, but its matching status has not been determined yet.

But now, to see the flaw, substitute `imp5` for `imp4` in `scr4`: then, when visiting `scr4`, we would detect a cycle (`imp5` was previously generated in `scr2` and its matching status is unknown)—although there is in fact no cycle! The problem arises because we quit examining a script (`scr2`) as soon as one of its imperatives (`imp4`) is unmatchable. (If we nonetheless inspected all the remaining imperatives, the idea would actually pull through.)

It turns out that `imp.matched == unknown` is a weaker condition than `imp.onCurrentPath == true`; the former is implied by the latter, but they are not equivalent.

⁵⁰ Note that the *worst-case* complexity of the Matcher, being a search algorithm, will always be exponential, even if one may reduce the *average-case* complexity with good heuristics.

6. Natural-language processing

- Another thing that seems obvious retrospectively is the simple fact that matching is a search problem. The first approach was a far less rigorous one, trying to cope with problems specific to our task of matching sentences against a Catalog instead of abstracting from the peculiarities and finding a solution in the realm of graph theory.

For example, it was first envisioned to have different types of rules, each of them connected to a certain way of dealing with them. This, however, is very inflexible in terms of modification (whenever a new rule would be introduced, one would have to decide upon its type or even define a new one). Generally, before the matching procedure took its current sophisticated (yet, what is important, unified) form, whenever one pulled at one end of the mechanism design, some things would not fit any more at the other end.

6.5. Rulebook

6.5.1. Overview

In the previous chapter we have seen how rules are used to transform user utterances in order to match them against the imperatives stored in the Catalog. We have, however, treated the Rulebook as a ‘black box’ up to now, with just an intuitive understanding of what rules do: they change the outward form of an imperative without distorting its meaning; i.e. rules operate on the syntactic rather than the semantic level.

In this chapter we will inspect the Rulebook’s setup and working in more detail. For short, the Rulebook is a set of *rules* (including methods for adding rules and writing itself to persistent memory through serialization). A rule is in turn a pair $(lhs \mapsto rhs)$,⁵¹ with *lhs* an imperative and *rhs* a script; so, more formally, the Rulebook may be considered a mapping from the set of imperatives to the set of finite sequences of imperatives. Rules may range from simple single-word translations (e.g. “download” to “fetch”) to complicated transformations involving the whole sentence structure. We have, in the course of this report, encountered several examples for rules; here is another one: (“compose e-mail to professor” \mapsto ⟨“compose e-mail”, “set recipient to professor”⟩).

The idea behind rules is to enhance the level of flexibility: the Barrier’s rationale is to allow for a more natural way of communicating with the computing system. This would not be possible if the user still were to know the imperatives that applications can cope with *in their exact form*, like an ancient shaman, not rewarded with an obedient system

⁵¹lhs = left-hand side, rhs = right-hand side

unless she memorizes commands syllable by syllable. However, by means of the Rulebook, the Barrier has a set of transformations it can apply to reshape imperatives, and the user can pronounce an imperative one way, while applications expect it another way; still, the match will be possible. This is crucial in natural communication: When someone wants to buy a glass of wheat beer in a Munich beer garden his order “Ein Weizenbier bitte!” will be understood even if the menu actually offers “Weißbier” instead of “Weizenbier”.

The positive consequences of allowing for scripts instead of single imperatives on the rhs of a rule should be pointed out: after decomposing a single imperative into several imperatives (the Matcher will do this by applying such rules), these are treated as though they had been sequentially entered by the user herself. They may be assigned to *different applications* each, depending on which application is deemed to be best suited for each job. So the user utters in fact only one single command, yet a multitude of applications may be involved in executing it! The user need not care about which subtask is given to which application, this is done by the Barrier for him transparently.

Factoring out such transformations into the Rulebook instead of leaving them to applications has several advantages:

- Application programmers need not entangle themselves with thinking about how users could possibly utter their wishes. Laying such a burden on them would be tedious and unrealistic, but above all it would infringe one of the Barrier’s tenets: user communication issues shall be hidden from applications!
- Rules are not stored in multiple copies in each application; they are centrally stored and managed. This diminishes the amount of redundancy. It might be possible that some rules do not apply generally but just in the context of single applications. Such rules must not be present in the Rulebook, they must indeed be maintained and applied by the respective applications themselves (an extremely unlikely case, as the Rulebook can store very rare rules just as well).
- If defining and applying rules were a task of applications they would be hidden from the Barrier. (As stated in the first item of this list, we want exactly the opposite: they shall be hidden from applications.) Then we could not exploit them in the Matcher, as delineated in the previous chapter. The whole matching process would be far less powerful.

6.5.2. Description

Rules

We have already mentioned that a rule maps an imperative to a sequence of imperatives. So this is exactly what a rule object stores: an imperative (the left-hand side or lhs) and a script (the right-hand side or rhs). Additionally the class `Rule` implements the methods for rule application which are to be described next.

Just as Catalog entries, rules may contain variables. When a rule is applied we store which variables match which concrete elements of the lhs; we can then use this information to resolve occurrences of the same variables on the rhs. This is a major advantage since it possibly makes one single rule applicable to a large range of imperatives.

Note that, while user utterances may only be of result category ‘sentence’, the sides of a rule may also be a less complete element, e.g. a noun. So referring to the lhs as an ‘imperative’ may actually seem a bit awkward. We simply stick to the nomenclature because we use the same data structures for both cases.⁵²

Before proceeding to how rules are actually applied we will briefly list the constraints rules must abide to:

1. Every variable that appears on the rhs must also appear on the lhs of a rule. Without this restriction variables could simply pop up on the rhs without any correspondence on the lhs. It would then be impossible to know the concrete element that is to replace that variable. This, however, would not make sense since all user input must always be variable-free; users cannot utter variables, and it would be nonsensical if variables could spontaneously appear afterwards by applying a rule. Note that it might on the contrary be sensible that, *vice versa*, a variable appearing on the lhs does not appear on the rhs any more: it is a way to delete information that is deemed irrelevant from the user utterance. While the user cannot say *variables*, she certainly can say *values* for these variables, and these would then vanish if such a rule was applied to the user command.
2. Since for rule application we use the Goal Test also used for Catalog matching, the constraint already mentioned in section 6.3.4 applies: variable nodes of the parse tree are expected not to have children. If they do, the children will simply be neglected.
3. *Decomposition rules*, i.e. rules whose rhs consists of more than one imperative, are applicable to the root of an imperative only. Such rules are meant to split

⁵²The phenomenon of referring to something more general by using the name of a special case is quite frequent in naming conventions—it is called *pars pro toto*: e.g. many people refer to the Netherlands as ‘Holland’, which is in fact just one part of the Netherlands.

an imperative into several sub-actions that will, if executed sequentially, fulfill the demand expressed by that imperative. Thus a decomposition rule makes sense only if applied to the whole imperative, i.e. to its root. We have already seen an example for a decomposition rule in this chapter: (“compose e-mail to professor” \mapsto ⟨“compose e-mail”, “set recipient to professor”⟩).

Rule application

When is a rule applicable to an imperative?—When the lhs appears in the imperative. This is analogous to the applicability of production rules in formal grammars: there a rule $A \rightarrow B$ can be applied to words xAy , i.e. to words that contain the lhs of the rule. The resulting word is xBy , and this is just what is the case with Barrier rules: applying a rule yields the initial imperative with one occurrence of the lhs replaced by the rhs.⁵³

The difference between rules in formal grammars and in the Barrier is that in the former, the words that are modified are ‘flat’ character strings, while the imperatives we use in the Barrier are hierarchically structured in a tree-shaped logical form (that is, the parse tree). So our applicability criterion has to be slightly altered and reads like this:

A rule is applicable to a concrete user input iff a subtree of the parse tree of the user input is derivable from the rule’s lhs in terms of the Goal Test.

This definition takes into account both the tree-like structure of imperatives (“subtree”) and the fact that rules may contain variables (“derivable”). “Goal Test” refers to the mechanism described in section 6.3. So on a high level the algorithm for applying a rule works like this:

- Traverse the imperative (i.e. the parse tree) recursively;
- during the traversal, check for each node whether it is derivable from the lhs of the rule by applying the Goal Test;
- if the Goal Test is positive for a subtree S , substitute the rhs of the rule for S in the imperative;
- before this substitution, if there were variables in the lhs, use the label table filled by the Goal Test to resolve the variables in the rhs of the rule.

To see why we need the last item in the list, consider imperative “go left” and rule (“go $\$x0$:direction” \mapsto ⟨“drive $\$x0$:direction”⟩): we must first replace “ $\$x0$:direction” with “left” in the rhs before substituting the rhs for the lhs in the imperative.

⁵³Additionally we must pay attention to variables, which may occur on both sides.

Figures 6.15 and 6.16 display the pseudocode for the rule application methods. It is `apply` of figure 6.15 which is called in line 14 of the `Matcher` algorithm in figure 6.12. The other method, `applyRec`, is only called from within `apply`, which first checks whether the rule is a decomposition rule (line 1); as such rules are applicable at the root only, there is no recursion involved and we may simply execute steps 2 through 4 of the above prose description. If the rule maps an imperative to a singleton script (consisting of only one imperative; line 10) then `apply` simply calls `applyRec` and returns the resulting imperative (wrapped as a singleton script). The recursive function `applyRec` implements all lines of the abstract prose description.

The `goalTest` method of lines 2 and 1, respectively, refers to the algorithm of figure 6.9.

6.5.3. Scenarios

In this section we shall list the rules that are currently used in our implementation. We will give the rules in the more intuitive notation employed up to now in this chapter, although they are in fact more sophisticated imperative and script objects, of course.

1. *Switch the order of two subsequent adjectives*: The order does not affect the meaning, what counts is merely which adjectives are there; e.g. “hire a yellow big taxi” has the same meaning as “hire a big yellow taxi”.⁵⁴
2. *Decompose “and”*: In our grammar this conjunction always combines two sentences (cf. table 6.1); so combining the two with “and” means the same as passing the two sentences to the Barrier separately; e.g. “seek and destroy” has the same meaning as the script ⟨“seek”, “destroy”⟩, which is the result of this rule. One could argue that it would be a simple matter for the user to utter two separate imperatives in the first place, but again: the Barrier’s goal is a natural way of communicating with machines, and for most humans it is natural to join sentences with conjunctions.
3. *Single-word translations*: Users might not always ‘hit’ the exact word that applications used when registering for an imperative. Translation rules ‘smooth’ over such discrepancies that are, from the intuitive point of view, irrelevant because they preserve the meaning of a sentence. Such rules might contain *synonyms* (in our case e.g. “message” and “e-mail”; synonyms are implemented by having two rules, one in each direction) as well as *hypernyms*;⁵⁵ in such cases one will have only one rule transforming a hypernym into a more special word (the *hyponym*), e.g. “fetch” into

⁵⁴This is why we combine `property` nodes of the parse tree by making them the children of a `properties` node in the XSL transformation described in 6.2.3.

⁵⁵A hypernym is a word whose “meaning encompasses the meaning of another word of which it is a hypernym”; cf. <http://en.wikipedia.org/wiki/Hypernym>.

“download” (downloading is always an act of fetching, but not every fetching action is downloading something; so the meaning is only narrowed down, not changed, by the rule).

4. *Decomposition rules:* e.g. (“compose \$x1 :virt-thing to \$x2 :person” \mapsto ⟨“make \$x1 :virt-thing for \$x2 :person”, “send \$x1 :virt-thing”⟩).

One can think of a multitude of further rules; e.g. resolving adverbial phrases as in (“get \$x0 :phys-thing in \$x1 :phys-thing” \mapsto ⟨“move to \$x1 :phys-thing”, “get \$x0 :phys-thing”⟩); e.g. (“get pretzel in bakery” \mapsto ⟨“move to bakery”, “get pretzel”⟩).

6.5.4. Development history, alternatives, and venues of improvement

Development history

We started out by defining a Rulebook containing decomposition rules only, and separately we had a list of translations. There was quite a hassle as to the order in which these different transformation types had to be applied to make sense; also the two could not be strictly distinguished: the lhs of a decomposition rule could have another name (although not another meaning) than the user’s utterance; then one would have to translate this lhs first. Again, only after a long time of entanglement in mind-boggling details did we take the *a posteriori* obvious step of considering translations rules, too, and combining them into one unified rule application mechanism.

Venues of improvement

Dynamic Rulebook extension As of now the rules come hard-coded in our implementation. It is, however, no problem at all to build in the possibility for applications to register new rules autonomously; this would be completely analogous to how applications can register for Catalog entries already now. So implementing the same mechanism for the Rulebook would not have demonstrated any new abilities, which is why we spared it.

Also, it would not yet be convenient for applications to add rules because, as has been mentioned in section 6.2.2, rules cannot be added in natural language yet. One must rather pass a pre-parsed sentence to the Rulebook (the parse can be obtained using the stand-alone OpenCCG command-line parser). This limitation is due to the fact that the syntactic and semantic structure is of big importance when defining a rule; one sentence may, however, have several such structures (because it may have several parses). It would then not be clear which one is to be chosen. This is not as much of a problem with Catalog entries, there we may simply store all the different parses and try them all when matching;

6. Natural-language processing

with rules it is more subtle because amongst all the parses there might be some that are not expressing what we actually meant in natural language and we might thus end up with rules that have unintended consequences on how a user utterance is interpreted. For this reason, the current prototype implementation requires the parse trees of both the lhs and the rhs to be defined directly.

Inter-language translations As the Rulebook allows for intra-lingual single-word translations, one could also conceive of a translation between languages (inter-lingual). Imagine the Barrier’s internal language being English, but the speech input module recognizing German sentences; then a German–English translator could be run before proceeding. Such a tool could be used by the Barrier as a black box. With such a block-box oracle, the Barrier could easily plug in the newest/most powerful translator. One could consider inter-language translation as one single rule, and translating from the input language to the Barrier’s working language would not be a specialized process but rather part of Catalog matching. Due to the exhaustive nature of the search implemented by the Matcher it would not even be necessary to know what the input language is: as the Matcher tries all rules (i.e. the inter-language translator for each known language), it would be sure to find the correct translation.

One could even conceive of applications using yet another internal language, say Italian. This would be made possible by the same mechanism: it would suffice to apply an English–Italian translator as the last rule before matching against the Italian Catalog entry. Note that for this particular inter-language feature to work, we would need parsers for the other languages as well—otherwise we could not know whether the Italian Catalog entry is grammatically correct, which is necessary for it to be admitted for registration.

Anti-decomposition rules In some cases, it might make sense for rules to depend not only on a single imperative to trigger, but on a list of them, i.e. a script. Although the transformation process is always *started* with a single imperative (namely the one just arrived and about to be matched), as the matching progresses, there can certainly be several imperatives that all need to be matched (if a decomposition rule was applied). Then, it could make sense to have applications put them back together again in other combinations. This might enable matches that would otherwise be missed. Therefore, for the sake of completeness, this is planned to be added in a future build of the Barrier.

As an example, consider an imperative “buy me a pretzel”. There might be a rule decomposing it into the four-imperative script: ⟨“go to the shop”, “find the salesperson”, “purchase a pretzel”, “take the pretzel to me”⟩. Imagine, then, that still there is no match. However, there might be another rule which recognizes just the operations in the shop, and

transforms “find the salesperson” *and* “purchase a pretzel” back into the single command “buy pretzel in the shop”, which matches to a registered imperative “buy \$x1 :phys-thing in the shop”.

Nothing about the remaining two imperatives said, this facilitates a matching where there potentially would have been none. It is therefore not only an optimization, but an extension to include such a type of rule.

6. Natural-language processing

```

1 <family name="verb.transitive" pos="V" closed="true" indexRel="verb">
2   <entry name="verb.np">
3     <complexcat>
4       <atomcat type="s">
5         <fs id="1">
6           <feat attr="index">
7             <lf>
8               <nomvar name="V"/>
9             </lf>
10            </feat>
11          </fs>
12        </atomcat>
13        <slash dir="/" mode="."/>
14        <atomcat type="np">
15          <fs id="2">
16            <feat attr="case" val="accusative"/>
17            <feat attr="index">
18              <lf>
19                <nomvar name="X"/>
20              </lf>
21            </feat>
22          </fs>
23        </atomcat>
24        <lf>
25          <satop nomvar="V:action">
26            <prop name="[*DEFAULT*]"/>
27            <diamond mode="verb">
28              <prop name="[*DEFAULT*]"/>
29            </diamond>
30            <diamond mode="sentenceMode">
31              <prop name="demand"/>
32            </diamond>
33            <diamond mode="patient">
34              <nomvar name="X"/>
35            </diamond>
36          </satop>
37        </lf>
38      </complexcat>
39    </entry>
40    <member stem="buy"/>
41    <member stem="rent"/>
42    ...
43 </family>

```

Figure 6.6.: The lexical family of transitive verbs in OpenCCG.

```

1 <diamond mode="action" class="action">
2   <nom name=":action" />
3   <prop name="make" />
4   <diamond mode="verb">
5     <prop name="make" />
6   </diamond>
7   <diamond mode="sentenceMode">
8     <prop name="demand" />
9   </diamond>
10  <diamond mode="beneficiary" class="person" var="true" label="$x2" />
11  <diamond mode="patient" class="virt-thing" var="true" label="$x1" />
12 </diamond>

```

Figure 6.7.: The logical form resulting from parsing the sentence “make \$x1 :virt-thing for \$x2 :person”.

6. Natural-language processing

```
1 <xml>
2   <lf>
3     <satop nom="m1:action">
4       <prop name="make"/>
5       <diamond mode="verb">
6         <prop name="make"/>
7       </diamond>
8     <diamond mode="sentenceMode">
9       <prop name="demand"/>
10    </diamond>
11    <diamond mode="beneficiary">
12      <nom name="n1:person"/>
13      <prop name=":person"/>
14      <diamond mode="noun">
15        <prop name=":person"/>
16      </diamond>
17      <diamond mode="label">
18        <prop name="$x2"/>
19      </diamond>
20    </diamond>
21    <diamond mode="patient">
22      <nom name="n2:virt-thing"/>
23      <prop name=":virt-thing"/>
24      <diamond mode="noun">
25        <prop name=":virt-thing"/>
26      </diamond>
27      <diamond mode="label">
28        <prop name="$x1"/>
29      </diamond>
30    </diamond>
31  </satop>
32 </lf>
33 <target>make $x1 :virt-thing for $x2 :person</target>
34 </xml>
```

Figure 6.8.: The logical form resulting from parsing the sentence “make \$x1 :virt-thing for \$x2 :person” without applying the XSL transformation.

Algorithm: goalTest(\$inputElem, \$catalogElem)

Input: \$inputElem: the parse tree of the user's utterance that is to be matched against the Catalog;

\$catalogElem: the parse tree of the Catalog entry against which the user input is to be matched

Output: true if \$inputElem is derivable from \$catalogEntry, false otherwise

Static: \$labelTable, a table storing, for each variable in \$catalogElem, the subtree of \$inputElem that 'fills' the variable slot

```

1  if ($catalogElem and $inputElem are of different element types)
2    return false ;
3  else if ($catalogElem represents a variable element)
4    if (semantic type of $inputElem is not same type as, nor subtype of, semantic
        type of $catalogElem)
5      return false ;
6    else
7      $labelTable.put(variable label of $catalogElem, $inputElem);
8      return true;
9  else if (attributes of $inputElem and $catalogElem do not match exactly)
10   return false ;
11 else if ($catalogElem and $inputElem do not have the same number of children)
12   return false ;
13 for ($i from 1 to number of children)
14   if (goalTest($i-th child of $inputElem, $i-th child of $catalogElem) == false )
15     return false ;
16 return true;

```

Figure 6.9.: The goalTest algorithm.

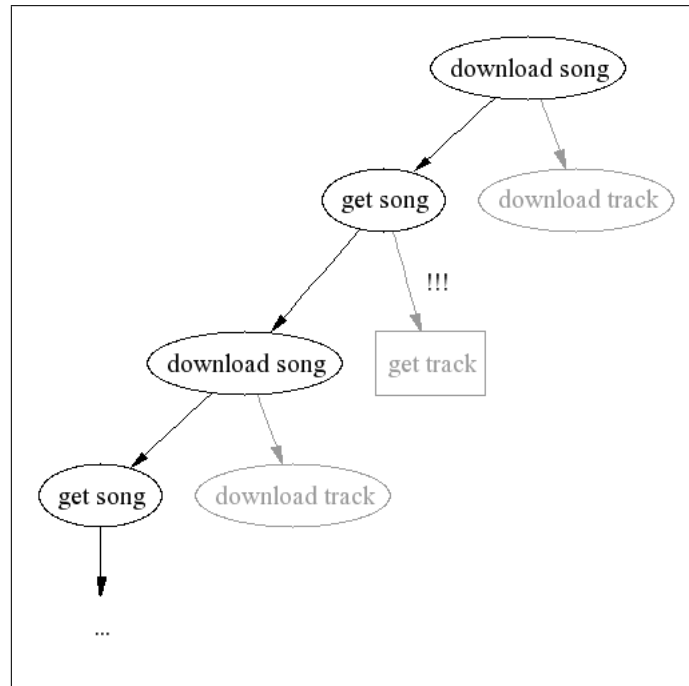


Figure 6.10.: A possible scenario if we use a tree-based instead of a graph-based search algorithm. The rule translating “download” to “get” is always immediately undone by applying the inverse rule. The rule (marked “!!!”) that would lead into a goal imperative (rectangle-shaped) can thus never be applied.

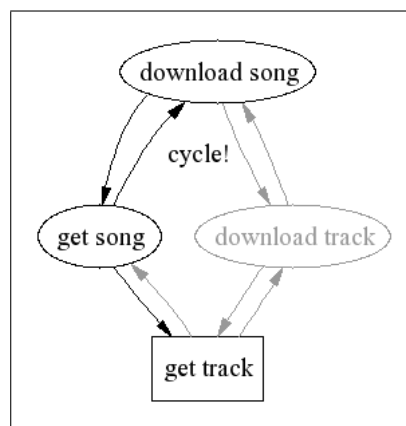


Figure 6.11.: The scenario of figure 6.10 when tackled with a graph-based algorithm. The cycle is detected and can consequently be avoided. The rectangle-shaped goal imperative is found.

Algorithm: `Matcher($scr)`

Input: `$scr`: a script, i.e., a list of imperatives

Output: `true` if the script can be matched with the Catalog, `false` otherwise

```

1  for (each $imp in $scr)
2    if ($imp.matched == true)
3      continue;
4    if ($imp.matched == false)
5      return false ;
6    if (goalTest($imp) == true)
7      $imp.matched = true;
8      $imp.matchingScript = script consisting of $imp only;
9      continue;
10   $overallSuccess = false ;
11   $bestQuality = negative infinity ;
12   $imp.onCurrentPath = true;
13   for (each $rule applicable to $imp)
14     $succScr = apply($rule, $imp);
15     if (any imperative $i in $succScr has $i.onCurrentPath == true)
16       continue;
17     $success = Matcher($succScr);
18     $overallSuccess = $overallSuccess || $success;
19     if ($success == true)
20       $quality = proximity of $succScr to the imperative with which the
21         algorithm was initially started;
22       if ($quality > $bestQuality)
23         $bestQuality = $quality;
24       $imp.matchingScript = concatenation of the matching scripts of all
25         imperatives of $succScr;
26   $imp.onCurrentPath = false;
27   if ($overallSuccess == false)
28     $imp.matched = false;
29     return false ;
30   else
31     $imp.matched = true;
32 return true;

```

Figure 6.12.: The Matcher algorithm.

Algorithm: `apply($rule, $imp)`

Input: `$rule`: a rule, i.e., a pair $(lhs \mapsto rhs)$, with lhs an imperative and rhs a script;

`$imp`: the imperative the rule is to be applied to

Output: the script that is the result of applying `$rule` to `$imp` one single time; `null` if `$rule` is not applicable to `$imp`

```

1  if ($rule.rhs consists of more than one imperative)
2    if (goalTest($imp, $rule.lhs))
3      $result = empty script;
4      for ($imp_1 in $rule.rhs)
5        $imp_2 = $imp_1 with variable labels resolved (as stored in the
           $labelTable filled by the call to goalTest);
6        append $imp_2 to $result;
7      return $result ;
8    else
9      return null ;
10 else
11   $imp_3 = applyRec($rule, $imp);
12   return script consisting of $imp_3 only;
```

Figure 6.15.: The `apply` algorithm for applying a rule to an imperative.

Algorithm: `applyRec($rule, $imp)`

Input: `$rule`: a rule, i.e., a pair $(lhs \mapsto rhs)$, with lhs an imperative and rhs a script (which, in this case, consists of one imperative only);

`$imp`: the imperative the rule is to be applied to

Output: the script that is the result of applying `$rule` to `$imp` one single time; `null` if `$rule` is not applicable to `$imp`

```

1  if (goalTest($imp, $rule.lhs))
2    return $rule.rhs with variable labels resolved (as stored in the $labelTable
           filled by the call to goalTest);
3  for (each $child of $imp)
4    $child_1 = applyRec($rule, $child);
5    if ($child_1 != null)
6      $imp_1 = $imp with $child replaced by $child_1;
7      return $imp_1;
8  return null;
```

Figure 6.16.: The recursive part `applyRec` of rule application.

6. *Natural-language processing*

7. Data mode and command mode

7.0.5. Overview

Data mode and command mode are the names of two input modes that the Left Automaton can be in. They are mutually exclusive. Command mode is the usual mode, in which the Left Automaton parses incoming strings as commands and tries to match them against its Catalog containing a list of all commands it knows.

When one of these commands needs additional data, the Left Automaton switches to data mode to gather that data, displaying a notice to the user. When the user exits data mode, the data is added to the imperative representing the command (section 6.2.2 described that imperative objects have a ‘raw data’ field) and then passed along the normal route.

7.0.6. Description

Imperatives come in two varieties:

- those that can be processed without further data and
- those that need additional data as an argument.

The modes that the Left Automaton can switch, then, are an indication of what it expects. That can either be a new command (if that is the case, then it is in ‘command mode’), or parameters for a command that needs additional data (then it is in ‘data mode’). Obviously, since data is always tied to a command, the ‘normal’ mode is the command mode.

Whenever the Left Automaton, while matching against the Catalog, recognizes a command for which additional data is necessary, it will automatically switch to the data mode.

Input modules are *not* aware of what the current mode is. All of these ‘technicalities’ are encapsulated in the Left Automaton. After the shift in modes occurred, the input modules continue to normally pass on user inputs to the Left Automaton—except now those will then not be matched against the Catalog of available commands, but rather be gathered as ‘raw’ data for the imperative that requires it. In conjunction with the mode shift, the

7. Data mode and command mode

Left Automaton sends out a notification message to the user, which will follow the normal routing path through the Middle Automaton, and subsequently end up with an apt output media, the choice of which reflects the user's preferences.

That notification message, it is important to note, fits into the normal mechanism of communicating data to the user. It is a 'normal' BSF containing a short message that data mode has been entered, along with instructions on how to exit it. Until the user inputs the special sequence (in our prototype, simply "end"), all input from all modules will be treated as 'raw data' for the last parsed command.

The mode shift also works in ensemble with command macros, i.e. a single user command which is decomposed into a list of commands by means of Rulebook application. These entries in the resulting script are also normal commands in themselves, and if they require data of their own, the normal mode shifts will occur.

7.0.7. Scenarios

Two scenarios will be shortly presented, namely

1. a simple one to illustrate the 'normal' case and
2. a more complicated one that illustrates the potential intricacies.

In the first scenario, the user, by means of the keyboard module, passes the phrase "turn off the following" to the Left Automaton. Since it can be matched directly with a single imperative, which also needs additional data (this is specified by the respective Catalog entry), the Left Automaton switches into data mode, asking the user through a Barrier-BSF to input the associated data. The user, telling the Barrier through the speech module "the light" and ending data mode by typing "end" on the keyboard module, complies. The Barrier, now satisfied, switches back to normal command mode, and sends the imperative along with a data tag "the light" to an application, which then turns off the light.

In the second scenario, the user again simply inputs a command, in this case "fetch me a pretzel". However, no single entry is registered for that command. Instead, it can be matched (using, among others, decomposition rules) against a whole barrage of imperatives, namely the script *langle*"drive downstairs", "buy \$x0 :phys-thing", "drive upstairs", "give \$x0 :phys-thing to user"*rangle*. While "\$x0 :phys-thing" can be substituted with pretzel without further ado, in this scenario each of these new imperatives that are now to be passed on needs further data.¹ As described in the section about the Rulebook, each imperative in a script is processed in turn, and as each of the four imperative becomes the

¹This is an assumption in the scenario.

top entry in the waiting queue, the Left Automaton needs to switch to data mode, ask the user for data, exit the data mode, and only then pass on the imperative.

Note that there is no robot that can fulfill that request anyways, but the outlined complexity persists nonetheless.

7.0.8. Technical description

In actually implementing these modes, some technical hurdles had to be overcome. In the end, it led to a rather sophisticated synchronization algorithm, which took a lot of development time, as described in the next two subsections.

For an illustration on how this did in fact cause problems, the second scenario described in the previous scenario section can be drawn upon. For each imperative in the script that was matched to the user input inside the Catalog, additional raw data might be required from the user. That means for *one* actual user input, the Barrier might have to switch to data mode multiple times, if that user input is decomposed into several commands in the matching process.

Synchronizations

Figure 7.1 describes how that conundrum was eventually solved and delivers the synchronization aspects of the algorithm that accepts new words from input modules in the Left Automaton (cf. figure 5.3). For some of the actual data processing (since figure 7.1 shows only the synchronization) cf. figure 5.3.

7.0.9. Development history, alternatives, and venues of improvement

Initially, the tentative solution to the difficulties with having scripts with *multiple* imperatives that all need data, was the version of the algorithm pictured in 7.2. Note that it, also, only depicts the synchronization part of that algorithm. Luckily, it was discarded for the more elegant solution from figure 5.3.

7. Data mode and command mode

Algorithm: `acceptGroupedWords($words)`

Input: `$words`: a set of words that have been determined to belong to one single user command by the Grouper

Output: none

Static: `$dataBuffer`: a buffer storing the words entered by the user in data mode

```
1 synchronized ($dataBuffer)
2   if ($dataMode == true)
3     if ($words == the command for quitting data mode)
4       $dataBuffer.notify();
5     else
6       append $words to $dataBuffer;
7     return;
8   else
9     $matchingScript = try to match $words with the Catalog;
10    if ($matchingScript != null)
11      for (each $imp in $matchingScript)
12        if ($imp is an imperative requiring data input by the user)
13          $dataMode = true;
14          $dataBuffer.wait();
15          store $dataBuffer in $imp;
16          clear $dataBuffer;
17          $dataMode = false;
18  if ($matchingScript != null)
19    process $matchingScript;
```

Figure 7.1.: Once again, the `acceptGroupedWords` algorithm. This time only with the synchronization mechanism in detail. Line 9 is short for lines 1–17 in the algorithm of figure 5.3; line 19 is short for lines 20–25 there.

Algorithm: `acceptGroupedWords($words)`

Input: `$words`: a set of words that have been determined to belong to one single user command by the Grouper

Output: none

mainBlockOccupied: a semaphore indicating whether any thread is currently in the critical section

Static: `$dataBuffer`: a buffer storing the words entered by the user in data mode

```
1 synchronized ($mainBlockOccupied)
2   if ($mainBlockOccupied == true)
3     $mainBlockOccupied.wait();
4   $mainBlockOccupied = true;
5
6   if ($dataMode == true)
7     if ($words == the command for quitting data mode)
8       $dataBuffer.notify();
9     else
10      append $words to $dataBuffer;
11      $mainBlockOccupied.notify();
12
13   else
14     $matchingScript = try to match $words with the Catalog;
15     if ($matchingScript != null)
16       for (each $imp in $matchingScript)
17         if ($imp is an imperative requiring data input by the user)
18           $dataMode = true;
19           $mainBlockOccupied.notify();
20           $dataBuffer.wait();
21           store $dataBuffer in $imp;
22           clear $dataBuffer;
23           $dataMode = false;
24
25     synchronized ($mainBlockOccupied)
26       $mainBlockOccupied = false;
27       $mainBlockOccupied.notify();
28
29     if ($matchingScript != null)
30       process $matchingScript;
```

Figure 7.2.: The *old* `acceptGroupedWords` algorithm's synchronization part. As can be seen by a simple comparison to the current algorithm, it was significantly more complex, and therefore prone to error.

7. *Data mode and command mode*

8. Module management

8.0.10. Overview

In addition to all that has been said by now, it is one of the Left Automaton's tasks to keep track of all the input and output modules known to the Barrier.

Module management is a *dynamic* process: both input and output modules need not be known to the Barrier in advance, they can be added and removed at runtime. This is a significant advantage and is actually the reason why modules are called modules: they must be entities of their own, detached from the core system, yet easily pluggable and unpluggable. The user, for example, might not want to be watched by the camera module at all times, rather activating it when she feels like using it to input commands.

Also, we do not want to restrict the user to any fixed means of communication (say, the keyboard for input and the screen for output): it shall be possible to develop different and/or better input 'preprocessors' at a later stage. This is why modules are not parts of the Barrier but rather add-ons that merely have to follow a simple communication protocol in order to be compatible.

8.0.11. Description

Modules as specialized applications

Modules and applications are quite different 'topologically': modules are placed on the far left, while applications, on the contrary, sit on the far right¹ (cf. figure 3.1).

Another difference is that applications can communicate with the Barrier in a bidirectional manner, whereas modules are essentially meant to operate into one direction only: input modules send strings *to the Barrier*, output modules receive BSFs² *from the Barrier*.

For some purposes, however, communication must be viable in the other direction as well:

¹Topologically, not politically.

² Remember that BSF is the data format used for the communication of the Barrier with the 'outside world' (for details, cf. chapter 12); all output arrives in the Barrier wrapped into BSFs.

8. Module management

- An input module, e.g., although normally just sending to the Barrier, must be able to *receive* a shutdown BSF from the Barrier. Otherwise it would have to run indefinitely once it has been started.
- An output module, although normally just receiving from the Barrier, must be able to *send* the registration and installation BSFs to the Barrier. Otherwise the Barrier could not know of their existence. (Remember that it is a requirement that modules can be added and removed dynamically.)

This is why we treat modules as specialized applications internally. They register just like regular applications (with the Right Automaton; details will be discussed in chapter 21); additionally, lists storing which modules are installed and which of them are currently running are stored within the Left Automaton. So when a module starts up it actually has to send out two registration BSFs: one in its function as an application and one in its function as an input or output module, respectively.

Starting modules

Another major difference between applications and modules is this: Whereas an application is automatically started if it is not yet running but is designated to process an imperative, this works differently with an output module when it is designated to process (i.e. display) a BSF containing application output: we assume that all output modules that can get BSFs from the Left Automaton are already running. We do not want modules to be automatically started because it shall be up to the user to decide which media she wants active; e.g. if she is just enjoying a healthy and relaxing cucumber mask, she most probably wants screen output to be disabled. So no BSFs are sent to the Left Automaton for delivery to output modules that are not running; the Middle Automaton knows they are not running and does not forward any BSFs to them.³ (This is different with applications: we want to hide the identities of applications from the user in a certain way. The identity of media for outputting data, however should obviously not be hidden.)

Given this, the user must be able to *explicitly* start both input and output modules. In the context of application management (section 21.0.35), we will present an elegant mechanism for that, using any of the input modules that are already running (e.g. by saying “start vision module”, which will be recorded by the already running speech input module). By treating modules as special applications, we will be able to exploit the same mechanism for them, too.

³The Middle Automaton also keeps track of which output modules are running: whenever an output module’s registration BSF ‘percolates’ through the Middle Automaton on its way from right to left, the Middle Automaton, which is in charge of threshold management (cf. chapter 18), sets the respective module’s thresholds to ‘active’ before passing the BSF on to the Left Automaton.

With the exceptions that have been listed in this section the mechanism for managing the module registry is very similar to the one for managing the application registry. So we refer the reader to chapter 21 for more details.

Initial thresholds

Chapter 18 will describe how the Barrier's choice of an output module for a given BSF type is changed over time to adapt to user preferences. In a nutshell, this is done by attaching 'thresholds' to each pair of BSF type and output module. Since the Barrier cannot know about an output module's nature (remember the requirement of dynamic extensibility), the modules can themselves provide a set of initial thresholds (by sending a dedicated BSF) that serve as a starting point for subsequent threshold adaptation. If they do not, standard values will be assumed.

Catalog broadcasting

It is definitely convenient for the user to have an idea about which commands are currently available. To be sure, for the Barrier concept in its purest form this would not be an indispensable requirement, since the ultimate goal is to enable a natural way of communicating with the computer—and a 'natural' servant certainly does not carry a billboard around her neck indicating all the jobs she can do. But this does not mean it would not be helpful! This is why the Barrier will always keep output modules up to date about which imperatives are currently accepted by the system. This is done by the Left Automaton: it publishes the Catalog's content (the imperatives realized in text, not as their XML logical forms) as a BSF

- in a broadcast to all output modules whenever the Catalog changes, i.e. when an imperative is removed or when a new one is added, and
- to a single output module that has just registered.

How output modules deal with this information is completely up to them; cf. the section about the screen module (11.1) for an example. Modules need not display the Catalog content at all; in our prototype this is the case for the speech output module: it would be very cumbersome to have the Catalog read out aloud every time it changes, since that media seems not to be suited for that task at all.⁴

⁴Note that many call center owners seem to think otherwise, having machines read aloud every possible action to the user on the phone.

Transferring output between output modules

Remember our above example of the user with the healthy cucumber mask. Imagine that, just as she is relaxing, an e-mail for her is being read out by the speech module, running:

“Dear customer, please find attached a picture of our newest manicure set.”

Then she would certainly want to interrupt the cucumber procedure and see that picture. But, alas!—The speech module cannot display pictures. To nonetheless cater to the lady’s wish the Barrier comes with a mechanism to transfer output from one module to another, by means of the special commands listed on page 174 in table 18.1: she would simply say “redirect output to screen”, and the most recent output, in this case the e-mail, is cleared from its current output module (i.e. the voice reading the message is stopped) and is transferred to the screen output module, where the user can look at the text and image content of the message.

In another scenario, if the user wants to fully revel in her beauty slumber, she might simply not want to be bothered by commercial e-mails. So, by saying “clear output”,⁵ the voice is simply stopped, without transferring the output to another module.

8.0.12. Technical description

Since in our implementation we regard modules as special kinds of applications, the classes for the input and the output module registries in the Left Automaton simply extend the respective classes used for the application registry in the Right Automaton (cf. chapter 21 for details).

⁵If she really means it, and *really* does not want to be bothered with such data again, she can also use the stronger version, which is “reject output”.

Part III.

Modules

9. Introduction

*Die Module spiel'n verrückt, Mensch ich bin total verliebt,
voll auf Liebe programmiert mit Gefühl.
Schalt mich ein und schalt mich aus, die Gefühle müssen raus.
Ganz egal, was dann passiert, ich brauch' Liebe.*

—German lyrics of the song *Das Modul—Computerliebe* (1995)

9.0.13. Overview

As alluded to in chapter 3, module is a term used within the Barrier to describe a *special application* (as defined in the context of the Barrier) *dealing with either input or output*. A module is in between the Barrier and the user (cf. figure 3.1: there, modules sit left of the Left Automaton).

They come in two varieties: input modules and output modules. While input modules convert user input into strings and forward them *to* the Barrier to undergo processing, output modules output data received *from* the Barrier—which in turn has generally received it from applications—, or portray available commands (in short, provide the user interface, graphical or other) to the user.

9.0.14. Technical description

Figure 9.1 contains a UML class diagram of both input and output modules, leaving out many details to portray the major relationships and dependencies. It also portrays connections to classes that themselves are only mentioned at this point without a proper explanation. Out of all our example modules, just one input and one output module are displayed.

For details on how a Barrier application works, refer to figure 24.1. For details on the

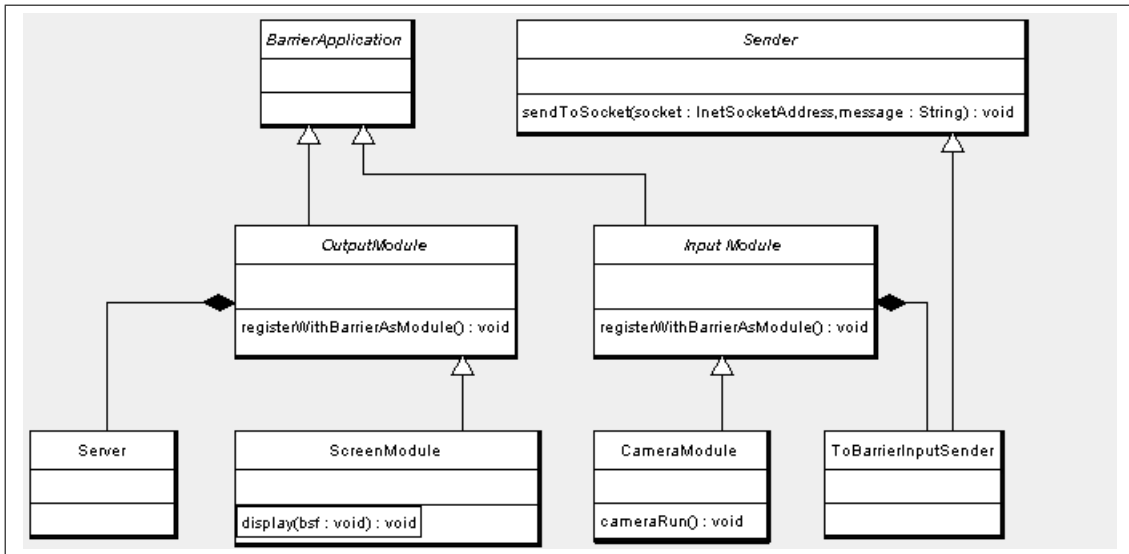


Figure 9.1.: UML class diagram of modules, only major relationships.

screen output module, refer to figure 11.2, or 10.4 for the vision input module. Finally, for information on the communication infrastructure (class `Server` etc.), refer to figure 27.1.

Just as is true for applications (see the note in section 24.0.45), using this class infrastructure when programming new modules/adapting existing solutions for use with the Barrier is *not* mandatory but rather an ‘offer’ to make development easier. Modules may be implemented in any way and programming language as long as they stick to the communication protocol described in chapter 27. So it is by no means compulsory to derive them from the abstract base classes we used for our example modules, as implied by the figure.

In the following chapters we shall describe the four input and the two output modules we implemented for the proof-of-concept version of the Barrier.

10. Example input modules

10.1. Speech input

10.1.1. Overview

Speech is certainly one of the most important, if not *the* single most important communication modality among humans. So it was indispensable to build an input module that understands spoken natural language. Speech recognition is definitely not yet a fully solved problem, and current recognizers lag behind human listeners considerably. Even though most companies tout accuracy levels of 98% or 99%, everyone who has ever tried their products will most likely have the feeling it is lower. This is because those high percentages hold only if strong conditions are met: e.g. the user’s voice characteristics must closely match those of the person who did the training,¹ the microphone must be properly adjusted, and the input signal must not be too noisy.

In a controlled environment it is, however, not too difficult to match the two latter requirements, and after a personalized training session the first criterion certainly holds, too. So if used by a person ‘known’ to the module, the professional software we use as a back-end—Dragon NaturallySpeaking (version 9), calling itself “the world’s best selling speech recognition technology”²—works without any major problems.

For other reasons, the speech input module is slated as the first input module to be substituted—not because it does not work, because it does (and quite well). Rather, an open-source version would be preferred, which would make the Barrier much easier to distribute and try out. Using this speech input module would require a license just to try it, and a system supporting and emphasizing multimodality is certainly not complete without a ‘standard’ speech input module.

¹That person, even if he did the training himself, might sound much different a few days after that training.

²cf. the company website <http://www.nuance.com/naturallyspeaking/>

10. Example input modules

10.1.2. Description

The way the speech input module works is simple: it leverages functions from a library that serves as a front-end to Dragon NaturallySpeaking, a popular commercial speech-recognition software package. It waits for spoken words to arrive over the microphone. Whenever this is the case, the most likely word sequence (i.e. the one with the highest level of confidence attributed by the recognizer) is passed to the Barrier's Left Automaton (for future grouping/merging/matching etc.) in the canonical way over a socket (cf. chapter 27).

When the module is started a dialog window for choosing one of the speakers that went through the training process is displayed. As is to be expected, recognition is notably better if a trained user enters words over the speech input module.

Note that, as NaturallySpeaking runs on Windows only, so does the speech input module.

10.1.3. Technical description

This module is implemented in Java. For accessing the functionality of NaturallySpeaking from the Java code, we use 'Cloudgarden TalkingJava',³ a full commercial implementation of the Java Speech API⁴ published by Sun Microsystems.

10.1.4. Development history, alternatives, and venues of improvement

There were several alternatives from which to choose, concerning the software to employ:

- The major alternative for Dragon NaturallySpeaking was *CMU Sphinx*,⁵ probably the best-known speech recognition framework. While NaturallySpeaking is commercial, ready-to-use software, Sphinx is an academic project with open source code meant for both application developers and researchers. Notwithstanding its quality, we concluded after experimenting with Sphinx that it is by far not as easy to use for development as NaturallySpeaking; this is why we opted for the latter in our prototype. As mentioned, due to licensing issues, Sphinx might, however, be the better solution for future versions.
- While using NaturallySpeaking, there were alternatives to TalkingJava, the library we used for interfacing to it. For example, there is a software development kit offered

³<http://www.cloudgarden.com/JSAPI/>

⁴<http://java.sun.com/products/java-media/speech/>

⁵<http://cmusphinx.sourceforge.net>

by Nuance, the makers of NaturallySpeaking, themselves.⁶ It is, however, designed for C++, C#, and Visual Basic, but as we preferred Java as the implementation language, we opted for TalkingJava.

- The same holds true for the Microsoft Speech API.⁷

10.2. Clumsyboard

Gothmog: *The Age of Men is over. The Time of the Orc has come.*

—J. R. R. Tolkien, *Lord of the Rings*

10.2.1. Overview

To show that really just any input module can be plugged into the Barrier we devised a rather unconventional one for slightly ‘awkward’ (in the sense of clumsy) or very casual users. We call it ‘Clumsyboard’.

The idea is to be able to enter simple patterns over the keyboard without having to pay attention which particular keys are actually hit. What counts is rather the direction and shape of the movement. Possible such patterns are:

- ‘east’ (finger slides right, e.g., keys `fghj`)
- ‘west’ (analogous)
- ‘south’ (finger slides down, e.g., keys `7zhn`)
- ‘north’ (analogous)
- ‘circle’ (finger draws a circle counter-clockwise, e.g., keys `7tfvbj7`)
- ‘wave’ (finger draws a sine wave, e.g., keys `xdrtzhbnk`)

These are also the patterns used in our implementation. Recognition of these patterns is very accurate. It must, however, be noted that, while Clumsyboard works flawlessly on a laptop computer’s keyboard, the keys of a regular PC keyboard are protruding too far to allow for smooth finger sliding.

⁶<http://www.nuance.com/naturallyspeaking/sdk/client/>

⁷<http://www.microsoft.com/speech/download/sdk51/>

10. Example input modules

The module waits for command-line input, feeds it into the estimation algorithm and passes the result (the most likely pattern, that is) to the Barrier. For example, if a counterclockwise circle pattern is recognized, the word “circle” is handed into the Barrier.

We will achieve the pattern recognition in three steps:

1. An off-line training process: The training program prompts the user to draw a certain pattern. This kind of trial is repeated several times for each pattern (40 times in our case), and so the program collects labeled training data.
2. From these training samples we estimate the parameters of the probability model (see next section).
3. When the program is run and input is entered by the user, we compute the pattern that makes this specific input most likely; i.e. a maximum-likelihood (ML) approach is followed (see section 10.2.2).

The first two steps are both off-line and the implementation comes ready-to-use with a pre-trained pattern model. (However, a fresh training process can still be conducted afterwards.)

10.2.2. Description

The model

The Bayes net We assumed the following model: the key pressed at time t , $K(t)$, depends only on the pattern P being drawn and on the key $K(t-1)$ that was pressed immediately before, at time $t-1$; i.e., we are using a simple Markov model. Also, the length L of the input key sequence depends on which pattern is being drawn. Figure 10.1 shows a graphical description of the Bayes net.⁸

P is the hidden variable that is to be guessed given the measured data $K(1), \dots, K(L)$, and L .

The model could be even simpler, by eliminating the variable L : This works most of the time, but difficulties arise: e.g. the key sequence `tfv` could itself mean ‘south’, but it also occurs in instances of the pattern ‘circle’: `ztfvbnjuzt`. Our estimation algorithm could then not know which is true since, due to the very simple Markov assumption, it considers only what has happened immediately before. In fact, the pattern ‘south’ makes the key sequence `tfv` less likely than the pattern ‘circle’ does; this is because in the training phase

⁸Note that there is in fact also an edge from P to $K(t-1)$; it is not visible, however, because this is the ‘subnet’ for node $K(t)$, not for $K(t-1)$. The actual, complete net is obtained by linking all subnets, i.e. by drawing arcs between neighboring K nodes from left to right and from P to each K node.

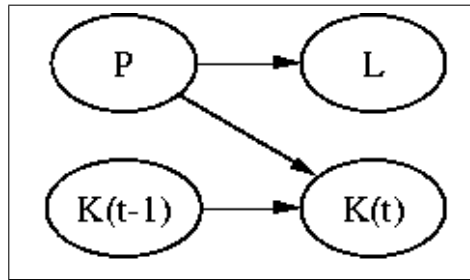


Figure 10.1.: The Bayes model used for Clumsyboard. P : the pattern drawn, L : the length of the input key sequence, $K(t)$: the key pressed at time t .

the ‘south’ down-stroke was made at a greater number of different horizontal coordinates on the keyboard than the circle (since the circle is broader than the ‘south’ stroke it can be made in fewer locations), which means it was done less often at `tfv` than the circle.

So the length provides the kind of global information we need: In the training the program learned that it is very unlikely that a circle consists of only three keys, but that a ‘south’ stroke is very likely to do so. In our maximum-likelihood estimation this will override the fact that the *sub*-sequence `tfv` is more likely for circles than for ‘south’ strokes.

Learning the net parameters The following probability distributions describe the whole model:

- The probability of a certain key being pressed, given the pattern being drawn and the key pressed before: $\Pr(K(t)|K(t-1), P)$.
- The probability of a key sequence having a certain length, given the pattern being drawn: $\Pr(L|P)$.

We estimate the parameters simply by counting in the example set collected during training:

$$\Pr(K(t) = x|K(t-1) = y, P = p) := \frac{\#(K(t) = x, K(t-1) = y, P = p)}{\#(K(t-1) = y, P = p)}$$

$$\Pr(L = \ell|P = p) := \frac{\#(L = \ell, P = p)}{\#(P = p)}$$

Applying the model for on-line estimation

Once the model has been computed off-line from the training samples we can run the actual ‘Clumsyboard’ program: the user enters a pattern by sliding his fingers over the keyboard. From the data entered we must now guess which one out of the known patterns

10. Example input modules

the user intended to input: We must guess which pattern p^* is most likely given the key strokes recorded:

$$\begin{aligned}
p^* &= \arg \max_p \Pr(P = p | K(1) = k_1, \dots, K(\ell) = k_\ell, L = \ell) \\
&= \arg \max_p \frac{\Pr(K(1) = k_1, \dots, K(\ell) = k_\ell, L = \ell | P = p) \Pr(P = p)}{\Pr(K(1) = k_1, \dots, K(\ell) = k_\ell, L = \ell)} \\
&= \arg \max_p \Pr(K(1) = k_1, \dots, K(\ell) = k_\ell, L = \ell | P = p) \\
&= \arg \max_p \Pr(K(1) = k_1, \dots, K(\ell) = k_\ell | P = p) \Pr(L = \ell | P = p) \\
&= \arg \max_p [\Pr(K(\ell) = k_\ell | K(1) = k_1, \dots, K(\ell - 1) = k_{\ell-1}, P = p) \cdot \\
&\quad \Pr(K(1) = k_1, \dots, K(\ell - 1) = k_{\ell-1} | P = p) \Pr(L = \ell | P = p)] \\
&= \arg \max_p [\Pr(K(\ell) = k_\ell | K(\ell - 1) = k_{\ell-1}, P = p) \cdot \\
&\quad \Pr(K(\ell - 1) = k_{\ell-1} | K(1) = k_1, \dots, K(\ell - 2) = k_{\ell-2}, P = p) \cdot \\
&\quad \Pr(K(1) = k_1, \dots, K(\ell - 2) = k_{\ell-2} | P = p) \Pr(L = \ell | P = p)] \\
&= \dots \\
&= \arg \max_p [\Pr(L = \ell | P = p) \Pr(K(1) = k_1 | P = p) \cdot \\
&\quad \prod_{t=2}^{\ell} \Pr(K(t) = k_t | K(t-1) = k_{t-1}, P = p)]
\end{aligned}$$

The last formula can be computed from the model that has been previously learned ($\Pr(K(1) = k_1 | P = p)$ can be computed by summing over the second key in the model). In the transformations we first used Bayes' rule; then the assumption that all patterns are *a priori* equally likely (this makes our approach a maximum-likelihood one, as opposed to a maximum-*a posteriori* one) and the fact that the *a priori* probability of the data measured does not depend on the pattern; then the fact that in our Bayes model L and the $K(t)$'s are independent given P ; we then pulled some terms into the conditional part; we made use of the Markov assumption; finally we iterated the last two steps.

There was another detail that had to be considered: Single entries of the *sampled* probability table $\Pr(K(t) | K(t-1), P)$ could be zero (because we never came across that specific situation in the few training examples) although the *actual* probability could be a small positive number. Then the product in the last formula would be 'flattened' to zero just because one single factor is (falsely) zero. We obviously do not want this because it destroys all the information we have. So when building the probability table we pretended that we had seen one single transition from each key to every other key (although we actually had not), i.e., we assumed that for every key there is a small probability to reach any other

```

Slide your clumsy finger over the keyboard. To quit type "quit".
> 6tgbv
I'm guessing you mean pattern 'south'.
Likelihoods:
  Under hypothesis 'south' the input has likelihood 4.871270300047839E-16
  Under hypothesis 'east' the input has likelihood 1.4764997423547915E-16
  Under hypothesis 'west' the input has likelihood 1.3529239574524164E-16
  Under hypothesis 'north' the input has likelihood 1.2270328738406306E-16
  Under hypothesis 'circle' the input has likelihood 1.0188608114486488E-16
  Under hypothesis 'wave' the input has likelihood 5.054440764335375E-17

Slide your clumsy finger over the keyboard. To quit type "quit".
>

```

Figure 10.2.: The Clumsyboard module's input prompt.

key from there next. We made the same assumption for the other table $\Pr(L|P)$. This is a common trick; e.g. it is described in [RN03, page 717].

10.2.3. Scenario

Figure 10.2 shows the Clumsyboard module's input prompt. The user has entered a top-down stroke (letters 6tgbv), which is correctly classified as pattern 'south', and the module is now waiting for the next input. For reasons of clarity, the output also includes the likelihoods the input has under *all* the different hypotheses that can be made.

10.2.4. Technical description

The implementation is straightforward given the mathematical considerations just delineated.

The name of the module in the code is `ClumsyModule`. The back-end that is actually doing the maximum-likelihood estimation is called `Clumsyboard`, the off-line model computation is done in the class `EstimateProbabilityTable`, pattern sampling is done by means of the `Sample` class, all three placed in the package `de.tum.in.clumsyboard`.

After estimation, the probability table is serialized to the file `probability-table.obj` in directory `$BARRIER_HOME/clumsy.config/`; it is computed from the patterns that the sampling class has stored in the same directory. The on-line estimation is done with the probability table that is deserialized at runtime.

10.3. Keyboard

10.3.1. Overview

While the Barrier’s declared goal is for the user to communicate with her computer system in a *natural way*, that does not preclude the Barrier from using the keyboard as a potential input module.

Quite the contrary, with certain phrases it has in fact become natural *not* to actually say them. That partially explains the difference between written English, especially in forums,⁹ and spoken English. Who, after all, would say “lol” to another fellow human?

It is important to note that the Barrier’s paradigm does not include users using the keyboard module in a shell-like manner (although it certainly could), learning hermetic command sequences. Instead, for certain words it might be much more practical to write them than to otherwise communicate them. This could be because

- the user is unable to pronounce the command correctly,¹⁰ or
- the user is unwilling to talk aloud in her current environment, or
- the user prefers using the keyboard module to ensure that rare words are correctly entered.

Another purpose would be for testing the Barrier. Turning off other input modules, by exclusively using the keyboard module, it can be ensured exactly what input the Barrier will have to work with. In stress testing, that can be invaluable. Otherwise, it always has to be checked whether the peculiarities of other input modules changed the Barrier’s behavior.

One can easily imagine that, out of all input modules, this is the simplest and thus most robust one.

10.3.2. Description

The keyboard input module is a simple application reading the input which is written into a window by the user. When the user presses the ‘return’ key, the accumulated text is passed to the keyboard module’s sender, which forwards it to the Barrier’s server.

Figure 10.3 depicts a screenshot of this module’s input prompt. The GUI is deliberately kept very simple to emphasize that its only task is to give the user feedback as to what

⁹cf. 1337sp3ak, <http://en.wikipedia.org/wiki/Leet>

¹⁰cf. “How much wood would a woodchuck chuck if a woodchuck could chuck wood?”

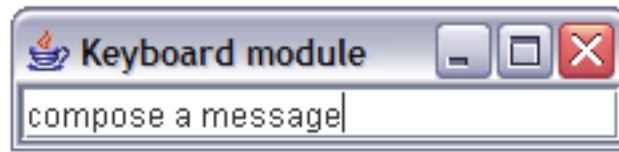


Figure 10.3.: A screenshot of the keyboard module’s input prompt.

she has written so far.

10.4. Vision

10.4.1. Overview

The vision input module serves as a prototype module to illustrate how visual signals can seamlessly be input and work in conjunction with the Barrier. Its vision recognition capabilities are rather simple, but enhancing them can be done without any further consideration of the Barrier itself, and is *purely* an image processing task. Existing image processing solutions can be adapted to the Barrier with minimal effort, as all to be done is to exchange the image functionality part of the prototype vision input module with arbitrary other (image processing) functionality.

The function of this specific vision input module is to discern whether the user turns her head to the left or to the right, signaling the word “left” or “right”, respectively. Because this thesis is not about facial recognition, the user is to wear a simple colored ‘hat’ to ease the image processing. Note that the algorithm can obviously be refined or substituted to do without such a color add-on.

10.4.2. Description

The functionality just described is realized through a finite state machine (FSM). While figure 10.4 shows the complete state transitions and states for a `CAMERA_LATENCY` of 2, the actual camera module uses a value of 4.

Processing starts in state 1. “Both” and “none” represent the camera seeing both colors in a frame or none, respectively, while “yellow” and “blue” signify the camera *just* seeing the namesake color in a frame. As usual, “ x/α ” above a state transition means that action α will be executed when that transition is triggered by the input symbol x .

The user in front of the camera, which in the prototype setup is simply a common webcam, wears a ‘hat’ or similar, which has a distinct color on each side of her face. When the face

10. Example input modules

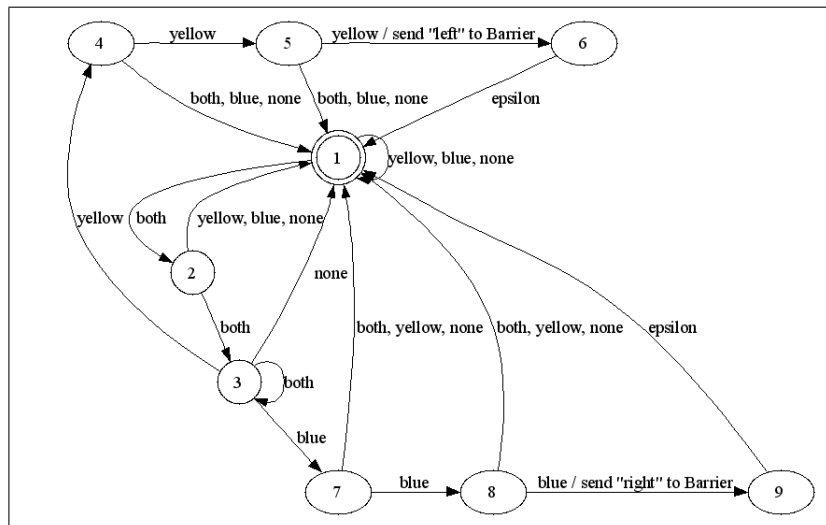


Figure 10.4.: Finite state machine describing the vision input module, with a `CAMERA_LATENCY` of 2.

is fully turned towards the camera, both colors are present (see figure 10.5 for an example scene). To avoid false alarms, the `CAMERA_LATENCY` describes how many such frames need to be seen in a row for the system to say with certainty that the user does in fact face the camera. Once these are over, the system will wait until only *one* of the two colors is seen. That can only happen when the user *turns her head*, thus signifying either the word “left” or the word “right”, depending on which color is seen. Again, to ensure that seeing one color was not merely a glitch, a sequence of length `CAMERA_LATENCY` is necessary. Once it did occur, the input module sends the appropriate word (“left”/“right”) to the Barrier’s Left Automaton. After that, a spontaneous ϵ transition takes the input module back into its starting state, waiting again for both colors to be seen for `CAMERA_LATENCY` cycles.

It should be noted that since the Left Automaton’s Grouper (cf. section 5.1) only groups words from different input modules into one phrase to be matched against the Catalog if they arrive from these modules *within a certain time frame*, it is *not* a problem if the user turns her head spontaneously *without* intending it to be a command (since the words “left” and “right” alone, not being imperatives, cannot be in the Catalog as such). Only if another input module *also* delivers a string to the Left Automaton *within that time frame* can a valid match with a command in the Catalog be found.

10.4.3. Technical description

In the Barrier prototype, the vision input module is called `CameraModule`.

A portion of the code for controlling the camera device and assorted tasks was taken from pre-existing classes, which were adapted and reused. Such reuse is indicated in the source

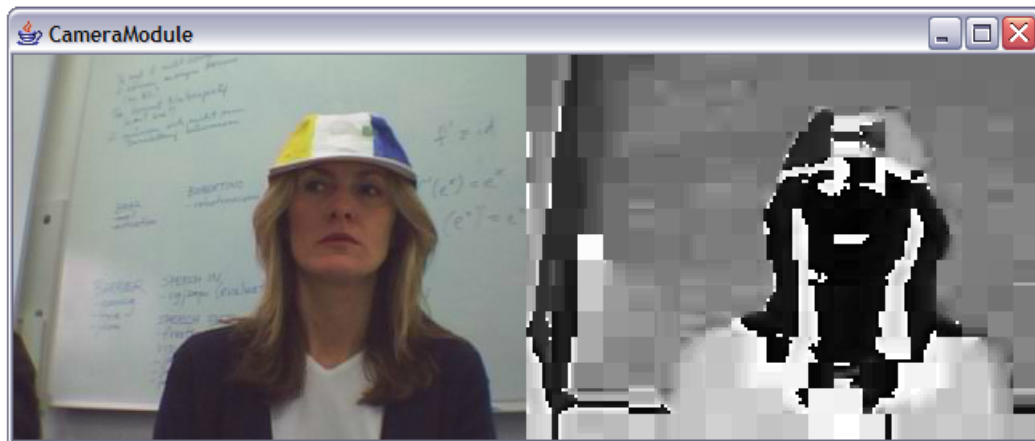


Figure 10.5.: A typical input scene for the camera module. The left-hand side is in RGB, the right-hand side in HSI coordinates (only hue channel displayed).

code were applicable. The algorithm itself, and the major part of its implementation is original code.

Given a picture, a color transformation from RGB to HSI (Hue, Saturation, Intensity) is done. In the HSI color space, lighting is not implicitly spread over three color channels—as in RGB—but rather explicitly stored as an ‘intensity’ coordinate. That leaves the color value residing mostly in the hue field, greatly facilitating the next steps, as they should not depend on the lighting in the room etc.

Then, a histogram of the picture is computed. More exactly, the histogram is concerned only with the hue color channel of the picture, which by then is in the HSI color space. This allows for a more robust processing that is far less dependent on fickle lighting conditions: while, with RGB, one would have to recalibrate the module before every use, this is not the case with HSI.

Lastly, that histogram is checked for a sufficient presence of one or both of the cue colors, which in the prototype are blue and yellow (as these are quite distinct in the color channel, not overlapping in any way). The result of these checks then serves as the new input symbol for the FSM, and can be any one of

{none, blue, yellow, both}.

Figure 10.5 shows a scene that will typically take place in front of this module’s camera, the user wearing the above mentioned hat. On the left, it is in unprocessed RGB coordinates, while the right-hand side contains the same scene in HSI color space, reduced to the hue channel.

10.4.4. Development history, alternatives, and venues of improvement

Using the ‘normal’ color space, i.e. RGB, was found to be too sensitive to changing lighting. Once the facial recognition software of Giorgio Panin,¹¹ also of the Technical University of Munich, is fully completed, it is intended to eventually substitute it for above the algorithm. Alternatively, the existing algorithm can be modified to do without two colors, by e.g. trying to track whether one eye is occluded (which happens when the head turns).

Ideally, once sign language recognition software becomes more readily available (and robust), a vision input module could be included to the Barrier’s ‘standard’ input modules, truly enabling certain classes of handicapped users (the mute above all) to interact with computer systems (at least with those programs that support the Barrier).

¹¹cf. <http://www6.informatik.tu-muenchen.de/~panin/>

11. Example output modules

11.1. Screen

11.1.1. Overview

The screen output module is an output module whose task it is to visualize data on a computer screen, making it the Barrier's GUI (or, if there are several such modules, part of it). These data comprise application output, as passed into the Barrier and from there to this module as BSFs, and the Barrier's current Catalog. Why the latter makes sense was described in the section about 'Catalog broadcasting' in chapter 8.

11.1.2. Description

More precisely, the screen output module's role is two-fold:

1. It has to display the currently available commands (i.e. the Catalog).
2. It has to display the BSFs it receives from the Barrier (all of which are of types it registered for).

In the screenshot of figure 11.1 one can see how these two jobs are done in our example implementation: the top part contains the set of all commands that can currently be processed by the computing system, while application output is shown at the bottom. In this case the output originates from the Bobertino application controlling a Robertino robot (cf. section 25.1) and represents a JPEG image taken by the robot's webcam. The user command that resulted in this output has been "take picture".

Concerning the first task, the screen module receives a broadcast (as described in section 8.0.11) containing the Catalog list reduced to string form (not in the verbose XML format Catalog entries have within the Barrier), whenever the Catalog changes. This is also done when the module registers itself (i.e. when it is started).

11. Example output modules

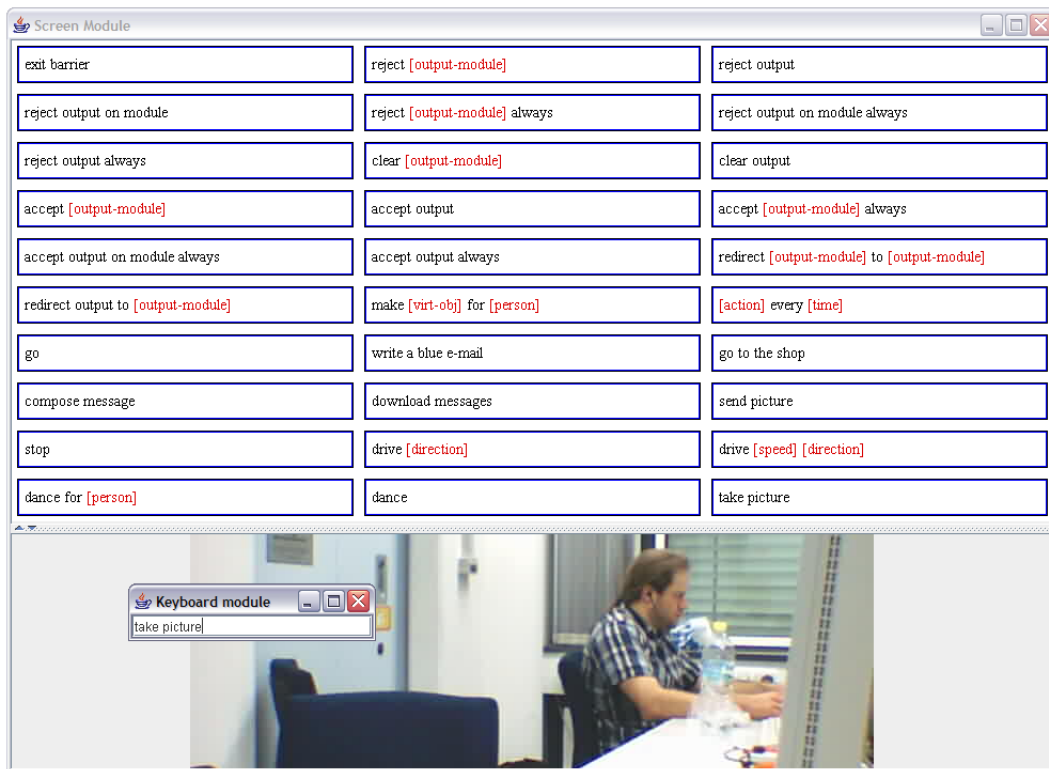


Figure 11.1.: A screenshot of the screen output module. The output displayed is a BSF encapsulating a JPEG image.

Concerning the second task, the module must be able to display those BSFs that it told the Barrier it can cope with.¹ Since it is a screen output module, these will for example include

- BSFs containing e-mails,
- picture BSFs, and
- most other text BSFs,

but *not* include

- ‘sound’ BSFs nor
- audio-book BSFs, etc.

Note that most of these BSFs are not yet specified, but can easily be supplied once there are actually applications that make use of them.

11.1.3. Technical description

The screen output module uses a `JSplitPane` object as the main plane, in which objects of two classes, of type `CommandsJPanel` and of type `BSFJPanel` are embedded. Both of these are extended from `JPanel`. The main task of the object `commandsJPanel` is to display the Catalog’s commands that are sent out by the Left Automaton as described. Conversely, the main task of the object `bsfJPanel` is to cope with the implicit hierarchy inside a BSF, and display its contents. This prototype output module does not differentiate between different BSFs, but reads in the BSF data part and displays both the data tags and the content, while retaining the original hierarchical ordering. JPEG images are an exception to this: they are decoded and displayed properly, like in the screenshot.

Figure 11.2 depicts the screen output module’s class diagram, focusing on the functional part only, as in contrast to the parts connecting to the Barrier etc.

11.1.4. Development history, alternatives, and venues of improvement

A screen module such as this will eventually be able to basically group commands semantically, offer different drawing modes etc., and offer everything that a nice looking GUI window can do. Extending the screen module can be done fully without considering the Barrier, and still that new module’s benefits will apply to all applications implicitly.

¹‘Told the Barrier’ meaning that it has indicated appropriate initial threshold values.

11. Example output modules

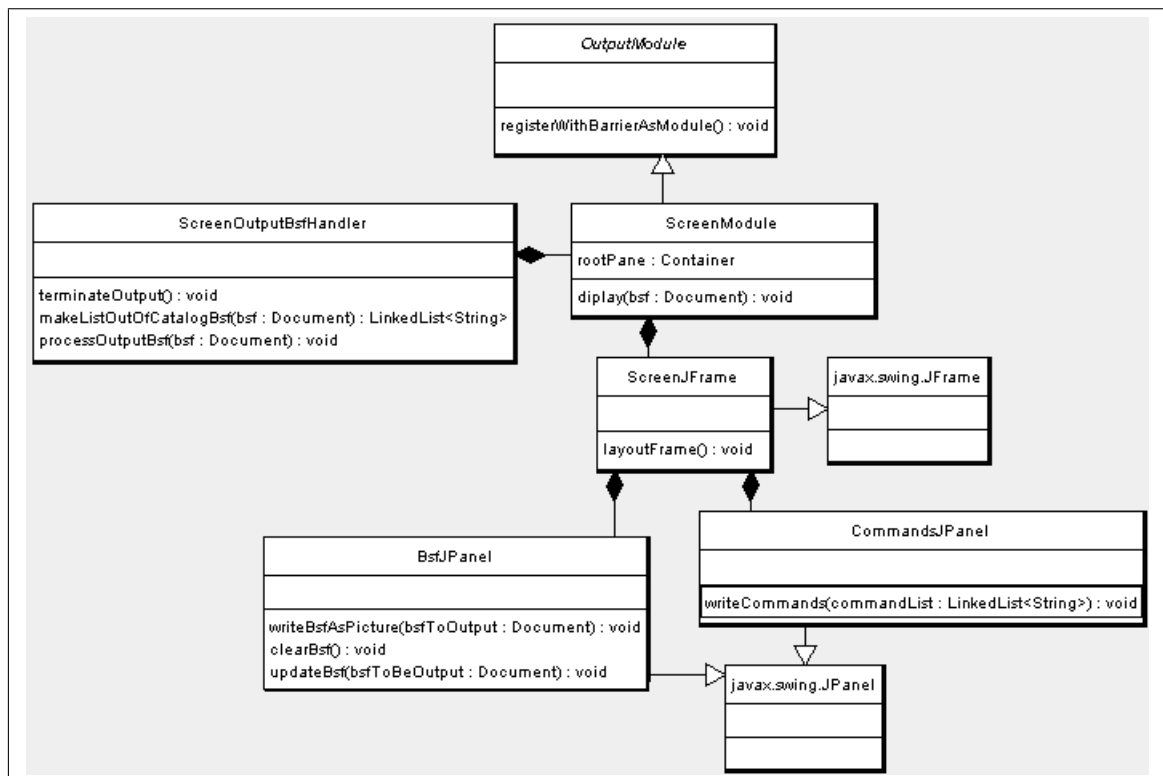


Figure 11.2.: UML class diagram of the functional parts of the screen output module; only example functions portrayed.

Since this is a prototype, enough room for improvement is left. It should be kept in mind that for the Barrier to be attractive in ‘real usage’, eventually a slew of output modules will be needed. The user will then have the choice between several screen output modules, each looking a bit different, and coping differently with the sorting of commands etc. This module is but a simple example of an output module in order to enable the Barrier to function (since it needs a way of outputting data).

A recently added addition, which was incorporated into the latest prototype build, is that picture BSFs, which contain the picture in binary form, will be specially recognized and have their data appear in graphical form (using the `sun.misc.BASE64Decoder` package).

To be clear about that point, ‘normal’ output modules will be expected to cope with as many individual BSF types as possible. That means not only displaying their contents in a generic way (as this prototype module does), but rather having a switch for many individual BSFs, that are then displayed in a way distinctly suited to their semantic meaning. For example, e-mail BSFs would be displayed in the standard e-mail client way. Only if no special handling of a BSF type was present (which would be no big drawback) would the module be forced to resort to the standard handling.

Customizing an output module to display different BSF types differently does *not* add to the complexity of the Barrier, and is in fact not even related to the Barrier itself at all. Instead, that customization is about visualizing content of a fixed type as graphically sound as possible, a task perfectly suited for typical UI programmers. Not to be forgotten, the module still has to *register* for all types it can display, and it is then its responsibility that either its specialized or the generic way of outputting are up to that task. Remember that output modules are used as the UI of all applications inside, which themselves do not have to worry about a GUI in the least.

11.2. Speech output

11.2.1. Overview

The speech output module’s purpose is speech synthesis for certain kinds of BSF data. It is not intended for displaying all available commands currently in the Catalog (as, in the prototype case, done by the screen output module), but merely for reading aloud those BSF types that describe mostly textual contents, such as e-mail BSFs.

11. Example output modules

11.2.2. Description

Incoming BSFs are, as described previously, in the XML format. To be able to read them, they must first be converted into a string (which is readable using the speech synthesizer engine described in 11.2.3). In addition, some semblance of the original hierarchy should be preserved. To achieve that, a regular expression is applied to the BSF data, which not only gets rid of the tag signs, but also includes pauses when hierarchical levels are changed; e.g. the tag

```
<receiverName>Edsgar Dijkstra</receiverName>
```

is transformed into

```
receiver Name:  Edsgar Dijkstra.
```

It must be possible to interrupt speech synthesis when the user wants the output to be stopped or transferred to another module (cf. the example in section 8.0.11). Therefore the module not only features a method for reading out BSFs but also a function `stopSpeaking`.

11.2.3. Technical description

The speech output module's functionality is implemented through FreeTTS,² an acronym for 'free text to speech'. It is itself implemented in Java, and based on another speech synthesis engine called Flite,³ a project at Carnegie Mellon University.

In the implementation, attention had to be paid that no two BSFs be read out at the same time. This is made sure by synchronizing the reading tasks over a common lock object. They are such queued.

11.2.4. Development history, alternatives, and venues of improvement

As is the case with the screen output module (section 11.1.4), the way a BSF is read out may depend on the specific BSF type in future versions of this module. Again, such changes would not concern the Barrier but *only* this separate module.

²available at <http://freetts.sourceforge.net/>

³available at <http://www.speech.cs.cmu.edu/flite/>

Part IV.

BSF

12. Introduction to BSFs

12.0.5. Overview

The term *Barrier Structured Format*, or short BSF, is used in two ways:

- It describes the *standard* that a file has to conform to in order to be a BSF file.
- It is shorthand for ‘BSF file’. Throughout this paper, this is the dominant usage of the term ‘BSF’. When we say, for example, “a BSF is received by the Right Automaton”, what is meant is that a file which is valid in the Barrier Structured Format is received.

A BSF can be subdivided in two parts:¹

1. A *header*, which is comprised of fields that identify the BSF and contains the properties necessary for handling it. For instance, with a picture BSF, this will include the type with which the BSF can be distinguished from other types, a time stamp etc.
2. A *data part*, which is how different types of BSFs differ from each other. With a picture BSF, this will probably include the picture data, along with information on its format.

BSFs are *the* major vehicle used to convey and transport information both inside the Barrier and to and fro. For information on the other vehicle for transporting data within the Barrier (direction ‘left to right’), confer the chapter about imperatives. BSFs are the *only* means of communication between the Barrier and applications.

Header

The header of a BSF contains both mandatory and optional fields. To give an impression of how a header is formed, figure 12.1 depicts an example.

¹Unlike Gaul.

12. Introduction to BSFs

```

1 <bsfHeader>
2   <bsfId>34254</bsfId>
3   <isRequestResponse>>false</isRequestResponse>
4   <timeStamp/>
5   <outputModule/>
6   <priority>8</priority>
7
8   <generatingApplication> de.tum.in.barrierapps.bmail </generatingApplication>
9   <bsfType> 9 </bsfType>
10  <tempId/>
11  <isBsfPersistent> false </isBsfPersistent>
12  <isBarrierBsf> true </isBarrierBsf>
13  <isCompositeBsf> false </isCompositeBsf>
14 </bsfHeader>

```

Figure 12.1.: Example of a typical BSF header.

Table 12.1.: Mandatory and optional parts of the BSF header. For each of these, information on where it is set, whether it is optional when entering the Barrier, and a short explanation of their role.

Field	Set by	Optional ²	Meaning
bsfId	Right Automaton	yes	identifies this specific data (not type)
isRequest-Response	Right Automaton	yes	true if the BSF answers a user request
timeStamp	Right Automaton	yes	entry stamp telling the BSF's 'age'
outputModule	Middle Automaton	yes	destination output module, considered best suited for this BSF type
priority	Application	yes	the priority the generating application considers this BSF to have
generating-Application	Application	no	the name of the application that generated this BSF
bsfType	Application	no	arguably the most important field, contains the type number of the BSF
tempId	Application	yes	repeats the request id if this BSF answers the corresponding request
<i>continued on next page</i>			

²Whether that field of a BSF has to be set to a value when it enters the Barrier (Right Automaton).

Table 12.1 *continued from previous page*

Field	Set by	Optional	Meaning
isBsfPersistent	Application	yes	if <code>true</code> this BSF may be saved inside the Barrier (data remains valid)
isBarrierBsf	Application	yes	indicates whether this is a Barrier-BSF (cf. chapter 12.1)
isCompositeBsf	Middle Automaton / Application	yes	for further information, see chapter 16
supertype	Application	no	if the BSF type is derived from another this field contains the type of the parent

Note that in table 12.1 many fields are optional when entering the Barrier (in this case via the Right Automaton), for the simple reason that if not set, a standard value will be assumed. For example, if the application did not give the BSF a priority value, which is an assessment of how urgent the application considers its data to be, a standard value will be assigned by the Barrier.

The ‘supertype’ tag enables a simple version of inheritance. It is, however, of importance solely for the threshold considerations, changing the switching against which the BSF is checked in order to determine whether it can be passed. This is described in more detail in subsection 17.0.20 on page 170.

Data

Figure 12.2 contains an example of what the data section of a BSF might look like. In fact, it is an actual data section of a BSF type used by the Barrier for single e-mails.

The structure of the BSF data of each BSF type is entirely up to the application designing it. However, when registering a new BSF type, the Barrier must be given a correct definition of that structure. Only then is the new BSF added to the BSF canon.

Raw data, e.g. binary data, is embedded directly into the BSF designed to carry it. Since the definitions used in the prototype (see the ‘technical description’ section) are quite powerful, applications are encouraged to define the content of each tag as narrowly as possible. For example, a BSF type carrying pictures might have an `imageData` tag in its BSF data. The content of that tag might ideally be restricted to only allow correctly formed images, instead of allowing e.g. all strings (which would be too lenient). This manner of designing new BSFs can preclude all kinds of errors by allowing the Barrier’s initial check when BSFs arrive (section 19.0.27) to check as thoroughly as possible.

12. Introduction to BSFs

```
1 <bsfData>
2   <email>
3     <senderName> Mrs. Money Penny </senderName>
4     <senderEmailAddress>
5       moneypenny@yourbankname.com
6     </senderEmailAddress>
7     <receiverName> CEO Barrier, Inc. </receiverName>
8     <receiverEmailAddress> aye@bee.see </receiverEmailAddress>
9     <subject> Congratulations! </subject>
10    <body>
11      The pre-approved loan has been deposited on your account,
12      see the attached confirmation.
13    </body>
14    <attachments>
15      <attachment>
16        <fileName> accountReport.jpg </fileName>
17        <fileType> exe </fileType>
18        <attachedData>
19          TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IG
20          IHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbmltYW
21          dGhlIG1pbmQsIHRoYXQgYnkgYSBwZXJzZXZlcmFuY2Ugb2
22          dWVkaGFuZCBpbmRlZmF0aWdhYmxiIGdlbmVyaXRpb24gb2
23          ZSBzaG9ydCB2ZWwhbWVuY2Ugb2YgYW55IGNhcm5hbCBwb=
24        </attachedData>
25      </attachment>
26    </attachments>
27  </email>
28 </bsfData>
```

Figure 12.2.: Example of the data part of a BSF. This is a possible data section of the actual ‘single e-mail’ BSF type.

In the example, the binary image data is Base-64 encoded, a standard for converting binary data to strings.

12.0.6. Scenarios

Here is an example list of tasks for which applications might generate a BSF and send it to the Barrier:

- A ‘container’ for transmitting a HTML-formatted website to the user.
- A message containing the application’s current capacity to the Barrier, to be used in the arbitration of requests (can be considered the ‘offer’ that an application makes).
- A message containing a request (e.g. for a picture) meant for *another* application (which is to execute it), which is sent to the Barrier to be distributed to the appropriate program.

An example list of tasks for which the Barrier might generate a BSF and send it to an application and/or a module:

- To tell the screen module to cancel its current output.
- To tell an application the id of an imperative it just registered.

Tasks for which *no* BSF would be used:

- A multi-threaded application communicating between its own threads.
- Communication with other applications that do *not* use the Barrier.

Note that some of these BSFs might technically be so-called Barrier-BSFs (see section 12.1).

12.0.7. Technical description

The BSF is implemented fully in the Extended Markup Language (XML). This guarantees both a powerful and widely spread basis and easy customization and adaptability. Also, Java comes with large and convenient packages for parsing and manipulating XML documents.

Each BSF type is defined in XML Schema.³ The XSD (XML Schema Definition) files containing these definitions can be found distributed along with the Barrier prototype.

³<http://www.w3.org/XML/Schema>

12. Introduction to BSFs

All these types share the definition of the BSF header, whose definition stays constant in over all the different BSF types.

After the decision of using XML to define the Barrier data type was made, there still was the question of how to exactly specify the format. The alternative to using XML Schema would have been using DTDs, short for Document Type Definition. That was the historically original method of specifying XML documents. Since we aimed, however, to create not only a few documents, but a real data type of its own, we preferred the most powerful tool to do so—which, in the context of XML, is certainly XML Schema.

In contrast to DTD, it is itself a valid XML Schema, and as such XML Schemas can be defined itself by using XML Schema. Not only does it offer support for namespaces (allowing Barrier definitions to be ‘unique’, and as such easier reusable), but since it also knows a greater variety of types by far, plus the ability to define arbitrary types of one’s own, the content of tags can be much more narrowly defined than it would otherwise be possible using DTDs (cf. the previous example about picture BSFs). Finally, after being recommended by the W3C, XML Schema is poised to remain the state-of-the-art method for defining documents for some time to come.

12.0.8. Development history, alternatives, and venues of improvement

Creating a fully proprietary format (without using preexisting formats as tools) could have vastly complicated matters. It would both have introduced many potential and unpredictable mistakes for which, in the time frame available, one would have been hard pressed to compensate, and curtailed the Barrier’s design philosophy of incorporating the newest technologies available. Using XML is a step in ensuring that for the foreseeable future the Barrier will remain state-of-the-art, if only the technologies incorporated.

12.1. Barrier-BSF

12.1.1. Overview

Barrier-BSFs are a special variety (i.e. a subset) of BSFs, namely those that are not meant for a user or another application, but rather for the Barrier itself.

12.1.2. Description

In table 12.2, all currently specified Barrier-BSF types are listed, with the recipient responsible for them, their type number, their name, whether the Barrier prototype has them implemented, and an explanation of their role. With the recipient, MOD denotes ‘module’, LA, MA, and RA the different Automata and APP ‘application’.

Table 12.2.: Exhaustive list of all Barrier-BSFs, i.e. all BSFs used by applications to communicate with the Barrier itself (in contrast to going through the Barrier to communicate with the user or another application).

Recipient	type	name	\exists XSD	role
LA	4	outputModuleReady	✓	for an output module to tell the Barrier it is ready
MOD	5	catalog	✓	for the Barrier to tell the output modules its current Catalog as a list of strings
LA	6	installInputModule	✓	for an input module to register itself
LA	7	installOutputModule		for an output module to register itself and to tell the Barrier its types and its standard thresholds
RA	8	uninstallApplication		for an application to uninstall itself, implying deregistration for all imperatives
RA	9	unregisterImperative	✓	for an application to unregister a specific imperative
RA	10	imperativeForApplication	✓	communicate among applications (contains imperative, which is forwarded by the Barrier)
RA	11	applicationReady	✓	for an application to tell the Barrier it has started and is ready for processing
RA	12	applicationCapacity	✓	for an application to tell the Barrier its capacity, scale 0–10 (float)

continued on next page

Table 12.2 <i>continued from previous page</i>				
Recipient	type	name	\exists XSD	role
MA	13	registerNewUnificationRule		for an application to register new rules for the BSF unification algorithm (cf. 16), might include XSD
LA/MA	14	inputModuleReady	✓	for an input module to tell the Barrier it has started
MA	15	moduleRefuseBsf		for a module to return a BSF to the Barrier for redistribution, e.g. when a module is busy
MA	16	registerNewBsfTyp		for an application to register a new BSF type with the Barrier, <i>must</i> include an XSD
APP	17	imperativeBsf	✓	for the Barrier to transmit an imperative to an application
RA	18	installApplication	✓	for an application to tell the Barrier it is now installed
LA/RA	19	registerForImperative	✓	for an application to register a (possibly new) imperative with the Barrier
(...)				
MOD	24	userInputError	✓	sent to the user if her input could not be parsed or matched by the LA
MOD	25	barrierError	✓	sent to the user if her input could not be parsed or matched by the LA
MOD	26	clearOutput	✓	for the Barrier to tell an output module it has to clear its current output
(...)				
RA	30	applicationShutdown	✓	for an application to tell the Barrier it is shutting down and not running any longer

continued on next page

Table 12.2 <i>continued from previous page</i>				
Recipient	type	name	\exists XSD	role
LA	31	inputModuleShutdown	✓	for an input module to tell the Barrier it is from then on not capable of parsing user input
LA	32	outputModuleShutdown	✓	for an output module to tell the Barrier it is from then on not capable of outputting BSFs
LA	33	registerNewRulebookRule		for an application to register new Rulebook rules for matching
APP	34	catalogIdForImperative	✓	the Barrier’s reply to an application after the application has registered for a new imperative

At this point, the notion of Barrier-BSFs has to be slightly expanded. As can be seen from table 12.2, not all Barrier-BSFs are entering the Barrier from applications and not all are destined for the Barrier only. Rather, the Barrier *itself* can also generate Barrier-BSFs that can be directed at both normal applications and modules. An example of that would be BSF type 24, containing a `userInputError` message, which can then be output by a module.

In this respect, these Barrier-BSFs share the name solely because they have been generated by the Barrier itself, and are considered to be of a “standard type” which each application or input module should be capable of processing. They are the ‘standard messaging system’ the Barrier uses to communicate its own intent to its peripherals.

12.1.3. Scenario

Here is an example scenario showing how BSFs are employed:

The Barrier being normally running, a new output module (e.g. speech) is started. This output module now has to communicate to the Barrier itself that it is ready for outputting application data it receives from to the Left Automaton. To do this, it creates a BSF of the appropriate BSF type, which is (cf. table 12.2) numbered 4.

Without confusion, one has to remember (as described in chapter 11 about output modules) that output modules are on the *left*—i.e. the environment’s side—*only* in their capacity as *modules*, but for other purposes also connected to the Right Automaton, just as any other application is. As the Barrier-BSF is certainly no user input (which could

12. Introduction to BSFs

enter from ‘left’), the BSF that the module created is now passed to the Right Automaton. Now having entered the Barrier, the Right Automaton checks whether it needs to concern itself with that specific BSF. It registers the module in its function as an application and passes the BSF on to the Middle Automaton (bypassing the processing that the Right Automaton normally does on BSFs representing application output!).

The Middle Automaton, then, does the same checks the Right Automaton has done, and upon finding that the BSF is a Barrier-BSF, bypasses its normal proceedings (thresholds, BSF storage, etc.), but instead handles the Barrier-BSF specially. All thresholds associated with the newly activated output module are set to active. Having done so, the BSF is given to the Left Automaton.

The Left Automaton, being the last one in the chain, proceeds just as its two counterparts, except that the handling differs. Here, it consists of the new output module being registered within the Left Automaton, and, most importantly, the current entries in the Catalog being sent to the output module. For elaboration why that is such a crucial step, cf. chapter 29 about automatic GUI generation on page 231.

At that point, all parts of the Barrier potentially responsible have had the chance to evaluate the Barrier-BSF, and consequently it can then be considered completely handled. It is therefore discarded by the Left Automaton.

12.1.4. Technical description

The Barrier can recognize Barrier-BSFs through the flag `isBarrierBsf`, whose values are Boolean, having the value `true`. It is an optional field in each BSF’s `bsfHeader`, however; if it not set it is assumed to be set to `false` implicitly. This is because Barrier-BSFs are considered to be not the norm but rather the special case.

As can be inferred from table 12.2, even those Barrier-BSFs processed solely by the Barrier can be meant for each of its three major parts, that is each of the three Automata. Hence, when a Barrier-BSF is received by the Right Automaton, it will mirror the way that every BSF traverses, namely being passed subsequently to each Automaton further left. In the Automata themselves, however, the Barrier-BSFs are processed specially. The scheme for that functions as follows:

- An Automaton receives a BSF from its right adjacent neighbor (in case of the Right Automaton, an application), then,
- if the BSF is a Barrier-BSF it will be given to a special routine `processBarrierBsfFromRight`, otherwise the *normal* processing takes place.

- The routine mentioned checks whether, given the type of the Barrier-BSF, it has any processing to do, after which,
- finally, the Barrier-BSF is passed on—just as normal BSFs would be—to the next Automaton on the left. As for the Left Automaton, which does not have any neighbor to its left, the chain terminates and the Barrier-BSF is considered to be completely processed (and therefore deleted).

Evidently, Barrier-BSF traffic in between the Automata is handled just like normal BSF traffic, using the very same functions to enter each Automaton. This is valid, as indeed Barrier-BSFs *are* BSFs as well. The pragmatics behind this reasoning is mainly to keep each Automaton's API as simple and intuitive as possible.

12.1.5. Development history, alternatives, and venues of improvement

To prepare the Barrier for more widespread deployment, additional guards against at least the most obvious exploits should be put in place. In this context that entails checking BSFs not only for their flag (and if that flag is indicating a Barrier-BSF, processing it as such), but also checking whether the type of the BSF, given in the `bsfType` field, is in fact among the list of Barrier-BSF types. This would bar applications from tagging BSFs wrongly as Barrier-BSFs, as the Barrier would double check that information against its information, instead of assuming by default that the flags are correctly set.

This might be of particular interest given that Barrier-BSFs are actively *controlling* the Barrier in many respects (cf. table 12.2 for details of these), and while the user is trusted in the current setup (where basically all users are considered administrators), applications should not be. Note that in a smaller scope, some similar reservations apply to Barrier imperatives.

12. *Introduction to BSFs*

Part V.

Middle Automaton

13. Introduction

“Du lebloses, verdammtes Automat!”

—E. T. A. Hoffmann, *Der Sandmann*

13.0.6. Overview

The Middle Automaton functions as the centerpiece of the Barrier, i.e. it holds most of the permanent data structures. Therefore, it can be considered to be a control unit that directs and filters traffic between the two Automata it connects, which are the Left and the Right Automaton.

According to user preferences (as thresholds), which are also encapsulated in the Middle Automaton, traffic can either be passed through, blocked and discarded, or blocked and stored. If traffic is passed through, it is also among the Middle Automaton’s functions to broker in what fashion it will be output, again according to the user’s preferences and the available output modules.

Conversely, when the Middle Automaton processes a command, which can be both from a user or from an application, it decides whether it can answer that request from its store of previously blocked BSFs, or whether that command needs to be forwarded to the Right Automaton and eventually to an application.

13.0.7. Description

Several ‘data paths’ lead through the Middle Automaton, and in order to explain its structure adequately, it is best to follow these. The two data streams, both significant, transfer

1. commands from left to right, i.e. in the direction from the user to the applications, and

13. Introduction

2. application output from right to left, i.e. in the direction from the applications to the user.

This, while slightly being an oversimplification, should be kept in mind for the descriptions to follow.

When commands (imperatives) enter through the API that the Middle Automaton provides to the Left Automaton, the first crossroads in the path is the decision whether that command is directed at the Barrier itself, and therefore has to be processed by the Middle Automaton, or is directed at an application. In the first case, the Middle Automaton will process the command according to the description in the section about Barrier imperatives (6.2.2), then pass it on to the Right Automaton. In the latter case, the Middle Automaton again has to decide whether it is actually necessary for that program to compute the command, or whether the command can be satisfiably answered by a BSF which is already in storage but has previously been blocked by the Middle Automaton. (For more information concerning BSF storage, also confer the data path in the other direction.)

In order for a command to be answerable by the Middle Automaton without any application being involved, two criteria must be met. First, the Middle Automaton must have in storage a BSF whose type has been recorded as a previous response to a command of the same kind. This command history, or rather request history is part of the request management. Second, that type-correlated BSF must have come from an application that is registered for that imperative.

With these requirements satisfied, that is if there is a BSF of a type that the Barrier has previously seen as a response to that request and which came from an application that could have processed that request, the Middle Automaton will not forward the imperative to the Right Automaton, but instead input that answer into itself as if it came from the Right Automaton, marked as a request response to ensure it will eventually be forwarded to the Left Automaton. In the other case, when the Middle Automaton cannot answer the imperative itself, it will record it as an open request (see request management, chapter 14) and forward it to the Right Automaton, where it will eventually be directed towards an application. This concludes Middle Automaton's first data path.

The second data path, as can be expected, processes BSFs coming from the Right Automaton, on their way to the Left Automaton. Two types of data in this stream have to be differentiated, namely BSFs that are intended for the user, and Barrier-BSFs, which serve as a vehicle for an application to communicate something to the Barrier. As usual, the latter are handled separately according to which Barrier-BSF type they are, and then passed on to the Left Automaton in case they also have to be processed there (Barrier-BSFs, analogous to Barrier imperatives, can necessitate more than one Automaton to be correctly processed).

The more complex case is when the data is intended for the user. Note that the application does not know by design which output media are present, nor which is the preferred media by that user for that specific type of data. It is therefore the Middle Automaton's task to correctly set the intended output module. But not only that, it also has to set whether that BSF is wanted by the user in the first place. Both of these criteria, which are central to the Barrier's design, hinge upon the user preferences and the output modules' capacity to display certain kinds of data. Both of which are combined in one central data structure, the thresholds.

If the BSF manages to pass those, it will have the appropriate output module set, and can then be given to the Left Automaton for eventual output. Note that the BSF that enters need not be exactly the one that exits, for through unification of BSFs (chapter 16) while in the threshold algorithm, it could now also contain information from BSF previously in the BSF storage of the Middle Automaton. This concludes the second data path.

13. Introduction

14. Request management

14.0.8. Overview

The request management is the mechanism the Middle Automaton, in conjunction with the Right Automaton, uses to track which imperatives have yet to be answered, and which BSFs have hitherto been recorded for each imperative type. These two tasks have to be distinguished.

The first one (tracking open/unanswered imperatives) is concerned with deciding whether an incoming BSF is the answer to an explicit request by either the user or an application or not. This distinction is of importance, as in the first case the Barrier must not block that BSF, while in the second case it can decide whether that information is to be let through. The tracking of open requests has the additional role of memorizing who the requester is, i.e. where the request came from. This need not be the user, but can also be an application. Hence for the Middle Automaton to forward any BSF in response to a request correctly, it needs to know the source of the request that the BSF answered.

The other task (recording the history of each type of imperative) is paramount for the Middle Automaton to make an informed decision of whether it can answer imperatives from its BSF storage, without having to consult or message applications.

14.0.9. Description

Requests are added to the list of open requests whenever they are about to be passed from the Middle Automaton to the Right Automaton. It is important to state that this takes place after the Middle Automaton decided whether to pass on the imperative to the Right Automaton, so only imperatives which are definitely headed towards an application will be saved as open requests.

Then, whenever a BSF is received by the Right Automaton, at the appropriate time it will be checked whether that BSF is in answer to an open request, as can be deduced from the description of the Right Automaton (page 180). If a BSF is indeed the answer to such an open request, then it is so tagged, and the open request deleted. An effect of this is

14. Request management

that by design an imperative/request can also be answered by only one single BSF. This is to avoid malicious applications sending a steady stream of ‘answer BSFs’, all of which are guaranteed to reach the user. Instead, when an application receives an imperative, it is also given the imperative’s id. This is the ‘key’ with which the application can then mark a BSF it sends to the Barrier as the answer to a request. All it needs to do is send that key along with the BSF, and the Right Automaton will then compare it to the open requests, and decide whether it is a valid key. As is described, the id as key is only valid for one BSF.

This concludes the management of open requests, leaving the management of the request history.

To build records for each type of imperative, the following is necessary:

As is described in section 13.0.7, the mandatory information, which also shapes the form of the request history, is:

1. For each type of imperative,
2. which BSF types
3. from which applications

have been recorded as answers?

Unlike with the data structure for the first task, it should be noted that this is an additive process. Therefore only new records are ever appended to the request history, enabling the Barrier to over time gain an accurate representation of which BSF types to expect for each imperative.

The right time to expand the request history by adding a new record is just after a BSF has been deemed to be the answer to a request. This, as mentioned, already happens in the Right Automaton. The information that is built up in the request history is used in the Middle Automaton, which is a rare case of information being spread or manipulated across the Automata boundaries. It is just after an imperative enters the Middle Automaton, and when it has to decide whether an answer is already in its storage, that the algorithm described in the chapter about the BSF storage (15) makes use of the request history.

14.0.10. Development history, alternatives, and venues of improvement

At first, the Barrier was meant to simply trust applications that when a BSF is labeled to be the answer to a request, it actually is. Later, it was decided that not only due

to security considerations, but also due to traceability of requests and their answer, that would be too simplistic.

Applications need not even be maliciously labeling all of their data to be in answer to a request, but simple malfunctions in an application, which is by definition *outside* of the Barrier, could lead to a conflicted state inside the Barrier, something which is obviously to be avoided. After all, it *is* the Barrier's task to shield the user from anything that is of no value to her, and an application wrongly labeling its data certainly falls in that category.

As a relic to the first simple approach, BSFs still contain a field `isRequestAnswer`, although the Barrier will now check the validity of that data, and reset it if necessary.

14. *Request management*

15. BSF storage

15.0.11. Overview

In a nutshell, the BSF storage is a data structure located in the Middle Automaton which saves BSFs. The BSFs it saves share three common characteristics:

1. They are BSFs destined *for the user*, i.e. they are neither Barrier-BSFs nor are they intended for any other application.
2. The application which generated them sent them to the Barrier's Right Automaton *spontaneously*, i.e. they are not in answer to any specific user (or application) request.
3. They *did not pass* the Middle Automaton's threshold algorithm. That implies their priority and type were not sufficient to 'convince' the Middle Automaton to forward it to any output module.
4. They have their 'persistency' tag set to `true`. That tag denotes whether the data will remain valid for some time to come, from the application's point of view.

These stored BSFs serve both for unification with other BSFs, and as answers to future requests with which applications need not be bothered then (saving resources), because the answer can be considered to already have been computed, and stored in the BSF storage.

15.0.12. Description

The BSFs in the storage can be characterized as data from an application that the user presumably does not specifically care about. The question arises, then, why to save it in a special storage in the first place? There are two major answers to that:

One is because these BSFs serve as input to the unification of BSFs (chapter 16), in which new BSFs of other types can be produced, resulting in possibly successful traversals of the thresholds. In other words, where one BSF might have failed, when combined with another BSF, a new, higher prioritized unified BSF might arise which passes the Barrier and can be catered to the user.

15. BSF storage

The other is because these BSFs could possibly enable the Barrier's Middle Automaton to answer a request from the user without having to consult an application. The criteria that a BSF in storage has to meet in order to be considered the would-be response are described with the Middle Automaton, in section 13.0.7.

The alternative, to simply discard application data, would amount to a waste of information. Worst case scenarios can feasibly be imagined and do not seem unrealistically construed: If the application had just finished an extensive analysis, which the user did not request, that information would then not only be blocked, but deleted. The user, then, if requesting just that analysis 5 minutes (or some other arbitrary amount of time) later, would cause the application to senselessly spend those resources *again*. This might indeed be the very worst case, but even with lesser cases, damage—in the sense of needlessly spent resources—would accumulate rapidly.

BSF storage belongs to those data structures that have to be persistent over different sessions of the Barrier. Therefore, it is serialized and stored on disk, as described in chapter 28.

To be clear about the implications and the design decision of making the BSF storage truly persistent, memorizing the BSF storage means that results that applications produced will be saved over different sessions. Hence, even with the computer system shut down, or the application offline, the user would still profit from the application's previous calculations, be it that they were done one minute ago, or one month.

15.0.13. Development history, alternatives, and venues of improvement

In light of planned additions to the Barrier, improvements to the BSF storage could be tailored to those. It should be kept in mind that the BSF storage is certainly the largest part of the Barrier's persistent state, in terms of size.

At the moment, BSFs are deleted from storage after they are forwarded to the user, or after they have been used for unification. The question is, however, whether that truly makes sense. The rationale behind it is that once a BSF leaves storage in direction to the user, the user will already have seen that data. Not deleting it would, then, amount to multiplying the same information, which the user would then possibly have to see twice. That is, however, just what the user could request. It has to be remembered that if the data in question is deemed to become obsolete with time, then that BSF will *not be saved* or stored anyways. Therefore, if data *does* find its way inside the storage, it is tagged by the application to remain valid.

Another addition would be to effectively have not one, but *two* BSF storages:

1. The one described in this chapter.
2. An additional storage where BSFs are transferred even after they have been forwarded to the user or used up being unified.

That second, new storage would function in a multi-Barrier setup as described on page 245 in chapter 32. In other terms, if other Barriers were to forward requests, the current Barrier could also sift through ‘old’ data that would otherwise already have been discarded. Obviously, the size of that ‘second storage’ would have to be limited such that it does not impede the Barrier’s ‘normal’ processing. If such a multi-Barrier setup is ever realized, it can be surmised that its capabilities would be extended considerably by joining it with this modification of the BSF storage.

15. *BSF storage*

16. Unification of BSFs

16.0.14. Overview

Unification of BSFs is a complex mechanism which enables BSFs to be combined according to unification rules. There are two types of unifications, namely

1. those combining BSFs in a ‘standard’ kind of way, forming a *composite BSF*; it contains all data of its constituent BSFs in an ordered manner.
2. The other type is more powerful, as it allows for arbitrary transformations, which are each associated with one rule.

Although rules are also used in the Left Automaton’s Matcher, these are not to be confused with this mechanism. They were concerned with transforming user input according to translations and decompositions, while these rules serve to combine arbitrary data from applications before it is output.

16.0.15. Description

Unification of BSFs takes place according to unification rules. As there are two types of unifications, there are also two types, albeit very similar, of unification rules. Namely, there are those that result in a unified BSF of ‘type 0’, and those that do not. BSFs of type 0 are composite BSFs by default. In comparison, those that result in another BSF type do not use the standard unification algorithm, but rather have to use a custom data transformation, which is connected with that specific rule. Table 16.1 depicts examples for each of these rule types.

Name	Requisite type ₁	Requisite type ₂	Requisite type ₃	...	Result type	Rule type	XSLT (optional)
EmailList ₁	21	21	–		0	composite	–
EmailList ₂	21	21	–		23	grammar	✓

Table 16.1.: Examples of the two kinds of unification rules.

16. Unification of BSFs

Both of these rule types share an obvious common trait: To trigger, certain prerequisites must be met. In the Barrier, there is no hierarchy of unification rules, nor are there rules that ‘always fire’. The order in which rules are evaluated *does* potentially make a difference, but that difference is considered to be negligible, as each unification is as ‘valid’ as any other. The prerequisites are based on the types of BSF that are processed in a rule, not any other individual characteristics. If all requisite BSF types are present, the rule will trigger, which entails removing all BSFs involved in triggering it, and replacing them by *one single* new BSF, which is called the ‘unified BSF’. Nevertheless, the unified BSF is, apart from its name, a normal BSF.

The actual unification in the Middle Automaton is more complex, though. It employs unifications as described above as a part of a much larger algorithm.

The *goal* of unification in the first place is to find out *all* possible combination of BSFs that the Middle Automaton has access to. When stating all possible, that is indeed what is meant. For such a scheme to function, once all transformations/unifications have been done on the sets of data at hand (the BSF documents), the algorithm will recursively check the (now changed) set of BSFs if, through that unification, other rules can then trigger.

However, it has to be kept in mind that the BSFs ‘used up’ in a unification will still remain in the set. That is necessary because they might be used for other transformations. To avoid data multiplicity, in the process each newly produced BSF frame is given a history of which BSFs were used in its creation, and these will not be used again *in that branch* of the recursive search for new rules to be applied. Only when no further rules can be found, i.e., a fixed point of the set has been reached, will the algorithm terminate. Because unification rules always result in *one* ‘new’ BSF, and because the number of rules is finite, it follows that such a fixed point must exist, and the algorithm does in fact always terminate with finding it, where $\text{unifyRecursively}(\text{set}) = \text{set}$.

Once all possibly reachable unifications have thus been computed, they are *all* put through the threshold algorithm¹ to ascertain whether *any one* of them passes and can be forwarded to the Left Automaton, and consequently the user. This test starts with the most unified BSF, since the number of BSFs that went into a BSF is correlated with the amount of information thereof.

The unification process then concludes with one of two possible outcomes:

- One of the BSFs put through the threshold test is output: All BSFs previously created in this unification process are discarded, and the constituent BSFs, i.e. those BSFs that made up the BSF which was let through, deleted (they may have been in storage before). To identify those, the history of that unified BSF is consulted.

¹For details on that, refer to chapter 17.


```

1 <bsfData>
2   <includedBsf>
3     <includedBsfType>42</includedBsfType>
4     <includedBsfData>
5       ...
6     </includedBsfData>
7   </includedBsf>
8   ...
9 </bsfData>

```

Figure 16.1.: XML structure of composite BSFs (minus the header).

Notice that the history of a unified BSFs, which may itself have resulted from unifying other already unified BSFs, contains not the (new) ids of those, but rather the original ids of the BSF originally in storage, i.e. the most exact composition. After this deletion, the history itself can be discarded as well.

- No BSF is let through the threshold: The same ‘cleaning up’ as in the last point is conducted, except that before that takes place, it is decided *which* BSF is to be saved. That choice will fall on the most unified BSF which has its ‘persistent’ tag active. For details on BSF storage and the ‘persistency’ tag, refer to chapter 15.

16.0.16. Technical description

Arbitrary transformations are saved as XSL stylesheets. The transformations themselves, including the ‘standard’ variety without stylesheets, are done by JAXP XML processing on the BSFs, which themselves are XML documents. While the transformation on the `bsfData` fields may vary, the transformation of the header is always the same. The priority of the unified BSF, for example, will be the maximum of the priorities of all the BSFs which were combined into that new BSF. Hence, a low priority BSF can ‘piggyback’ by unifying with a high priority BSF, provided that a correct rule for unification for these exists.

For technical purposes, when a number of BSFs unify, they are always initially treated as forming a composite BSF, whose BSF data will look like depicted in figure 16.1.

If the rule triggered was indeed only combining BSFs into composite BSFs, the process is complete. However, if an extra XML transformation according to a stylesheet is to be done, it will take this composite BSF to do its transformation on. Therefore, for both types of rules, the composite transformation is always undertaken, while only one type of rule will do further transformation on that data.

16. *Unification of BSFs*

When new rules are introduced by an application, it follows that the XSL stylesheets accompanying the BSF must be aware that their input will not be disparate BSFs but that, just before the XSL transformation is executed, the BSFs are already combined as in figure 16.1.

16.0.17. Development history, alternatives, and venues of improvement

Before XML was chosen to be the language in which BSFs be implemented, much thought was initially given on how BSFs could arbitrarily be recombined. That question was resolved quickly after the aforementioned choice was made, since XSL stylesheets allow for exactly that.

Since it is easiest for stylesheets to operate on a single document, instead of transforming a variable number of documents into a new document, it was then decided that transformations take place on documents already unified by means of the ‘standard unification’, which is done by the Middle Automaton without using XSL.

While this pre-step was not strictly necessary, the line of reasoning was that the application programmer providing the actual XSL stylesheets be bothered as little as possible.

17. Thresholds

Mephistopheles: *I must confess that forth I may not wander,
My steps by one slight obstacle controlled,—
The wizard's-foot, that on your threshold made is.*

—J. W. Goethe, *Faust I*

17.0.18. Overview

Thresholds implement the Barrier's namesake functionality, i.e. they block/let pass each incoming piece of data from applications. In either case, they also determine the output media (output module) best suited for that individual BSF, and save that information in the BSF itself. They are situated in the Middle Automaton. *Each* single threshold is specific to

1. a user,
2. a certain kind of BSF type, and
3. an output module.

Its value can be interpreted as meaning how hard it is for a specific BSF of that type to be passed to a specific output module when a specific user is logged in. The test itself is that value being compared with the BSF's individual priority (which it is given by the application when generated).

17.0.19. Description

In addition to those already explained, thresholds do have other properties as well.

Thresholds can either be active or inactive. Whenever a threshold is active, it is assumed that its associated output module is so, too. When an output module is unavailable for any reason, it will signal that to the Barrier, which will then make the Middle Automaton

17. Thresholds

set the corresponding thresholds to inactive. Similarly, when an output module registers itself, the Middle Automaton will set its thresholds, which will be inactive, back to active. A threshold being inactive is akin to it not existing, while an active threshold is, as the name implies, actively participating in the threshold algorithm. If an output module does not have any active thresholds representing it, no data *can* be directed its way.

The value of a threshold incorporates *two* major pieces of information

1. How well a media is *technically suited* for outputting a certain kind of data, and, rolled into the same value,
2. how likely *the user* is to *want to see* that data.

Both these considerations can be combined into a single value. This combination may seem awkward at first, but does in fact lend itself to this task very nicely: For the Barrier, it only matters whether, *all things considered*, a piece of data should be output. That notion of ‘all things considered’ becomes the value of the threshold. In other terms, the user’s stated desire to see something on the screen will balance—to some degree—that something’s bad suitability for being visually outputted.

17.0.20. Supertypes

As seen in figure 12.1, each BSF can contain a field called ‘supertype’. If that field contains an entry, this means that the BSF will *not* use the thresholds associated with its type, but instead the thresholds of the *supertype*.

Example An application generates a BSF of a very rare and specific type. Because that type is so seldom used, the application can expect with reasonable confidence that the Barrier is *not* well trained as to how to decide in which way to output it. Consequently, the application, before sending the BSF of that rare type, adds a field supertype to the BSFs header. In it, it gives a more common type number. By doing that, the application signals two things:

- The BSF can be considered *as if* it was in fact of the supertype.
- An output module that can output the supertype is certain to be able to output the actual type as well.

Making sure both of these invariants hold is the responsibility of the application, since it is tagging the supertype. However, in the case that it falsely labels a BSF, no lasting harm will be done, except a possibly unsuitable output module notifying the Barrier that it cannot output the actual BSF type.

Supertypes can be viewed as an inheritance scheme, which is only affecting how the Barrier treats certain data; as always, applications and modules can process/output BSF data any way they see fit.

17.0.21. Scenarios

In the example scenario to be presented here, the Barrier is started with *no* output modules active. Since all thresholds are inactive by default, that means that there is no output media for the Barrier to relay information to the user at all. Several messages from applications are incoming, but stored in the Barrier, since there is no valid output module. Note that this scenario is extremely unlikely, as in conjunction with the Barrier there will most certainly *always* be a number of standard output modules started.

Next, a speech output module is started by the user (as a normal program). This output module is capable of reading aloud certain kinds of BSFs (i.e. output), among them e-mail BSFs. When this module registers itself with the Barrier (assuming it has already been installed), its thresholds are set to active.

The next incoming BSF is of a type indicating it is a picture. Since the speech module cannot cope with pictures, it has no thresholds (alternatively: thresholds set prohibitively high, the choice is to the module) in the Barrier specialized for those types, and the picture is also blocked.

The next data to be output, however, is an e-mail with a priority of 5 (from a scale of 0 to 10). Since the e-mail BSF passes the corresponding threshold of the speech module, it is sent to that module, which reads it aloud to the user. According to what can be described as a ‘trodden path’ doctrine, that threshold is then *lowered* from 5 to a lower value, making it *easier* for subsequent data of that type to be given to the speech module.

Since the e-mail in question, unfortunately, was a payment reminder, the user in this scenario tells the Barrier to ‘block the speech module’. Consequently, in addition to a ‘clear current output’ message that the Barrier sends to the speech output module (which then cancels the reading of said e-mail), the threshold is made stricter, by heightening it by a set amount. How such explicit threshold adaptation works is described in the next chapter (18).

17.0.22. Development history, alternatives, and venues of improvement

Thresholds are probably the single part of the Barrier most intended to be ‘modular’. In conjunction with threshold adaptation, this seems the perfect fit for learning AI and

17. Thresholds

machine learning algorithms, such as neural nets. With this in mind, threshold adaptation was kept strictly separated from the thresholds themselves, as these can probably stay unaffected, while only the simplistic change functions are candidates for being upgraded into AI-controlled adaptive algorithms.

18. Threshold management

18.0.23. Overview

Threshold management describes the mechanism by which the Barrier adapts to an individual user. Over time, thresholds, i.e. the defining factor in deciding

- what the Barrier blocks, and
- by means of what media each type of data is given to the user,

will closely mimic user preferences. There are many venues for the user to directly or indirectly influence a threshold. Thresholds are based in the part of the Barrier most concerned with the state of the Barrier, the Middle Automaton.

18.0.24. Description

Threshold management can be divided into two kinds: *direct* and *indirect* user commands. Direct commands in this context are those whose function it is to *explicitly* alter the manner in which something is output.

Altering the manner, then, means subtracting (adding) a fixed number to the appropriate threshold whenever an accepting (rejecting) effect—either a direct or indirect user command—occurs.

As a general rule, direct commands are valued vastly more than indirect user commands in respect to their influence on thresholds. Using direct commands will let the user swiftly ensure or alter specific behavior by the Barrier, such as that incoming BSF data of a certain kind (e.g. type 231 = tax refund report) are without prompt read aloud, while another kind (e.g. type 232 = payment request from the IRS) is quietly discarded in the Middle Automaton.

In contrast, indirect commands are mostly normal interactions with the user, which the Barrier also uses to slowly converge to the desired settings, without the user having to set her wishes explicitly.

18. Threshold management

id	command	meaning
11	“reject \$x1 :outputModule”	
12	“reject output”	reject most recent output (strong version of reject); affects thresholds on all modules
13	“reject output on module”	reject most recent output (strong version of reject)
14	“always reject \$x1 :outputModule”	
15	“always reject output on module”	lock the threshold to always reject most recent output on that module
16	“always reject output”	lock all threshold of most recent output to always reject it
17	“clear \$x1 :outputModule”	
18	“clear output”	clear most recent output, i.e. stop displaying or reading, etc.
19	“accept \$x1 :outputModule”	
20	“accept output”	accept most recent output; affects thresholds on all modules
21	“always accept \$x1 :outputModule”	
22	“always accept output on module”	always accept the output on this module
23	“always accept output”	always accept the output on this module (equivalent to 23)
24	“redirect \$x1 :outputModule to \$x2 :outputModule”	redirect current output on module \$x1 to module \$x2
25	“redirect output to \$x1 :outputModule”	redirect the most recent output to module \$x1

Table 18.1.: Exhaustive list of ‘threshold imperatives’ that explicitly influence thresholds. Note that other imperatives may also have side-effects on threshold values.

Table 18.1 is an exhaustive list of all *direct* ways with which the user controls thresholds, and their assorted meaning. (Note: All modules in this list must obviously be output modules. If the meaning of a command is self-explaining based on its phrasing, no ‘meaning’ will be given.)

The role of the key word “always” in that table needs further elaboration. While all the influences on the thresholds are by default transient, i.e. can be changed and further adapted, this could come as an inconvenience to the user. That inconvenience could be due to non-standard patterns of usage, such as a user *always* wanting certain types of data displayed on the, say, screen, regardless of all redirections to other modules or clears he enacts on that data type. To solve this, and give the user a means of permanently fixing something to his liking, the notion of fixed thresholds was introduced. A fixed threshold is one not subject to either direct or indirect influences, in fact, it is just what the name

implies—fixed. In other terms, the command with id 12 would still reject the most recently output data, in the sense of sending a “clear” message to that output module. *However*, the secondary effect of heightening the threshold for that type and module, could *not* take place if the threshold is *locked*, e.g. through a previous command involving the key phrase “always”.

Apart from these direct influences, another mechanism is in place which is designed to, ideally, slowly converge towards the actual (in the sense of theoretical) user preferences, *without* the user having to explicitly direct the Barrier. These are user commands that exhibit side effects on thresholds, for example just by often invoking certain BSF types in response. This leads to a seeming adaptation of thresholds without *any* user input. So the user *is* often indirectly involved, as she requested that data in the first place. Whenever the Middle Automaton passes on a BSF which is about to be output, the threshold for that BSF type and the output module it is directed to is made more permeable. This signifies that a so-to-speak ‘trodden path’ is established, which will eventually (without user interference) become a strong bias of a certain BSF type towards a certain output module, simply on the basis of customary law (i.e. “it has always been like that”). Also, the “clear” message in itself serves similarly to “reject” in itself, albeit like a *much* weaker version of it.

Figure 18.1 shows an example of such a command at work, namely how the “accept output” command functions. Several comments are necessary: The command’s purpose is for the user to express that the data which was output *last* meets with her approval, and she wants future data of that type to be let through easier. Hence, all thresholds that exist *for that type* are lowered by a certain amount, making future passings more likely. Therefore, picture 18.1 only shows thresholds for those (output) modules that *have* thresholds for the specific BSF type that was last output. Note that “accept output” is based on that data in general (affecting all such modules), rather than the current combination data–output module. There are other commands if just a single threshold is to be changed.

An example scenario for the threshold imperatives 18 and 25 of table 18.1 is given in section 8.0.11.

18.0.25. Development history, alternatives, and venues of improvement

Threshold adaptation in the current prototype follows a rather simple scheme. There are many techniques from the realm of machine learning and statistics that might be tried to make the mechanism more precise and robust and to converge more quickly. The best starting point would probably be to set up a *Bayesian network* (nodes for every threshold and every threshold imperative) and to then incrementally learn the net parameters, i.e.

18. Threshold management

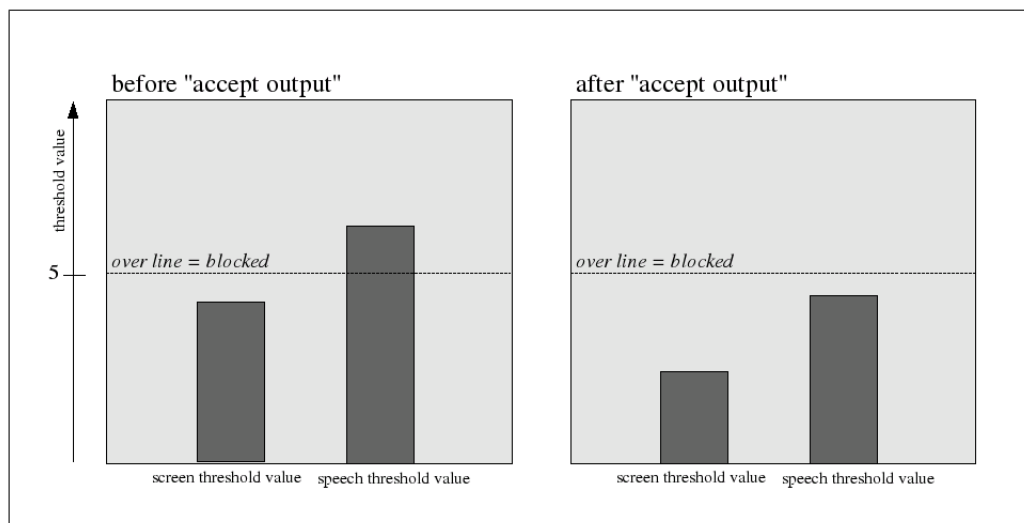


Figure 18.1.: Before/after picture of the “accept output” command, which makes the data type *most recently output* more likely to be let through in the future to any output module it has a threshold with.

probability tables, as described, e.g., in [RN03, page 720ff.].

Also, the values by which the threshold values change are *constant*, which would eventually need to be changed (e.g. a user repeatedly rejecting the same data might want that repetition, which is a kind of emphasis, to be weighted higher than linearly, e.g. polynomially).

Part VI.

Right Automaton

19. Introduction

“Ihm nach – ihm nach, was zauderst du? – Coppelius – Coppelius, mein bestes Automat hat er mir geraubt – zwanzig Jahre daran gearbeitet – Leib und Leben darangesetzt – das Räderwerk – Sprache – Gang – mein – die Augen – die Augen dir gestohlen. [...]”

—E. T. A. Hoffmann, *Der Sandmann*

19.0.26. Overview

The Right Automaton controls the third of the Barrier that is closest (and connecting) to the applications. To them, it provides the Barrier’s API, processes their data and sends them their commands. Whether the applications communicate to the user or among each other, the Barrier is by design always in between—and the Right Automaton’s API is all of the Barrier that the applications need to be concerned with.

19.0.27. Description

As with the other Automata (Middle Automaton and Left Automaton), there are two data paths, one in the direction from user to application, and one *vice versa*. The latter can be sketched as follows:

The Right Automaton accepts a new incoming BSF, which is transmitted as described in the chapter about the communication setup (27). Then, before the BSF is let further into the Barrier, it is checked for being well-formed, i.e. corresponding to its respective definition. Further information about the definition of BSFs can be found in the BSF chapter, starting on page 139.

After that check has been successfully completed (otherwise processing terminates at that point), the Right Automaton will set all BSF headers that need to be set, such that the Barrier can always correctly identify each individual BSF. Chief among these headers are

19. Introduction

the time stamp and a unique BSF id, which can serve as the primary key for that BSF. The BSF will be forked to a separate method if it is marked as a Barrier-BSF, i.e. if it is meant for the Barrier itself. In that case the next steps are skipped.

Deciding whether the BSF is an answer to a request requires a more complicated algorithm. This consists of checking the BSF's temporary id, which is set by an application (but is optional and need not be set at all) against the list of open requests, whose management is described in chapter 14 about request management.

If there is a match, the BSF is marked to be the answer to a request, ensuring that it will not be blocked anywhere in the Barrier. However, there is more to that. If the BSF is expected as the answer to an open request, then there is also a record of where that request actually came from, i.e. who issued it. This need not be the user, but can also be another application. If so, then that answer BSF is sent directly to that application. In all other cases, namely if the requester was the user, or if there was no requester at all, or if it was a Barrier-BSF that has been handled by the Right Automaton, the BSF is forwarded to the Middle Automaton for further processing.

The other major pathway runs in the counter-direction. There, the Right Automaton starts with accepting imperatives from the Middle Automaton, handles them and eventually terminates by forwarding them to an appropriate application for real processing. What does it mean, then to be an appropriate application? The receiver-to-be of an imperative must satisfy several criteria:

1. The application *must* be registered in the Right Catalog (description on page 183) for that type of imperative to be eligible for processing it.
2. The application must be started, and must have registered itself with the Barrier to be ready for processing.
3. If there are several candidates satisfying the above criteria, the application must have won the contract negotiation by having the biggest capacity among its competitors (cf. arbitration, chapter 22).

If there are no such candidates to be found, the Right Automaton can try to rectify the second of these points on its own, by starting an application that is registered for that imperative. There must be such an application, otherwise the Left Automaton could not have matched the command against its own Catalog. Once the application is started, it will register itself as ready (on its own), after which the Right Automaton can finally forward the imperative to the application. The steps involved in the forwarding operation itself—i.e. the communication process once both the message (which is the imperative) and the recipient (which is a specific application) are determined—is described in chapter ‘Communication with the applications’ on page 197.

19.0.28. Technical description

Applications are started by looking up the start command which is saved alongside the application entry in the list of installed applications. That command (say, represented by the `String` variable `startCommand`) is then executed by calling

```
Runtime.getRuntime().exec(startCommand),
```

which runs the command as though it were entered on the Windows command prompt (the shell-like DOS environment invoked by executing the program `cmd`) or on a Linux shell.

The validation of incoming BSFs—described on page 179—is implemented by means of XML validation. The package used is `javax.xml.validation`.¹ It works such that an object of class `Validator` is created, already tuned to a certain XSD. The XSD, in our case, represents a specific BSF type. By calling the validator's `validate` function with a specific XML document as a parameter, a `SAXException` is returned if the validation is unsuccessful.

So, in short, that method seems to be state-of-the-art for validating XML against a schema.

¹Documentation available at <http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/validation/package-summary.html>

19. *Introduction*

20. Right Catalog

20.0.29. Overview

When first introducing the Catalog concept in section 6.3, we made the remark that some of its data and functionality is accessible to the Right Automaton only (cf. 6.3.3). These features will be described now.

Remember that the Catalog stores all the services (in the form of imperatives that form the Catalog entries) that are offered by the collective set of applications. While the Left Automaton only needs to know *which imperative* (i.e. what services) are offered, the Right Automaton must also know *which applications* are registered for each of them, for it is this part of the Barrier that is in charge of forwarding an imperative to the application which it considers best suited for the job. Also, it must keep this information up-to-date whenever an application wants to register or unregister itself for a new imperative.

20.0.30. Description

Registration for an imperative can be done in two ways:

1. The imperative can be passed in the shape of its *logical form*, i.e. as an XML document as produced by the Barrier's parser (cf. 6.1.5).
2. The imperative can be passed in *natural language*. (This method is vastly easier for the application programmer.)

In the latter case the natural-language sentence must first be parsed (by the Barrier!), for Catalog entries encapsulate imperative objects, whose core data structure is a logical form. After that the first method may be called with the resulting parse. Using natural language for registering an imperative is more accommodating and definitely more realistic than for the application to parse the sentence beforehand—we cannot expect application programmers to gain any insights in what kind of parser the Barrier uses before making their program support the Barrier. This is why *only the second method* is implemented in the prototype as of now.

As soon as we have a valid parse,¹ the application that initiated the registration process is added to the Catalog entry that represents the imperative to register. If no such imperative is to be found in the Catalog yet—if no other application has registered for that command before—, a new one is created. If an application wants to *unregister*, the registration is undone simply by removing that application’s name from the Catalog entry representing the respective imperative.

Applications register and unregister for an imperative by sending a BSF to the Barrier,² identifying themselves with their unique name (e.g. `de.tum.in.barrierapps.bobertino`) and indicating in natural language which imperative is to be registered (e.g. “compose a message”) and whether it requires additional raw data (cf. chapter 7).

Each entry in the Catalog has a unique identity number (*id*). When later the Barrier forwards a user command to an application, it will not send the logical form of the imperative, but rather the imperative’s Catalog id. The reason for this is obvious: applications shall not be bothered with handling logical forms, for this is a Barrier-internal mechanism that should be transparent (in the sense of invisible) to the outer world. So when an application registers for an imperative it must get in return the id of the Catalog entry representing that command, to later know which number stands for which command. This is done by the Right Automaton sending a BSF of type `catalogIdForImperative` back to the application.

20.0.31. Development history, alternatives, and venues of improvement

As already mentioned, when an application registers a new natural-language imperative in the Catalog, we first check whether there is an existing entry in the Catalog representing such an imperative. But what does “representing such an imperative” mean? The natural-language imperative may have several parses, and so does each Catalog entry. Do we require the sets of parses to be equal, or should one set be a subset of the other, or should their intersection just not be empty? We chose the last and weakest of these conditions: as soon as two sentences share at least one logical form we consider them having the same meaning.³

In terms of development history, it is worthwhile noting that, to begin with, we envisioned

¹This might not even be possible if the imperative to register is not conforming to the grammar (in this case a simple English grammar allowing only commands). Then the process is interrupted with an exception.

²Cf. types `registerForImperative` and `unregisterImperative` in the table of Barrier-BSFs (table 12.2).

³The only conceivable problem with this would occur in this constellation: let there be three imperatives with sets of parses $I_1 = \{a, b\}$, $I_2 = \{b, c\}$, and $I_3 = \{c, d\}$, respectively. I_2 is already in the Catalog and I_1 and I_3 shall be added. Then I_1 and I_3 would end up in the same Catalog entry (that of I_2) although they have no parses in common. This scenario seems, however, utterly contrived, and we could not come up with of a real-world example for it.

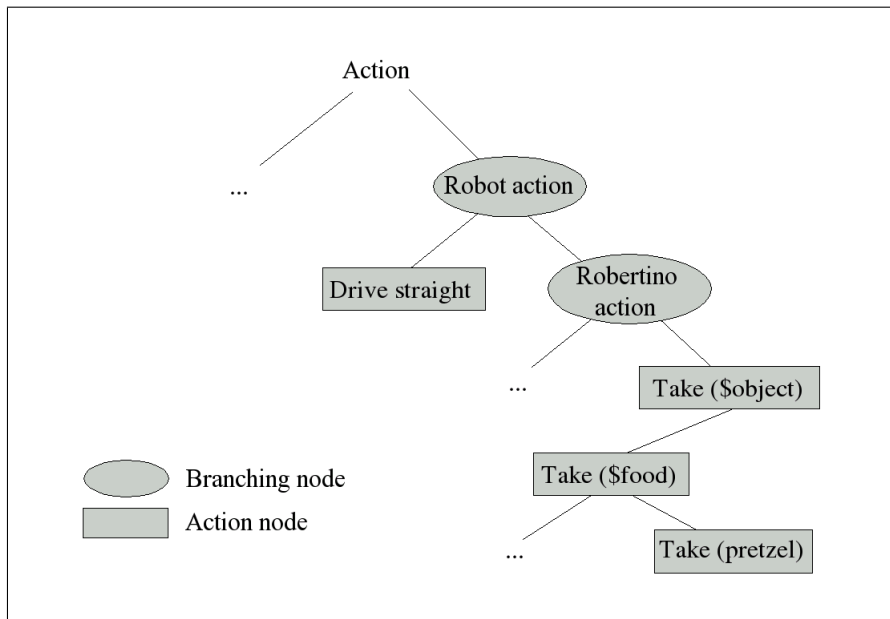


Figure 20.1.: Part of the old ontology initially planned for use in the Barrier.

something like an *‘imperative ontology’*—a set of all possible standard Catalog entries—prescribed at the Barrier’s design time. Applications would then have registered for these pre-defined entries. An extension of the set of possible entries on an application’s behalf would have been an extra feature. Applications would have had to indicate the entry in the form defined by the imperative ontology. Figure 20.1 shows part of the type hierarchy initially envisioned.

With the introduction of OpenCCG into the Barrier this was no longer necessary, as applications could now use natural language: the format for passing entries to the Barrier for registration is now independent of the internal representation in the Catalog—a conspicuous advantage. This way every imperative that complies with the grammar can be registered; the grammar *implicitly defines* the imperative ontology that otherwise would have had to be manually and explicitly assembled—another conspicuous advantage.

Nonetheless, a basic set of imperatives would be a valuable feature, for without such a standard, different applications could happen to register slightly different imperatives for what is semantically equal if the Rulebook does not contain rules for converting one into the other. This is definitely not contradictory to the Barrier’s concept, but still it would be desirable for the sake of perspicuity if the same actions bore the same names. So we could conceive of a ‘recommendation’ of imperative formulations that would replace the fixed ontology envisioned in the beginning.⁴ Note that these would in fact just amount to recommendations, which are written down not ‘in code’, but as part of the documentation.

⁴One such recommendation could be: “It is recommended by the Barrier developers to express the action of downloading electronic mail as ‘download e-mails’.” (As opposed to, e.g., “fetch messages”.)

20. *Right Catalog*

21. Application management

21.0.32. Overview

An important task of the Right Automaton's is application management, including:

- installing and uninstalling applications
- registering and unregistering applications
- picking the best application for a given user command
- starting applications

In this chapter it will be shown how these mechanisms have been implemented in the Barrier prototype.

21.0.33. Description

Figure 21.1 shows the two data structures used for application management:

The class `Application` provides a description of single applications, which are fully defined by these data:

- Each application has a *unique name*, which in itself fully identifies it. We follow the Java naming convention for package names:¹ a unique application name should start with the reverted domain name of the company or organization in charge. This is why our example applications start with `de.tum.in`, e.g.

`de.tum.in.barrierapps.bobertino.`

- An application's *start command* must be known to the Barrier. This is for the case that an application that is meant to process a user command is not running yet and must thus first be started. A start command must be the name of an executable file.

¹cf. <http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html>

21. Application management

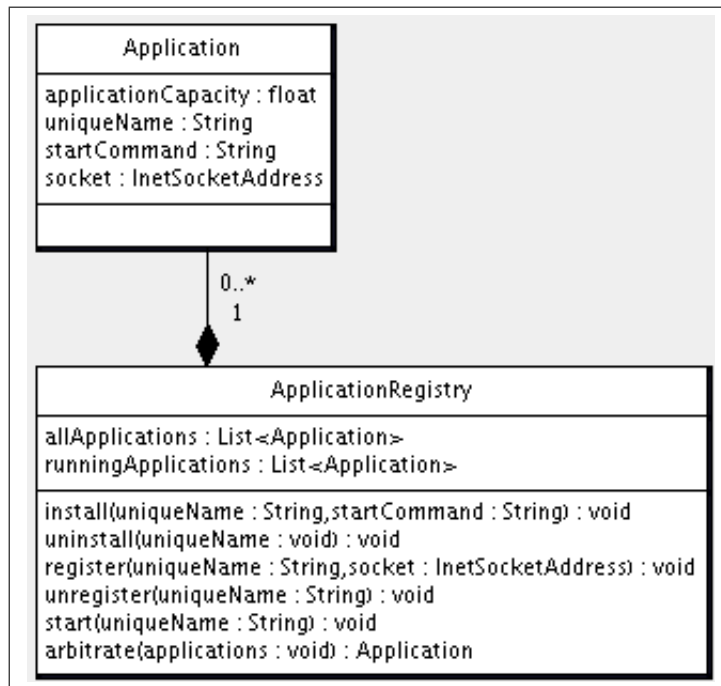


Figure 21.1.: Overview of the application management in shape of a UML diagram.

- For communication between the Barrier and an application to be possible, the latter’s network address (in form of a socket) must be stored in the description.
- As described in chapter 22, arbitration is done based on a *capacity* (or ‘resource status’) applications can communicate to the Barrier. If an application does not inform the Barrier about its current capacity, a standard value is assumed.

Note that the class **Application** has *nothing* to do with the actual implementation of an application; it serves solely for describing its properties within the Right Automaton.

Now that we have seen how information about a single application is stored, let us look at how we keep the set of all known applications up-to-date. This is done in the class **ApplicationRegistry**, whose core data are two lists:

- A list of *all known applications*, i.e. of all Barrier applications that are currently installed.
- A list of all *running applications*, i.e. of all applications that are currently registered with the Barrier.

We require that *only installed applications can register*: First an application must be introduced to the Barrier by being installed (“Hello, my name is ...”), only then can it say “I’m up and running” (by registering).—The set of running applications is a subset of all applications. Installation need take place only once: after that it is enough for the

application to register whenever it starts up and to unregister when it closes down.

Note that the set of applications is *dynamic*, i.e. applications that are not previously known to the Barrier may be installed at any time. This is just what is the case with any modern operating system; there, as long as an application uses the operating system's interface functions properly, the application will run properly.

Such maintenance actions are performed by the class's methods:

- When **installing**, an application must provide its unique name and its start command. It is then added to the list of all known applications.
- When **uninstalling**, it is simply removed from the two aforementioned lists.
- When **registering**, the application identifies itself with its unique name. Additionally, it must indicate its current network location (a socket); this is not done at installation time already so that an application is not bound to one single computer: the network location may vary from start-up to start-up.
- When **unregistering**, the application is removed from the list of running applications. It remains in the list of all known applications.

These four actions are initiated from outside the Barrier by sending dedicated BSFs (Barrier-BSFs) containing the respective data for each action (as listed). When they are received by the Right Automaton, the above methods are called with the data from the BSF.

The remaining two methods are called when a user command (an imperative) reaches the Right Automaton 'from left', before it is relegated to an application:

- If several applications from the list of running applications are found to be eligible for processing the imperative (based on the Catalog, cf. chapter 20), one has to be chosen. This is done by the method **arbitrate**, whose argument is the list of eligible applications and whose result is the winning application. The details of arbitration are laid out in a separate chapter (22).
- If no apt application can be found among those in the list of running applications one from the list of all known applications must be picked and started. Note that in this case no arbitration is possible because this process currently relies on data provided by the applications themselves—and to do so they must be running in the first place. In this context, arbitration would therefore not make sense, because no application could apply, or 'vie', for a contract. The Barrier would have to find out for itself which of the (offline) applications to use. That, however, is an addition tagged for future incorporation.

21.0.34. Technical description

To make the above more conspicuous, figure 21.2 displays an example for the type of BSF an application must send to the Barrier in order to register (just the data part is shown, for the BSF to be complete it must, of course, also have a header).

```
1 <bsfData>
2   <applicationName> de.tum.in.barrierapps.bobertino </applicationName>
3   <ipAddress> 131.159.60.93 </ipAddress>
4   <serverPort> 1099 </serverPort>
5 </bsfData>
```

Figure 21.2.: Example of the data part of a registration BSF.

While implementing the `start` method, attention had to be paid in one place: The time the start-up process takes may vary from application to application; it might be several seconds long. We can, however, communicate with the application only after it has registered (since we must know its network location). So we must not simply return from `start` as soon as we have called

```
Runtime.getRuntime().exec(startCommand)
```

(Java’s way to execute an external program) but rather wait until the application has sent a registration BSF (or until a certain timeout is over). This is done by using the `wait-notify` mechanism that implements the semaphore concept in the Java language: We `wait` in the `start` method after issuing the start command to the runtime system and `notify` the semaphore when `register` is called for the respective application. Only then will we return from the `start` method.

Also, one must be careful to maintain the invariant mentioned before: that an application can only register when it has previously been installed. Additionally, we do not want an application to register or install twice. All such irregularities are answered by throwing an appropriate exception.

21.0.35. Development history, alternatives, and venues of improvement

Within the framework described here, it is also easily possible to merely start an application—without any specific intended action. For this the respective application, say Bmail, simply has to register for an imperative such as “start Bmail”, which will, just as any other imperative, be added to the Catalog. Now imagine Bmail is not running yet and the user utters “start Bmail”. As Bmail will be the only application registered for this imperative it will be started as delineated above and be sent the imperative “start Bmail”, which it will

simply ignore, since the only goal of that command—starting the application—as already been achieved!

For this to be possible, application names would have to be words in the vocabulary of the grammar. One might envision, for a future version, that whenever an application is installed, its name is automatically added to it. If the vocabulary is made extensible (as sketched in section 6.1.7), the application could also itself register its name as a new word.

Additionally, an application could register for a more general imperative such as “start e-mail client”. Then, other e-mail applications could register for that imperative as well, and the user, when giving that command, would get one of these applications started.

Still, there is additional room for improvement. As described at the end of subsection 21.0.33 on page 189, arbitration could also be brought into play when deciding *which* of several *offline* applications to start. A sensible approach to that would be for applications to provide some kind of persistent capacity, which the Barrier memorizes even if the application is offline. While this might be somewhat delusional (trusting applications to correctly report their capacity), other more sophisticated ideas would be to

- use information about the *computer* the application was running on mostly by now, or
- keeping count of how many requests an application answered, thus creating a kind of ‘favorite applications’ registry. This last suggestion is, however, rather distant in the future.

Enabling offline arbitration, when tied in with the previous proposed improvement, would enable the following scenario:

The user invokes the command “start e-mail client”. Three applications are registered for that rather generic command, all of them can be assumed to be e-mail clients (they would not sensibly register otherwise). The Barrier would then use the arbitration mechanism to find one ‘winning application’, which would then be started. It would be considered best suited.

21. Application management

22. Arbitration

22.0.36. Overview

Arbitration takes place whenever the Right Automaton has more than one application to choose from when deciding where to transmit an imperative. It is based on the Right Automaton recording the ‘offers’ that applications make, and then deciding which of these is best. The winner then gets the ‘contract’, i.e. receives the imperative for processing. The Right Automaton does not yet function as an arbiter afterwards, and adapt its rating according to how ‘well’ an application performed its task (no post-contract evaluation).

22.0.37. Description

The arbitration is meant to be one of the parts that are interchangeable. The current implementation of this part is still lacking, as it only uses basic indicators rolled into one single measure. In addition, it trusts the applications with their information without any verification.

At the moment, applications can send a special kind of Barrier-BSF to the Right Automaton, which will update their ‘resource status’, which the Right Automaton tracks for each installed application that is running. Then, when several applications are in a position to vie for an imperative, i.e. are registered for handling it, this resource status is the crucial factor in deciding which application is selected.

At this stage, this is a straightforward process: simply the application that has indicated the highest capacity is chosen.

22.0.38. Development history, alternatives, and venues of improvement

At the moment, the contract negotiation is based solely on the resources that each application has available. In addition, the Barrier relies on each application reporting those accurately, the values are not rechecked for accuracy. If no capacity is provided, a standard value will be assumed.

22. Arbitration

It could prove to be a worthwhile and obvious candidate for extension to base the negotiation on other factors as well. For example, one might have the Barrier judge the capacity of applications on its own, ranging from factors as network load to memory available; as depending on the command to be executed, requirements may vary, and different resources might be weighed differently.¹ In that case, the data which applications report themselves could be used just to supplement the evaluation the Barrier has already computed from these new factors.

The major such factor would be, as mentioned, to include a post-contract evaluation of whether an application has processed a specific imperative in a satisfactory manner. That knowledge would then be used to heavily bias the selection process for or against that application in future arbitrations. But how could the Barrier gain that knowledge? There are several possibilities (not exhaustive):

1. The application could report itself whether it was able to process the command correctly.

(Obviously the same flaws as to the current algorithm apply. The application is still trusted implicitly, and also: How can an application even judge whether the data it has computed is ‘correct’?)

2. The user could, when receiving the data computed by the application in response to one of her commands, tell the Barrier optionally whether she considers that data to be correct.

(Several drawbacks: What about commands issued not by the user but by another application? Also, the user is trusted to be able to tell correctly which data is correct and which is not. That judgement might be false. However, since the system is meant to adapt to the user to the highest degree possible, this might not be a bad consequence. After all, if the user insists on false information, who is the Barrier to contradict her?)

3. The Barrier could simply check whether an open request *is* eventually answered by the application to which it was forwarded. If no answer is forthcoming, ‘bad’, otherwise ‘good’.

(Supposedly an appropriate general heuristics, however it fails for certain cases: What about commands that are not *supposed* to generate any feedback [i.e. BSF in answer], but are still fully—and correctly—processed by an application? These would be—incorrectly—labeled as faulty processing by the application.)

¹ When designing such a multi-factor arbitration (or evaluation) function, one has again two choices: (1) hard-code it—certainly not the best way to do it—or (2) learn it. The latter could be done elegantly with neural nets: the different capacity features would serve as input neurons, and the weights of the neural net would be learned by backpropagation in a training session.

These three approaches are all, as described, situational. However, they can be considered to complement each other, if all implemented. Each would contribute a ‘weight’ towards the Barrier’s evaluation of whether a command was processed correctly. Even if one of these turns out to be in the ‘false direction’ (correct processing considered incorrect or *vice versa*), the general convergence towards the optimum should be unimpeded; i.e. the Barrier could still in most cases (presumably) reflect how well an application copes with an imperative. That would hold true as long as at least two of the three above measures contributing to the final evaluation are correct most of the time. The validity of that assumption would certainly have to be checked empirically, but it seems reasonable to assume so.

Not a major approach for that specific improvement, but rather a detail: There are many design methodologies already in existence for the general area of contract management, specifically from the ‘agents’ subfield of computer science. The above approaches are tailored to the specific needs of the Barrier; they would nevertheless need to be rechecked and themselves evaluated based on knowledge from that field.

22. Arbitration

23. Communication with the applications

23.0.39. Overview

In chapter 12 we introduced the Barrier Structured Format, BSF for short. Its purpose is to serve as a format for messages exchanged between the Barrier and applications, in both directions. Applications wrap their output in BSFs and send them to the Barrier so they are passed to well-suited output modules.

Here we will go into more detail about how BSFs are used to send back and forth data between the Barrier and applications.

As mentioned, we can distinguish two directions:

1. *Application to Barrier*: An application sends its output to the Barrier in shape of a BSF; also applications need to ‘speak’ to the Barrier itself, e.g. to tell it that they want to register for an imperative.
2. *Barrier to application*: An imperative representing a user command is sent from the Barrier to the application (the one that has been found to be best-suited for doing the job).

23.0.40. Description

Application to Barrier

The simple protocol applications have to follow to register for an imperative is described in chapter 20 about the Right Automaton’s Catalog. Here it suffices to note that this is done the canonical way: by sending a BSF.

Note that in this situation the application wants to communicate with the Barrier only, rather than using it as a broker for data meant for the user. For all such situations applications make use of dedicated BSF types which we call Barrier-BSFs (see section 12.1). A list of all Barrier-BSF types and their usages can be found in table 12.2; e.g. an application also needs to send a Barrier-BSF when it wants to tell the Barrier that it has just been started or that it is about to shut down.

23. Communication with the applications

```
1 <bsfData>
2   <imperative>
3     <request id="123"/>
4     <catalog-entry id="345"/>
5     <label-table>
6       <entry label="$x1">
7         <value val="e-mail"/>
8       </entry>
9     </label-table>
10  </imperative>
11 </bsfData>
```

Figure 23.1.: Example of the data part of an imperative BSF.

When the Right Automaton receives a BSF that encapsulates an application’s output, i.e. when it wants to communicate with the user rather than the Barrier, it is first checked whether the BSF conforms to the respective type definition by using an XSD validator. More details on this and on how the BSF is further processed inside the Right Automaton before eventually being sent to the Middle Automaton can be found in chapter 19 about the Right Automaton.

Both regular and Barrier-BSFs ‘percolate’ through all three Automata: whenever one of them has processed a BSF appropriately (this also comprises the case that no action is performed at all) it passes it to the Automaton left of it. This is obvious for regular BSFs, since they must reach the output modules via the Left Automaton (unless they are not admitted for display, see chapter 17), but it is also true for Barrier-BSFs: one Barrier-BSF might be of interest to several Automata; e.g. when an application registers a new imperative it must be stored in the *Right Automaton’s* Catalog (chapter 20) and the *Left Automaton* must notify the output modules of the event so they can display the up-to-date Catalog to the user (chapter 8).

Barrier to application

The only information that runs in this direction is imperatives.—As only BSFs are exchanged between the Barrier and applications, imperatives are wrapped in BSFs, too.

23.0.41. Technical description

Figure 23.1 shows an example of a BSF wrapping an imperative (type `imperativeForApplication`).

The `request` element serves to uniquely identify this user command. The `catalog-entry` element stores the id this imperative has in the Catalog. This is because we do not send the imperative itself to the application but rather its unique identifier in the Catalog.

This is, however, not enough if the imperative as registered in the Catalog contains variables. The example id 345 might, e.g., stand for “send \$x1 :virt-thing”. The user supplied concrete words to fill the variable slot and they, too, must be sent to the application alongside the imperative’s id in the Catalog. Such information can be found in the ‘label table’ that is attached to imperative objects (section 6.3.2 described how this table is filled). So what must be added to the BSF is an XML representation of the imperative’s label table: for each variable label, the concrete words that took the place of the variable in the user utterance are supplied.

This is, however, not straightforward, since an imperative object stores the user command in its parsed logical form (cf. section 6.2); thus also the label table—which is computed from the latter—takes this form. But we do not want to encumber applications with this representation, which is meant to be Barrier-internal. We rather want applications to receive the label table in *natural language*, e.g. we want them to receive

“e-mail”

instead of the parsed XML structure of figure 23.2.

```
1 <diamond mode="patient" class="virt-thing">
2   <nom name=":virt-thing"/>
3   <prop name="e-mail"/>
4   <diamond mode="noun">
5     <prop name="e-mail"/>
6   </diamond>
7   <diamond mode="num">
8     <prop name="singular"/>
9   </diamond>
10 </diamond>
```

Figure 23.2.: Logical form representing the word “e-mail”.

The mechanism to transform a logical form back into natural language is called *realization*, which may thus be considered the inverse of parsing. We deployed the realization tools provided by OpenCCG to produce an XML representation of the resulting label tables, similar to figure 23.1.

Note that there may be, for one the single label (like \$x1), several elements of the form `<value val="..." />`. The reason is that realizing one logical form may result in several

23. *Communication with the applications*

natural-language expressions.

23.0.42. **Development history, alternatives, and venues of improvement**

Alternatives for label table realization

As has often been the case while developing this prototype, the simple looking way of passing the realized label table to applications was not at all obvious from the start.—In fact the whole idea of a label table was developed for this purpose! The need for realization is also why we had to write an XSL stylesheet that reverts the one that makes logical forms more easily processable for the Barrier (cf. 6.2.3).

The alternatives to the realized label table would have been:

1. to send the un-realized logical form to the application, or
2. to realize the *entire* user command rather than just the expressions that replace variables.

As mentioned, we consider the first alternative too cumbersome for application developers. So let us look at the second one:

Assume “send e-mail” as the result of the realization operation. We would also have realized the corresponding Catalog entry, which could possibly contain variables; assume the Catalog entry realization “send \$x1 :virt-thing”. From this natural-language string we would have created a regular expression, replacing variables with wildcards; in our case this would yield “send (.+)”. The next step would have consisted in matching the concrete user input with the regular expression, memorizing the matches of bracketed sub-expressions, which would have been something like an implicit label table.

Such a solution works, however, flawlessly just in a simple situation as the “send e-mail” example. We encounter problems when dealing with Catalog entries that feature more than one variable: e.g. in “send \$x2 :person \$x1 :virt-thing” (assume the concrete user input “send me an e-mail”), the above mechanism would produce the regular expression “send (.+) (.+)”, in which the first bracket would match “me an”, whilst the second one would match “e-mail”—which is obviously wrong, the correct solution being “me” and “an e-mail”.

Storing the label table explicitly turned out to be an elegant and robust way to circumvent such problems.

Application failure

Currently we assume that, once an application has been chosen to fulfill a user command, it will be able to do so without failing. Especially for Catalog entries with variables this might, however, not always be the case: an application might not be able to do the job for any ‘virtual thing’ substituted for a variable like “\$x1 :virt-thing”.

For such cases one might build into future Barrier versions such a contingency mechanism: If an application fails to correctly process an imperative, it can communicate this failure back to the Barrier by means of a special BSF, which also contains the label table that the application was given. The Right Automaton can then check for another application registered for the same imperative, and resend the imperative to that one.

23. *Communication with the applications*

Part VII.

Applications

24. Introduction

24.0.43. Overview

This section's purpose is two-fold:

- to explain how applications are connected to the Barrier, and implicitly from that how existing applications can be modified to also support the Barrier (mainly by stripping the GUI I/O and associating commands with strings instead of buttons) and
- to document the two example applications that the Barrier prototype works with.

24.0.44. Description

Applications subscribe/register for commands, the data structure of which, called imperative, has often been encountered by now. If an application is registered for a command, it is expected to be able to execute it. Given more situational commands such as “close file”, or “open file”, it is expected that an application registers for those only during time spans in which it can cope with them. Contextual registering/deregistering is not only possible, it is encouraged.

Consider the following chain of events as an example:

- All applications are idle. Since no file has been opened, no application has registered for “close file” or “close \$x1 :file”. If the user were to issue such a command anyways, it would *not* be passed to any application, but rather returned to the user with a request for clarification or a message that, although the command was parsed, the system cannot execute it (which is correct, since it cannot—having no application available that can).
- An application opens a text file for editing (either on its own, or prompted by the user). Since *from that point in time* on, there is an open file that can be closed, that application will register for its favorite expression for closing a file (e.g. “close file” or “end editing”). Since imperatives are registered by applications on a simple per-string basis (which may include variables), the phrasing is up to the application.

24. Introduction

- If the user, or by proxy another application invokes that command and the file is closed, the application that registered for “close file” will then unregister for it, since that operation cannot be executed any longer, just as was the case at the very beginning.

Another example in a different context, with a fictitious file sharing application “eel-eagle”:

- An application starts up and registers with the Barrier (but *not* for the command “upload \$x1 :file” nor “log in as \$x1 :name”).
- The application connects to the central server in Sweden, *then* registers for “log in as \$x1 :name”.
- After the user has issued the log-in command to the Barrier, the application will then *register* for “upload \$x1 :file”, and *unregister* the log in command.

Evidently, an application is expected to only accept commands for those periods of time when they actually make sense for that application. This also much facilitates disambiguation, because if only one running application has opened a file, only one application will receive it (only it will be registered for it in the Right Catalog).

As usual, applications will also understand variants of commands, as long as these represent most common variations in the English language. To be more precise, as long as the Parser can parse them as valid sentences and the Matcher match them against the command as stored in the Catalog, using (among others) translation rules. However, the translation rules incorporated at this point are anything but exhaustive, but that can easily be remedied.

The Barrier prototype comes with two example applications, namely Bobertino and Bmail.¹

24.0.45. Technical description

Figure 24.1 shows a UML class diagram for Barrier applications, as implemented in the Barrier prototype.

Important note on developing applications for the Barrier prototype

Everything described in this entire part about applications (part VII) is *not* fundamental for the Barrier *itself*. Instead, the whole existing application infrastructure, including

¹Technically, it comes with a few more, since modules are also specialized *applications*. For reasons of clarity, however, they are excluded for these considerations. For details on them, refer to chapter 9.

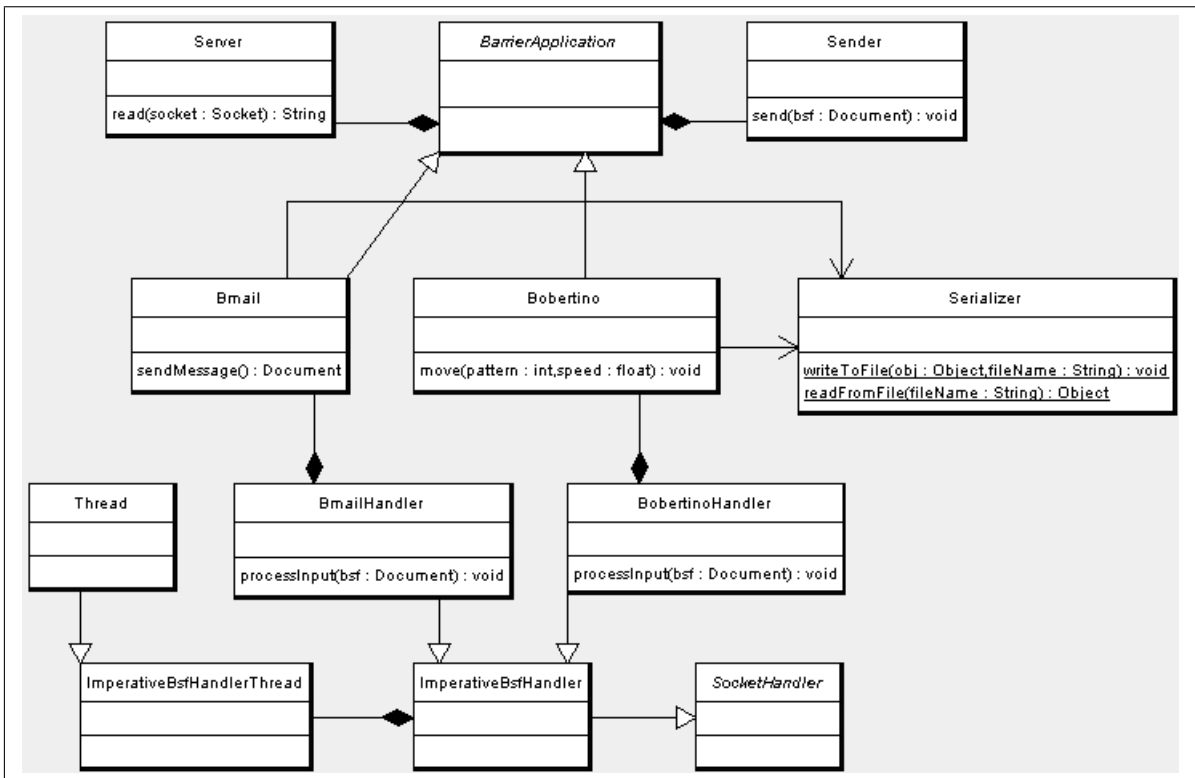


Figure 24.1.: UML class diagram for the two applications' Barrier infrastructure; only example functions portrayed.

24. Introduction

classes with their serialization etc., should be regarded as ‘offers’ for making an application Barrier-compliant more easily.

To clarify that notion: When using the application classes as given in the Barrier prototype’s source code, close to all effort can go into the application’s actual functionality, as even the connection overhead to the Barrier will already be dealt with. However, any application conforming to the Barrier’s communication protocol (i.e. connecting to the correct port on the correct machine, and knowing what kind of answers to expect, and how to register data types with the Barrier etc.²) will work just as well in conjunction with the Barrier. It need *not* be extended from, or using any functions of, the classes used by the example applications.

The example applications were in fact necessary for the Barrier prototype to have something to interact with, not because they are an integral part of either the prototype or the Barrier specification (although much effort did go into them, as well as designing reusable classes to be able to extend that ‘offer’ to Barrier developers).

²cf. chapter 27

25. Example applications

25.1. Bobertino

25.1.1. Overview

Bobertino¹ is an application controlling a Robertino robot. Robertinos, a sample of which can be seen in figure 25.1, are small, mobile robots equipped with an omnidirectional drive.

The robot's core processing board is a PC 104 which has 64 MB of SDRAM at its disposition. It is running RT (real-time) Linux and can be controlled via Wireless LAN (802.11g): there exists a C++ function library for connecting to the robot from stationary PCs. So Bobertino can in principle run on any computer in WLAN reach of the robot itself.

Robertino's commercial version is called 'Robotino' and is manufactured by Festo.²

25.1.2. Description

The Bobertino application is able to execute the following commands:

- “take picture”: captures an image using Robertino's camera (a Logitech webcam), and sends it to the Barrier.
- “stop”: stops the robot.
- “drive \$x1 :direction”: Bobertino will order the robot to drive in the direction specified, lasts until canceled or substituted, or until a collision is detected by Robertino's infra-red distance sensors. The direction can be any of the following:
 - “forward” (also: “north”),

¹Short for **Barrier Robertino**.

²For a detailed technical description of Robotino, cf. <http://www.festo-didactic.com/int-en/learning-systems/new-robotino/the-mobile-robot.htm>.

25. Example applications



Figure 25.1.: A Robertino robot, as controlled by the Bobertino application.

- “backward” (also: “south”),
 - “left” (also: “west”),
 - “right” (also: “east”),
 - “clockwise” (also: “circle”): will cause the robot to drive a clockwise circle, and
 - “wave”: will cause the robot to drive a sine wave.
- “drive \$x1 :speed \$x2 :direction”: works analogous to the previous command, except that the standard speed is substituted by the speed specified, which can be any of the following:
 - “fast”, or
 - “slowly”.

If no speed is given, a ‘normal’ velocity, which is the average of the two given above, is used.

- “dance for \$x1 :person”: only few select persons did currently make it into the Barrier’s dictionary, and since the Left Automaton must initially parse the command from the user by means of that dictionary, few options exist for this command to be executable.

25.1.3. Technical description

The Bobertino application is, just as is the Barrier, also written in Java, but since the library provided is in C++, it relies on a Robotino API Language Extension (‘RobotinoWrapper’³) that provides a programming interface for accessing the C++ library from Java Code.

It is implemented⁴ using the Java Native Interface.

A side note regarding the command “take picture” that is first in the above list: images come from the webcam in JPEG format, which is binary. Bobertino’s answer to the “take picture” command must, however, be a BSF, i.e. plain ASCII code. For converting the binary image data into a string representation we use the standard way called *Base-64 encoding*.⁵ The screen output module, before displaying such a picture, decodes it back into binary data.

25.2. Bmail

25.2.1. Overview

The Bmail application is an example e-mail client that is compatible with the Barrier. It only provides the most basic mail functions, and is mostly a proof-of-concept to show how an e-mail application can easily interact with the Barrier.

25.2.2. Description

The Bmail application supports the following commands:

- “download messages”: will fetch new messages from the mail server and forward them to the Barrier, where they will be delivered to the requester (need not be the user, can also be another application!).
- “compose message”: causes Bmail to prepare a new message and wait for its constituent fields to be set. Once “compose message” is initiated, Bmail will *unregister* for “compose message”, and register for “set recipient”.

³Implemented by Robert ‘Wrapper’ Isele of the Technical University of Munich and available at http://www.openrobotino.org/index.php/Robotino_API_Language_Extensions.

⁴cf. http://www.openrobotino.org/index.php/Robotino_Java_Wrapper

⁵Base 64 is e.g. the standard for encoding e-mail attachments.

25. Example applications

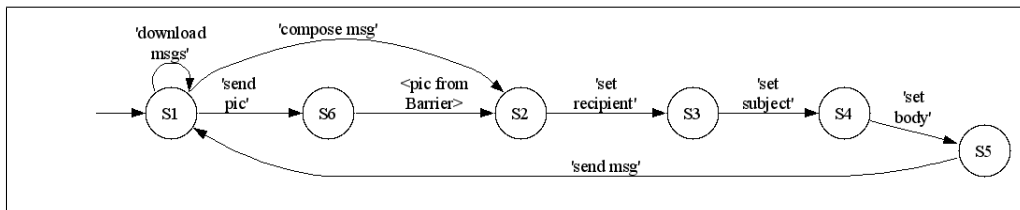


Figure 25.2.: A finite state machine representing Bmail's behavior.

- “send picture”: same as “compose message”, additionally, after Bmail received it, Bmail will *without the user being involved* issue a command to the Barrier requesting a picture that is to serve as an attachment to the e-mail. See 31.0.24 for further explanation.
- “set recipient”: causes Bmail to set the recipient for the message to be sent. Once complete, Bmail will unregister for that command and register for “set subject”.
- “set subject”: sets the subject field in the message, then proceeds analogous to above by unregistering for the current command and instead registering for “set body”.
- “set body”: sets the body of the message. When complete, this will ‘unlock’ “send message” (by registering for it).
- “send message”: last command in the sequence to send a new e-mail, will actually send the message over an SMTP server.

All of the above commands that serve to set a field in the e-mail (i.e. all except “compose message” and “send picture”, which just kick-start the process) need a ‘raw data’ section, for which the user will be prompted by the Barrier. “Send picture” is the exception, as Bmail will request its own data from the Barrier for that.

Figure 25.2 shows a finite state machine representing Bmail's behavior. Edge labels enclosed in single quotes stand for the imperatives that cause the state transitions. This implies that Bmail, when in state S_i , is registered for all the imperatives that label its outgoing edges. When such an edge leads into a state $S_j \neq S_i$, then the respective imperative will be unregistered before entering state S_j . One ‘user session’ is completed in one graph traversal from S_1 back to S_1 .

To get to S_6 , Bmail makes a request “take picture” to the Barrier, and the subsequent transition to S_2 is made as soon as the requested picture has arrived from the Barrier. Confer section 31.0.24 for a more detailed description of this very scenario, which is an example for how applications can communicate with each other through the Barrier—without even knowing of one another's existence!

25.2.3. Technical description

Bmail's e-mail functionality is realized through the use of JavaMail,⁶ which is a “framework to build mail and messaging applications”.⁷ The JavaMail API Reference Implementation is known under the same name.

As of April 19, 2006, the developers of JavaMail, Sun Microsystems, Inc., have made JavaMail OpenSource under their CDDL open source license, which made it more attractive for usage in an example application, since it can then be bundled with the Barrier prototype with no concerns towards licensing issues.

To change the mail-server etc. settings, the appropriate parameters in the `Bmail` class need to be changed, and Bmail recompiled (obviously not the whole Barrier, since Bmail is a *separate* application. This is definitely not the most convenient way, but, after all, our implementation—particularly of the example applications—is only meant to be a proof-of-concept, and making such data directly customizable for the user is feasible by making just a minor technical change in Bmail.

⁶available at <http://java.sun.com/products/javamail/>

⁷also <http://java.sun.com/products/javamail/>

25. *Example applications*

Part VIII.

Miscellaneous and advanced topics

26. System configuration and performance

$$1 = \sqrt{1} = \sqrt{(-1)(-1)} = \sqrt{-1}\sqrt{-1} = -1$$

—Mathematical proof?

26.0.4. Overview

The Barrier was tested mainly on the two systems that it was developed on, both running a significantly different configuration (including the operating system).

26.0.5. Description

Test setup

The first notebook (system 1) was a 1.6 GHz, 2 MB L2 Cache Intel Centrino notebook, using 512 MB of SDRAM. It was running an updated version of Windows XP (including Service Pack 2).

The second notebook (system 2), with lower performance, was a 1.4 GHz, 1 MB L2 Cache AMD Athlon notebook, using 256 MB of SDRAM. It was running SuSE Linux, kernel version 2.4.21. Since the speech recognition module only works with Windows, for all tests that have it run on that system, the operating system was changed to Windows XP.

The test cases used to judge the Barrier's performance were the following:

1. The Barrier and all modules and applications running on system 1 (or system 2, respectively), called test case 1a (1b).

26. System configuration and performance

2. The Barrier running on system 1 (system 2), with the modules and applications running on system 2 (system 1), called test case 2a (2b).
3. Case 1, but without the CPU-intensive speech recognition, called test case 3a (3b).
4. Case 2, but without the CPU-intensive speech recognition, called test case 4a (4b).

Results

In the first test case, CPU usage has peaked at 100% for both variants (both systems). However, with system 1, the response time remained adequate for working with the Barrier (subjectively) close to real time. In case 2, the system was workable as well, but not as smoothly.

In the second test case, the system that was running the Barrier itself ran similarly to the third case, whereas the counterpart system running modules and applications did not show significant performance difference in comparison to the first case.

For the remaining two test cases and variants, it can be uniformly said that CPU usage seldom increased above 50%, and there was no highly noticeable increase with using the Barrier. With both systems, uninterrupted and unimpeded usage of the Barrier was possible.

The *conclusion* to be drawn, then, is that not the Barrier itself was draining resources, but resource-intensive peripheral components.

This suspicion was confirmed through running the speech recognition software on its own, without the Barrier turned on. Note that this comparison is not 100% applicable, as the Barrier uses only the API, whereas the stand-alone product has other tasks associated as well (GUI, training methods, etc.). However, a correlation seems highly likely, especially given that the difference between the first two and the last two scenarios was just the speech recognition software.

The performance results of the Barrier's prototype can therefore be called quite satisfactory. The efficiency (or lack of it) of modules and applications, as seen with the speech recognition example, should not be transferred onto the Barrier's performance itself. Given an efficient application (or module), the overhead of making it compatible with the Barrier is minimal, and mostly consists of communicating with it (by opening a Java server, as described in chapter 23 about communication with applications).

These results seem sensible, and were to be expected, as the Barrier mainly implements *control functionality*, as opposed to *data mining/data processing* functionality. Note that building a more sophisticated prototype (confer the multiple 'venues of improvement'

subsections of the single chapters) *might* significantly increase resource requirements, but that is to be expected. Then again, hardware resources have for some time increased steadily (or rather exponentially) as well.¹

¹cf. 'Moore's Law', i.e. the number of transistors on integrated circuits doubling every 18 months

26. *System configuration and performance*

27. Communication setup

27.0.6. Overview

Although it is strictly divided into the three Automata, the Barrier exists as one entity in our basic concept, while input and output modules as well as applications are considered entities of their own. Here is a list detailing what that means:

- Programmers¹ must be able to build modules and applications without any knowledge of the Barrier's inner workings. They are not supposed to know any of the data structures used internally.
- As the Barrier, written in Java, is itself platform independent, the Barrier should also allow applications and modules of all kinds of systems:² no matter on which operating system they are running, it must be possible to plug them into the Barrier.
- Barrier compatibility shall also not depend on the programming language in which a module or application is written.
- The Barrier setup in a broader sense (including modules and applications, that is) is meant to be a system that need not run on one single machine: it shall be possible for the actual Barrier (the three Automata) to run on one computer, while the modules and applications may be spread over a multitude of machines. This calls for a network-enabled communication setup.

All these abstractions make one communication paradigm particularly apt: sockets. The socket concept realizes all the abstractions listed: from implementation details, programming language, operating system, and network location. Requiring programmers to call Java interface methods would allow to treat the actual Barrier as a black box, too, by abstracting from implementational details, but it could not afford the remaining abstractions. Using a system for distributed object management, such as CORBA (which abstracts in fact one more step from sockets), could solve the problem as well, but as we shall see, the

¹When speaking of programmers in this section, we always mean programmers of input/output modules or applications.

²This is crucial in the context of ubiquitous computing, with the Barrier intended to provide control over many different devices throughout the room.

27. Communication setup

data that are communicated are very simple in structure (basically just character strings), and using something as powerful as CORBA would be more of an overhead than of a boon.

Permitting the system to be spread over a network is definitely an advantage: this way the workload is distributed and it is no more one single computer that must process all the information; in our tests this turned out to be particularly useful in context with the speech input module, which, as it uses memory-intensive natural-language recognition software, was not well suited to be run on the machine hosting the Barrier; so it was simply started on another one that communicated with the first over the network.

Also some applications or modules may only run under some operating systems and not on others. This was, again, the case with the speech recognition software (Dragon Naturally-Speaking) whose libraries our speech input module uses: it supports Windows only. Even if the Barrier runs under Linux, this is no problem in the network-enabled communication mechanism, for the speech recognition module and the Barrier can simply run on different machines.

In a future scenario of ubiquitous computing (which is definitely not viable with our prototype yet) one could even image a house that features small sensors with controllers attached to them all over the place. They would serve as input modules by recording the user's utterances and passing them on to the Barrier, which could run on a PC in the basement. After processing the user's demands, the results could be displayed by means of output modules that would also be spread all over the house.

Note that, as all communication is done through sockets, sending data to remote computers is done the same way as to local modules or applications.

27.0.7. Description

The communication setup is displayed as a UML diagram in figure 27.1. The classes are not listed with all their fields and methods, of course, we rather restricted the diagram to just show the essential parts.

The two ends of data communication are taken by the **Server** and the **Sender**.³ A **Server** has a **ServerSocket** that listens to a specific port on the machine it runs on; this must take place in a separate thread because *listening* means *waiting* for incoming connections, which blocks until there eventually is a sender that wants to send data to the server. So each server runs one single **ServerThread**; it is started when the server's **start** method is called and does nothing but wait (in an infinite loop) for connections initiated by **Senders**.

³Whenever we write 'server' or 'sender' in this section, we actually mean a **Server** or **Sender** object, respectively.

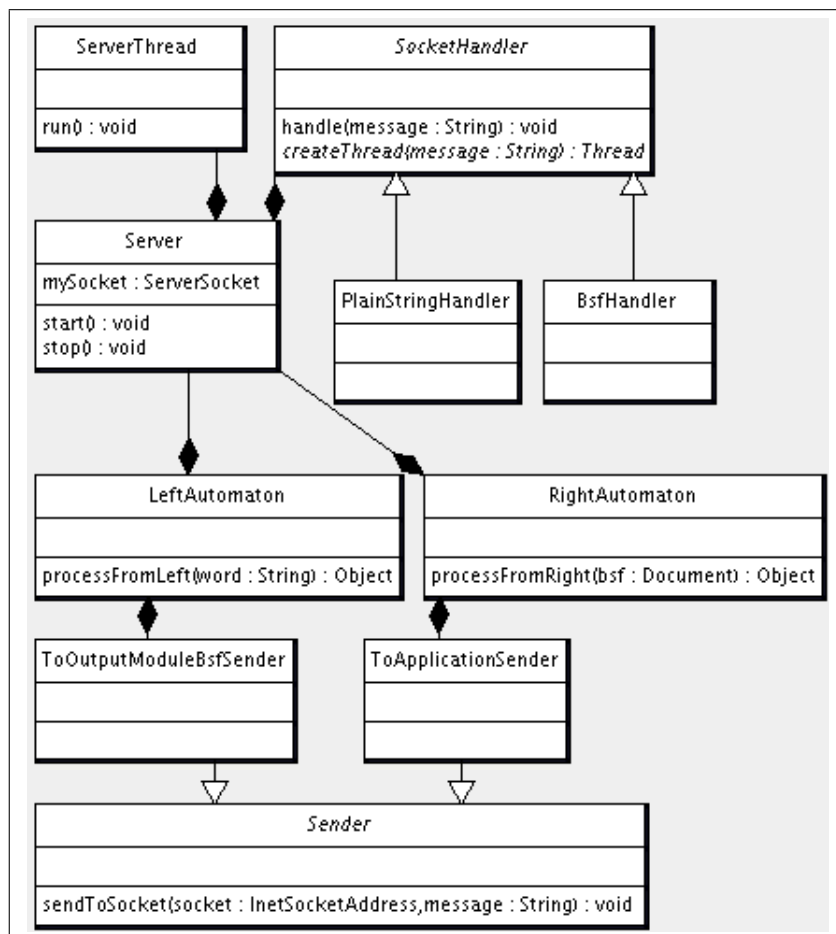


Figure 27.1.: UML diagram of the communication setup.

27. Communication setup

Whenever it accepts a connection it merely reads the data from the socket but does not deal with it itself, it rather passes it to the server's `SocketHandler` by calling this object's `handle` method. This is again to avoid any blocking while the request is being processed; the server thread shall be able to quickly, with as short delays as possible, accept connection after connection. Thus, its single task is to tunnel incoming data to the handler, which performs the appropriate action in yet another thread. (`handle` calls `createThread` and starts the resulting thread; all the data-processing functionality is encapsulated in this thread's `run` method.)

The system design makes data handling quite simple: There is only one server class for all purposes, although different owners might want their servers to handle the received data in different ways: e.g. the Left Automaton expects simple character strings from input modules, whilst the Right Automaton only deals with strings representing BSFs, which are particular XML documents, from applications.

To enable such distinct behavior, the server's handling methods are 'outsourced' into the *abstract* `SocketHandler` class. To specify what must be done when data are received all one must do is extend this handler class by implementing the abstract `createThread` method; as mentioned, all the functionality must be put into the `run` method of the thread it returns. Referring back to the example of the Left and Right Automata, the former uses a `PlainStringHandler` in its server; the handling thread simply calls the Left Automaton's `processFromLeft` method with the data read from the socket. The latter has a `BsfHandler` instead, which calls the Right Automaton's `processFromRight` method. (Data need not even be read from the socket by the handler, this is done by the server itself.)

Data can be sent to a server as strings through the `Sender` object. The protocol is intentionally kept simple by allowing unidirectional communication only: data can be sent from a sender to a server, but not the other way; this makes the basic protocol *asynchronous* because no reply is to be expected for any data sent.

27.0.8. Scenarios

To make the process more conspicuous, we will just briefly look at one example: Suppose the camera input module has recorded the word "left" and wants to pass it to the Barrier. It sets up a socket connection with the Barrier (port 1077, for the sake of completeness) and writes the string "left" to it. The Left Automaton's server accepts the connection in its server thread and reads "left" from the socket. It calls `handler.handle("left")`, which in turn passes "left" to the Left Automaton in a new thread, so the server has done its job and can immediately wait for the next connection from an input module.

27.0.9. Development history, alternatives, and venues of improvement

The factoring out/separation of data processing from the basic server functionality, as laid out in the description, makes programming less prone to errors and keeps the code clean. This turned out to be a great advantage because we used the same server and sender classes not only in the Barrier itself but also in our example applications and modules. This was easily possible because all of them are written in Java, too.

As pointed out at the beginning of this chapter, however, this is by no means mandatory: we could have written them⁴ in any other language. This is where the simple, asynchronous communication protocol and the string-only data format come in handy: all that is to be done is to use a server socket in the programming language chosen (all common languages offer socket implementations) and read a character string from it. Sending is even easier: simply connect to one of the Barrier's servers (the left or the right one) and pass a string to it.

⁴The applications' and modules' servers and senders, not the Barrier's.

27. *Communication setup*

28. Serialized data

28.0.10. Overview

Many of the data the Barrier needs in order to function properly are not of a transient nature: they rather need storing over the system's whole life-span. During that life-span the computer on which the Barrier is running may, of course, be shut down, resulting in the Barrier's being shut down, too. So not to lose that data, there must be a way of securing it. This will be done by writing the data to persistent memory, i.e. by storing it in files on the hard-disk. Java provides a simple and robust way for that purpose if the data involved are Java objects, just as in our case; this process is called *serialization*.

An example of a data structure that must be maintained persistently is the Rulebook the Left Automaton's Matcher uses for transforming incoming user utterances into imperatives conforming to the Catalog of available commands: it is defined only once and shall then be accessible to the Barrier all the time, even if the system is shut down and restarted.

28.0.11. Description

Table 28.1 shows all classes that are serialized in the Barrier's prototype. It also shows the file names that the serialization results in, and a very short allusion to the purpose of that class. Of course, only the most crucial data is saved in persistent serialization. From the short description, it should be evident why each specific object of such a class had to be saved. Each of them are necessary (but also sufficient) to guarantee the state of the Barrier will be preserved *across shutdowns*.

As described in chapter 24 about applications, since neither those nor modules need to subclass from the class `BarrierApplication` (but can be written in another language, even running on another operating system), the entries in table 28.1 that refer to either application or module need not be considered, and are instead simply 'offers' for persistent storage mechanisms already in place, storing/reloading the data needed in order to communicate with the Barrier. Using it enables a developer supporting the Barrier to further minimize the development overhead, and concentrate nearly solely on the actual functionality he aims to provide.

	class filename	role	serialized data file name
	ApplicationRegistry.java	information on installed and running applications	applicationRegistry.ser
	BsfTypeDefinition.java	all known BSF types and XSDs	bsfTypes.ser
	Catalog.java	commands and their parses	catalog.ser
*	Bmail.java	Bmail startup information	de.tum.in.barrierapps. bmail_hasBeenStarted.ser
*	Bmail.java	stores ids of all Bmail-processable commands	de.tum.in.barrierapps. bmail_imperativeMap.ser
*	Bobertino.java	stores ids of all Bobertino-processable commands	de.tum.in.barrierapps. bobertino_imperativeMap.ser
*	KeyboardModule.java	stores ids of all KeyboardModule-processable commands	de.tum.in.barriermods. input.keyboard_imperativeMap.ser
*	DragonModule.java	stores ids of all speech-processable commands	de.tum.in.barriermods. input.speech_imperativeMap.ser
*	ScreenModule.java	stores ids of all screen-processable commands	de.tum.in.barriermods. output.screen_imperativeMap.ser
	InputModuleRegistry.java	information on installed input modules	inputModuleRegistry.ser
	OutputModuleRegistry.java	information on installed output modules	outputModuleRegistry.ser
	RequestHistory.java	usage history for that Barrier	requestHistory.ser
	Rulebook.java	the Matcher's Rulebook	rulebook.ser
	StorageBsfs.java	blocked BSFs for later unification and/or output	storageBsfs.ser
	Threshold.java	all thresholds present	thresholds.ser
	Threshold.java	output modules with thresholds	thresholds.ser_modules
	UnificationRule.java	rules applied when unifying BSFs	unificationRules.ser
	UnificationRule.java	counter so that every rule has a unique id	unificationRules.ser_counter
Note: * not a Barrier serialization but rather specific to an application/module; serialization for applications/modules provided through inheritance from class BarrierApplication			

Table 28.1.: Serialization in the Barrier: classes, files, and usage.

So, while table 28.1 also includes the serialization of applications and modules for purposes of documenting the Barrier prototype's peripherals, we only explain the serializations performed by the Barrier itself in what follows.

28.0.12. Technical description

Serialization takes place by making the class whose objects are to be serialized implement the interface `Serializable`.¹ It occurs as the Barrier is terminated in a regular manner, and the deserialization (data being loaded into objects from disk) as the Barrier is started up.²

Consequently, if the Barrier prototype is not terminated in the canonical way (by sending an "exit Barrier" command), *no* data from that session will be serialized. In a full-fledged commercial implementation of the Barrier, or most likely even in a more extended beta build, the Barrier will probably be including another thread whose task it will be to periodically serialize the Barrier's data (most likely while system load is low), guaranteeing that even with an unexpected runtime exception or similar, as little state as possible is lost (cf. the 'safe-points' concept).

For further details concerning what files the Barrier will create to store its persistent state, cf. table 28.1.

28.0.13. Development history, alternatives, and venues of improvement

Initially, it was planned to transform the important stats of the Barrier into XML form, and save them in that manner. That, however, was later considered to be prone to error as it consisted of a lot of manual modifications. Thus, the current approach was decided on. Although it has the drawback of the serialized data not being easily editable by hand, it is a robust standard method, for which many standard bugs have already been rooted out based on extensive usage by the Java developer community.

¹Documentation available at <http://java.sun.com/j2se/1.4.2/docs/api/java/io/Serializable.html>.

²The prototype applications/modules work accordingly, i.e., when they start up/quit regularly.

28. Serialized data

29. Automatic GUI generation

29.0.14. Overview

Within the Barrier, the automatic generation of graphical user interfaces (or GUIs)¹ is considered to be of such importance as to warrant a chapter of its own in this report.

We think that what is enabled by this feature may well be a new paradigm in UI programming: the elimination of the very task of UI programming itself from the process of application development!

The task of designing a UI (graphical or of whatever kind) may in our system be completely neglected during the design of a program's functionality. All user interfacing is done by output modules of which an application and its programmers do not even know.

In short, output modules always have a current list of all user commands that can be input and processed by the Barrier. Out of this list, which is not in a parsed format, but rather in the more intuitive form of natural-language strings, an output module can generate arbitrary UIs of its own liking, representing the commands and data offered by applications it will never know (and that is considered an advantage).

Figure 11.1 showed how our example screen output module makes use of that list of commands to generate a simple GUI.

29.0.15. Description

Commands in the GUI can be categorized into two kinds:

- 'Standard' commands that the Barrier always provides, even without *any* applications being installed. These are for instance all commands concerned with threshold management.

¹Actually, of user interfaces (UIs) in general, it is only the prototype that has an emphasis on graphically displaying commands. Therefore, whenever this chapter talks of 'GUIs', one should think of universally applying the very same to UIs as well.

29. Automatic GUI generation

- Commands that were registered by applications. These represent actions that the applications that registered for them can process. As long as the command is a parseable natural language expression (by the Left Automaton), no limits exist.

The output modules getting that list are now fully free to decide how best to display it, or (see the ‘scenario’ subsection 29.0.16 of this chapter) whether to display them at all, and instead just concentrate on ‘normal’ BSF output. The output module simply has all the data it needs available; it is fully among its powers to decide independently what to do with it.

Specifically, the aspects that output modules can decide upon include the standard variety of UIs: ordering, menu structure, general layout etc. This may also include many other factors, or some may not be applicable at all. An output module specializing in speech might just as well read aloud all or some of the commands. It might be programmed to say aloud “new commands now available”, followed by a list of these newly received new commands (which the output module knows by comparing the updated Catalog, broadcast after each change, with the older version, which the application was free to keep).

Concerning GUI generation, however, not *all* is left to the discretion of the output module. There is one rather important restriction by design:

Output modules do not know which applications are registered for which commands. Even the Left Automaton does not know that, as that knowledge is encapsulated solely in the Catalog of the Right Automaton. While traditional GUI methodology might not agree with withholding that information, it is imperative to the new paradigm advocated through this paper that these layers be entirely separated.

Rather than hinder the correct ordering of commands, we consider it to be an advantage: finally doing away with the arbitrary distinction of commands along program lines. The user, it is surmised, does not care about which application handles her commands, as long as it is an application qualified to do so. Forcing the user to make that choice herself, being concerned with program names, having to know what program provides what capability etc. etc. is rather an unduly heavy burden for a user. It is also unnecessary for a user to have to know which application to tell her request. It should be enough to just tell it to the system, and the actual applications that lend the system their capabilities can remain in the background.

Although users today are used to all that knowledge of what the individual capabilities of each program are, it might provide an entirely different experience when these context-dependent boundaries are lifted.

It is then the only thing an output module can do to order commands along *semantical* lines, grouping commands such that their ordering *makes sense*, not such that the com-

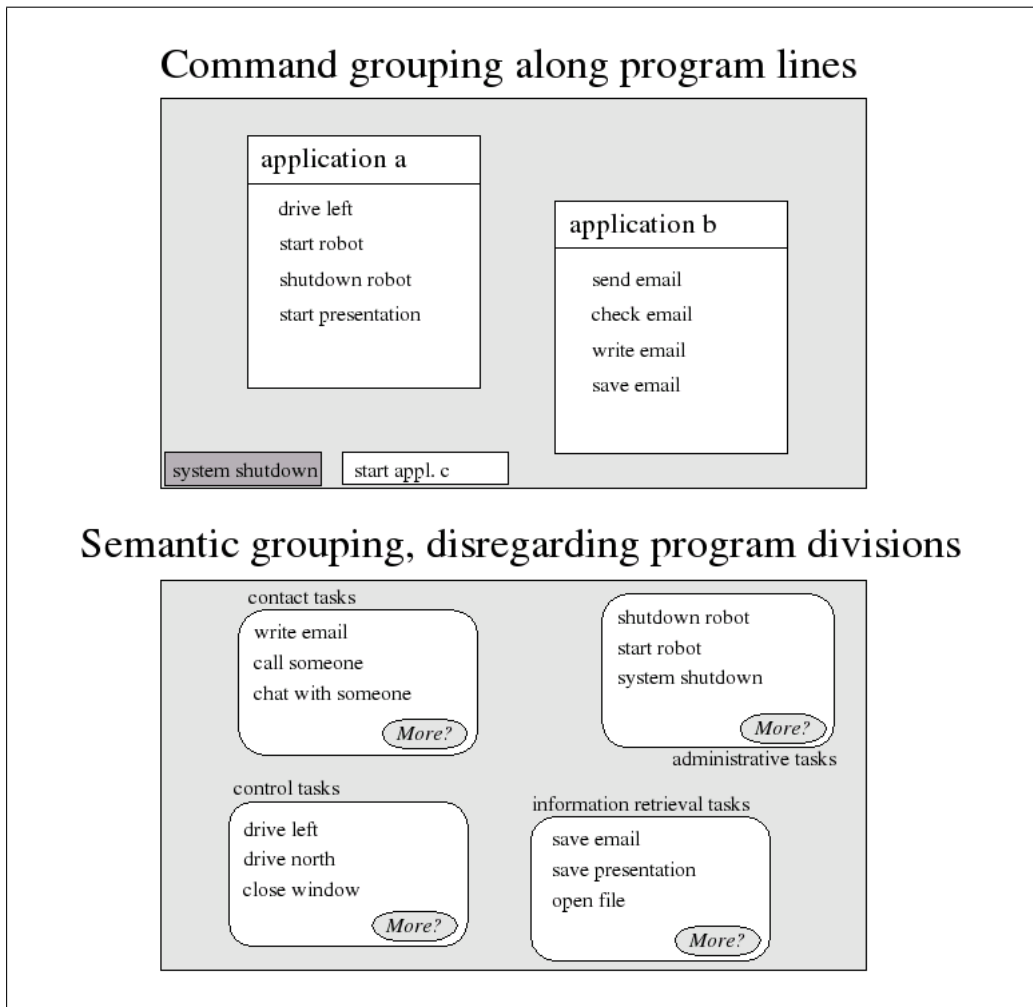


Figure 29.1.: Juxtaposition between a normal desktop and one using semantic grouping enabled by the Barrier.

mands each application provides are located within one window, as is the practice in nearly all contemporary programs. The possibilities enabled by this approach are worthy of being explored further, figure 29.1 portrays one juxtaposition between the ‘normal’ paradigm and the one made possible by the Barrier. Each rectangle is signifying a ‘desktop’ environment.

29.0.16. Scenario

It is entirely feasible to imagine an output module dedicated *solely* to formatting and displaying a list of all allowed user commands (i.e. the GUI). Creating such a ‘GUI-only’ module is a simple task (registering it with the Barrier, that is! Graphically sound representation of commands with an intuitive layout is a whole different matter). The only thing needed to accomplish it is to take a normal output module and *not* register it for

29. Automatic GUI generation

any BSF at all.² This would mean that the output module will not have any associated thresholds, and will thus never get any BSFs to output, leaving it fully free for its GUI representation. (For it will still get a broadcast of the Catalog, as every output module does. There is no way to avoid getting it, even for output modules that do not even display any commands. Those can, however, just disregard the broadcasts whenever they do get them.)

In the prototype that is provided with this paper, GUI generation does not have a dedicated output module of its own, but is a shared task in the screen output module (cf. section 11.1). It simply shows every action available as a button-like shape, while still reserving enough screen space for ‘normal’ output. It does not make use of the semantic grouping over program lines that is proposed here, but then it does not need to, because such a grouping is now very much possible, while its specifics are for other fields to elaborate on.

29.0.17. Technical description

An important ingredient for enabling output modules to continuously display the current GUI is for them to always have an accurate map of what user commands can be processed by the Barrier. This is ensured by a broadcasting system:

Whenever the Catalog changes, a broadcast action to all output modules registered with the Left Automaton is triggered. This results in each registered (which is not to be confused with installed—registered is identical with ‘up and running’) output module receiving a Barrier-BSF containing a list of strings. The broadcast also takes place every time an output module registers itself with the Barrier.

These strings represent all registered actions, and are in parametrized form. For example, the string “send \$x1 :thing to \$x2 :person” would indicate that the user could input (over various media) the command “send a thank-you note to our supervisor”, which can then be processed by some application that the Barrier will notify.

²Or, alternatively, set the thresholds to be prohibitively high, with the same effect. The other method is, however, more efficient.

30. Periodic commands

30.0.18. Overview

Periodic commands are special commands from the user to the Barrier. The user only has to issue a periodic command *once*, and the Barrier will *memorize* the command and convert it into a regular command, which will continuously be issued after the periodic interval that was specified by the user.

30.0.19. Description

Periodic imperatives contain two parts. For example,

“download e-mails every ten minutes”

is a periodic imperative, which consists of

1. a normal command, i.e. imperative (“download e-mails”), and
2. a time interval (“every ten minutes”).

Since a dedicated timer in the Middle Automaton will generate a ‘normal’ imperative in intervals as indicated by the periodic command, each of these imperatives will have the full authority of the user behind it. Periodic commands serve as a convenience and shortcut method for causing the Barrier to send periodic commands *in the name of the user*. There should be a certain reflection, therefore, before issuing a command such as “download e-mails every five seconds”, which might otherwise drain the system resources unnecessarily.

30.0.20. Technical description

Figure 30.1 depicts the data flow of periodic commands.

Processing periodic commands should be considered with the description in 30.0.19 closely in mind. As can be inferred from figure 30.1, periodic commands arrive in the Barrier’s

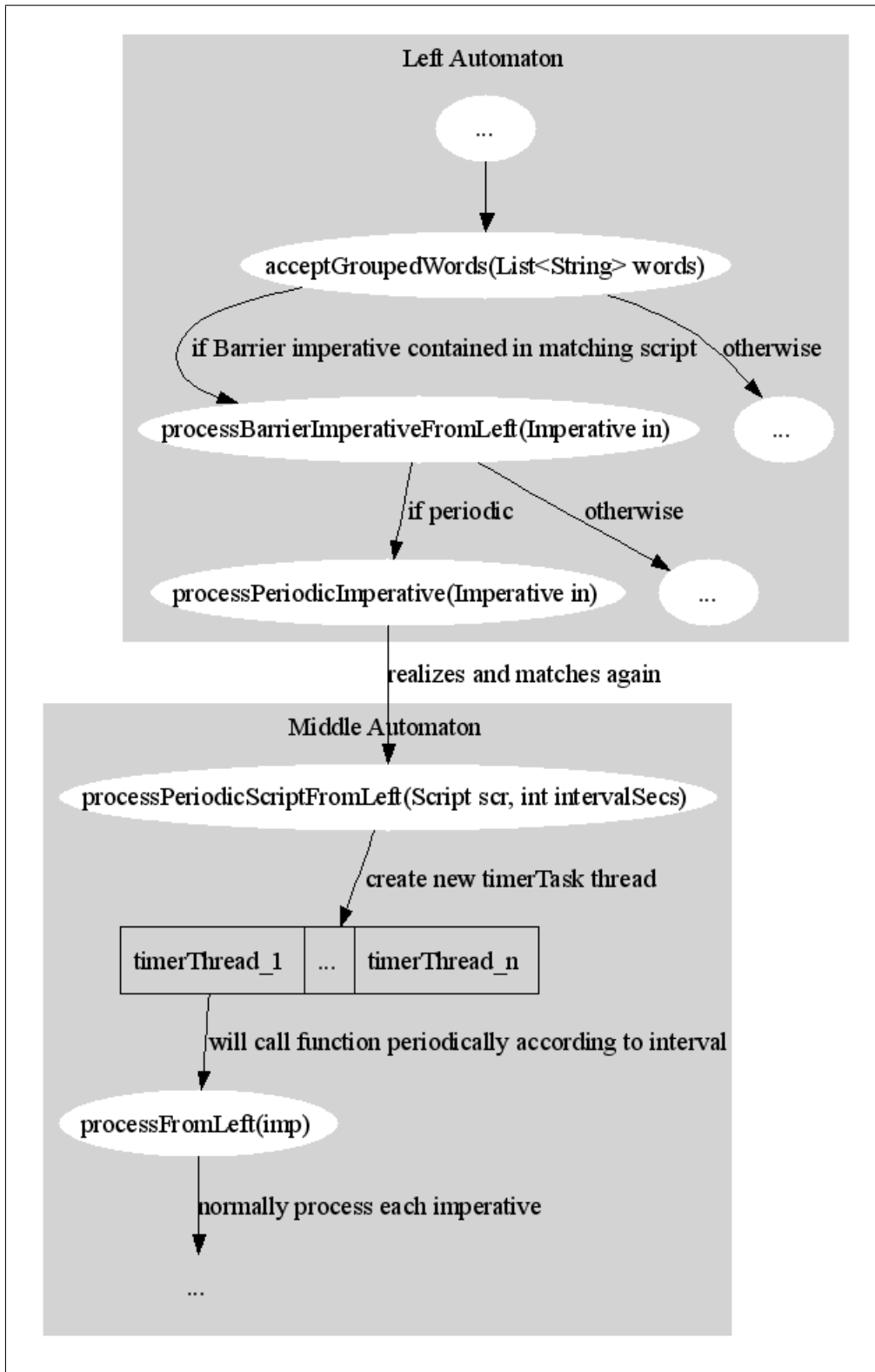


Figure 30.1.: Periodic commands data flow, only the part specific to periodics.

Left Automaton fully analogous to all other commands (as should be the case, since periodic commands *are* commands just the same). They will therefore eventually arrive in the `acceptGroupedWords` function (cf. figure 5.3), which will match them against the Catalog, using transformations according to the Rulebook. From this a script will result in the canonical way. The Catalog is able to match a periodic command iff it can match the normal command it contains. This is because the Catalog, whenever an imperative is registered, also registers a periodic version for that imperative (by extending it with a ‘duration’ part).

Then, *only* if one entry in that script (which consists of commands in imperative form waiting to be executed in order) is a Barrier imperative, and only if that Barrier imperative is a periodic imperative, will execution fork to the `processPeriodicImperative` function for *that specific* imperative. It will be of the form described in section 30.0.19. In the scope of that function, the periodic imperative will be partially realized to extract the interval information that is its second half. (Note: The term ‘realization’, when used in the context of imperatives, means to retransform a parsed string back into its original string form.)

The realized string containing the time interval information will then be converted into the actual time (given in seconds). This is implemented based on a table converting the unit measure into seconds, and the name of numbers being hard coded. Obviously the range of numbers that can be given is hence limited, but that is a constraint that will easily be lifted simply by extending the database of parseable numbers.

The rest of the imperative—the ‘regular’ imperative—will then be *matched again*, and the resulting script along with the extracted time interval information (gained as described in the last paragraph) is passed to the Middle Automaton.

There, a new thread will be created which is responsible for periodically generating that command. It will be controlled by an instance of class `Timer`, which is responsible for controlling all of the threads that are in charge of one individual periodic task each.

An important distinction to make is that at this point, meaning when the `Timer` object awakens the individual thread for an imperative, that thread will call the normal `processFromLeft` function, just as if it had not been a periodic imperative initially at all. From then on, that imperative will be normally processed. Hence, this is the point, as also evident from figure 30.1, at which the periodic command data path *merges back* into the common data path for all commands.

The decision to base that `Timer`-type thread (and with it ultimately the control over periodic imperatives) in the Middle Automaton was made because it is mostly that part of the Barrier which deals with *state*, persistent over arbitrarily many Barrier sessions. For simplicity and conceptual clarity, such state-based information are to be based in the

Middle Automaton.

As an aside, it follows from the above description that *scripts* can contain periodic imperatives. This might be something to be re-evaluated at a future point in time from a design perspective. If, for example, the user gave a fictitious command “update the computer”, it might as a side-effect trigger a script containing periodic commands, without—and this is the point to make—the user being conscious of this. With the Barrier still being in alpha, or early beta stadium, this consideration is, however, too early to be considered a serious alternative or venue of improvement.

30.0.21. Development history, alternatives, and venues of improvement

In the course of development of this feature, both its rationale, and the specifics of its implementation underwent extensive modifications. It was not clear for a long time whether periodic commands would make it into the prototype implementation at all. The decision to then include them after all came about following three considerations:

1. Periodic commands are a highly convenient way for a user to interact with the system, especially since *unlike* ‘normal’ operating systems or programs, they only take the addition of a time interval phrase to *any* command (periodic scheduling at its easiest).
2. Periodic commands, which lead to the Barrier itself periodically generating commands, emphasize well the capabilities and ease of use of the Barrier. If the user can interact with input modules in a natural way, she can expect the Barrier to support that natural interaction to a much larger extent using periodic commands, which, if entered more manually, could through being too repetitive seriously break immersion (i.e. the illusion that one can naturally communicate with any synthetic reasoning system).
3. Most importantly, there was found a way to work periodic commands into the Barrier’s already existing structure with only few modifications (for the major modifications/additions that were necessary, cf. figure 30.1).

One improvement would be to additionally enable the user to *schedule* tasks, i.e., not specify an *interval* of time after which the task is to be reiterated, but rather a *point* in time. As an example, the user could then be able to issue the command “defragment hard drive next Tuesday” (e.g. by saying to the speech input module “defragment next Tuesday”, while making a spinning move with his hand towards the camera, which would signify “hard-drive”). The Barrier prototype with its current capabilities, however, is not able to do that kind of scheduling; the user would not be able to express that which exactly

she wants at the moment. But such an add-on would easily be feasible by implementing a mechanism that is completely analogous to the one described in this chapter, with the one difference that there, a task would not be scheduled for periodic but rather for one single execution.

30. *Periodic commands*

31. Communication among applications

31.0.22. Overview

Although most of how communication *among* applications is handled has already been hinted at in the previous chapters, this serves as a concise summary of the mechanism. Just as users, applications can send imperatives to the Barrier, which will then go through the normal process as if they were input from input modules (from the user). Only when a BSF is received in response from the other application that the request was forwarded to will that answer not be aimed at an output module (to the user), but rather be redirected to the application which posed the request.

This feature may allow for a new paradigm of communication between applications: they need not know each other's API—they need not even know each other at all!—, they simply exchange messages and requests in *natural language!*

This cooperative model described herein—enabling applications to use each other's service without knowing each other—is fully implemented and working in the Barrier prototype.

31.0.23. Description

To enable applications to communicate among each others using the Barrier as a broker, two aspects needed to be functionally implemented:

- Applications need some facility for transmitting imperatives of their own to the Barrier, akin to input modules.
- BSFs from applications need to be distributed to the correct recipient, as now not all BSFs are directed to the user.

The first aspect is solved through a special Barrier-BSF, i.e. a BSF that is meant solely for the Barrier (neither another application nor the user). This Barrier-BSF is of type `imperativeForApplication`, and contain in its data part an imperative of its own, complete with parameters (filled variables). The format of this imperative would be akin to that of the user, namely natural language. The string it contains could as well have been

31. Communication among applications

```
1 <bsfHeader>
2   ...
3   <bsfType> 10 </bsfType>    <!-- 10 = imperativeForApplication -->
4   <isBarrierBsf> true </isBarrierBsf>
5   ...
6 </bsfHeader>
7 <bsfData>
8   <imperativeString>deliver this message to the professor</imperativeString>
9   <rawData>Implementation complete.</rawData>
10 </bsfData>
```

Figure 31.1.: Example of a Barrier-BSF of type `imperativeForApplication`, containing a concrete imperative in its data field.

entered by the user. This could potentially enormously facilitate communication with applications unknown to each other, as they could use natural language that will be parsed by the Barrier, and with the ambiguities erased in the parsing process, still be received in a technical format.

The Barrier-BSF containing an imperative is outlined in figure 31.1.

Given the imperative contained in the Barrier-BSF, the Right Automaton then memorizes from what application the request came from. When the Barrier-BSF arrives in the Left Automaton, that ‘hull’ (the BSF) can therefore be discarded, and the imperative alone fed back into the Left Automaton, just as any imperative coming from input modules (i.e. the user). From there on, no deviations from the normal imperative processing, forwarding, etc. are needed.

Concerning the second aspect, in short, when applications receive a request, they will never know who the requester is. Based on the Barrier’s delocalized architecture, it might be the user, but it might as well be another application running on a physically separated client. Hence, each application only deals with (and has to worry about) the Barrier; it simply sends all its replies to the Barrier, more specifically the Right Automaton. When an application answers a request, the Right Automaton will either start the process of displaying it to the user, or forward it to the requesting application, depending on where the request originated.

31.0.24. Scenario

This example is from the actual Barrier prototype.

Consider two applications, one for controlling a webcam, and one for sending e-mails. The

e-mail application has received an imperative asking it to send a current camera picture to a certain person. That application, however, needs to get a picture to fulfill that task. Therefore, it needs to ask another application for the picture first. Since it does not know any application which can do that, it simply poses the request to the Barrier, using natural language.

Then, without knowing whether there even *is* an application capable of making pictures, or what that application might be, it sends a special Barrier-BSF to the Right Automaton, which contains the simple string “make a picture”. The Barrier, then, before processing comes to entering that string command into its Left Automaton, will memorize in the Right Automaton that it did not come from the user, but from an application. After the string is parsed, the Barrier eventually notices that the camera application is registered for that imperative. Hence, it is given the request.

The camera application, without knowing where the request originated, goes ahead in producing that picture and sends it back to the Right Automaton, marking it as a response to the request. The Right Automaton, in turn, recognizes that the request came from the e-mail application, and forwards the BSF containing the picture there.

Finally, the e-mail application receives the picture it asked for from the Barrier, without knowing where it came from (as that is ultimately of no importance as long as someone *could* process the request). It is then able to send the picture.

In conclusion of that scenario it should be pointed out that in all of this process, the two applications communicating never needed to be aware of each other in any way. If the e-mail application were to be charged with a periodic command, such as “send a picture every ten minutes”, it would not even notice if the camera application were replaced by another one or even if it were transferred to another computer, as long as *any* application was still registered for “make a picture”.

If an application issues a command that is matched not by a single imperative, but by a list of imperatives in a script (a common case), many other applications would be involved with answering that first application’s request, themselves potentially starting additional command cascades. The initial application, when receiving the answer, will never know how extensive the implicit cooperation between applications became just to answer its request. This already implemented and working scheme is not too dissimilar to the multi-Barrier setup described in chapter 32 (which is in contrast to this chapter, however, just a venue of improvement).

31. Communication among applications

32. Multi-Barrier setup

*Eine kleine Barrier die war nicht gern allein,
da lud sie sich zum Osterfest
neun neue Barriers ein.
Ein klein, zwei klein, drei klein, vier klein, fünf klein Barrierlein,
sechs klein, sieben klein, acht klein, neun klein, zehn klein Barrierlein.*

—German child song, lyrics modified

32.0.25. Overview

This chapter showcases just one direction in which the Barrier could be taken. It does not warrant any major changes to the Barrier, but rather has to do with working with the system creatively, and exploiting its capabilities to the fullest extent. This chapter should be treated as an extended ‘venues of improvement’ subsection relating to the concept of the Barrier as a whole, rather than just one part of it.

In a nutshell, this chapter outlines a proposal on how several Barriers could work together, solving tasks for each other that could otherwise not be handled. In a time where applications are still installed and used mostly on single systems, an approach that lets systems make use of each other’s unique capabilities seems very worthwhile, especially if the additional overhead is minimal and the cooperation implicit. It should not be forgotten that in a different realm, that of economic trades, that is exactly the underlying assumption: Different countries with different capabilities all trading their unique goods and services, in a classical win-win situation.

While networked computers such as in the Internet have enabled *users* to gather information regardless of where it is physically stored, an approach as outlined in this chapter would enable *applications* to do the same—if they can use the framework of the Barrier.¹

¹We are not interested at this point to help applications that do not submit to the Barrier.

32. Multi-Barrier setup

Remember how we likened the Barrier to a receptionist in the secretary metaphor of section 1.0.3. To illustrate the idea of a multi-Barrier system within that scope, we would have to think of one secretary getting into contact with the secretary of another company, exchanging the services of their clerks. For example, if a client at the reception desk of a porcelain company wanted to buy an elephant, the secretary would phone her counterpart at Harrods,² which would deliver the elephant to the porcelain company, where the waiting client would pick it up.

The Barrier as just another application

Imagine the following scenario. A Barrier is set up like in figure 32.1 (that setup should come as no surprise by now).

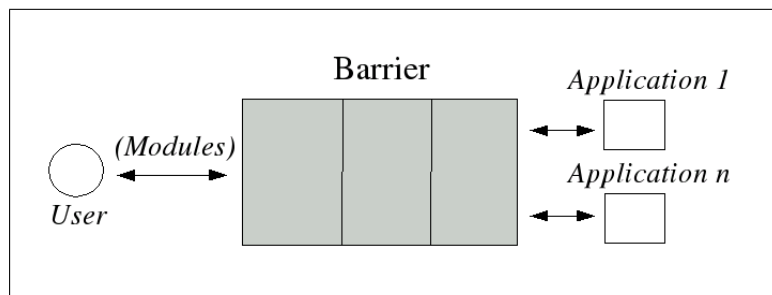


Figure 32.1.: A single Barrier.

Imagine then one application being replaced by a connection to *another Barrier*, as in figure 32.2.

One can see that the Barrier₁ is connected to Barrier₂ just as if Barrier₂ was just another application of Barrier₁. This follows a paradigm of *networking Barriers*.

Many network topologies could be mirrored by Barriers. In figure 32.3, the network takes the form of a 'distributed bus'. In this respect, no further discussion is needed. An analysis about optimal networking shapes with multiple Barriers would certainly be interesting, but such network topology specifics would hardly pertain to this matter. Rather, this analysis would take place when such a setup can actually be created, as that would allow for experimental validation and more specific data about this extension.

32.0.26. Description

Why, it might be asked, could such an approach be useful, and how could it be feasible? The reason, besides the obvious geometric beauty of it, is creating a dynamic *collaboration*

²Purportedly, at Harrods of London one can buy anything.

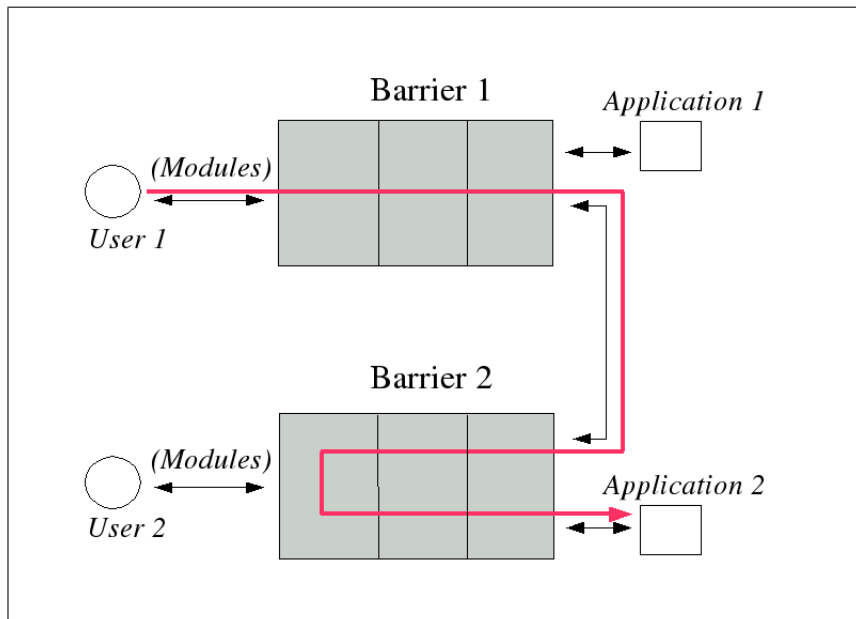


Figure 32.2.: Two Barriers, showing how a user's command is forwarded to an application of another Barrier.

across different system, which is task-dependent. Such a system will make extensive use of the Barrier's capability of *anonymous communication*, i.e. an application answering the request of another application via Barrier without knowing what application that request came from.

Then, with the Barrier registering *as an application* with *another* Barrier (and *vice versa*), the Barriers could pass on requests when

- the Barrier does not have an answer to a command in storage, and for some reason is averse to passing the command to an application (e.g. because the corresponding application has an extremely low capacity), or
- the Barrier has no application registered which is able to process the command. That would require some further modification, since at the moment commands that cannot be matched against the Catalog automatically generate an error message. In that case, parseable commands could also pass further to the right (meaning to the Right Automaton), even without being matched. (Reminder: parse = sentence understood, match = found a corresponding command in the Catalog.)

To actually implement the schema outlined in this subsection, no substantial changes would be necessary, despite the huge ramifications and uses such a system would have. The only major change would be defining a transaction protocol. Even that would still mostly be in the existing framework. As part of the request management, the Barrier already memorizes where a request is going, and, when a result is forthcoming, how to

32. Multi-Barrier setup

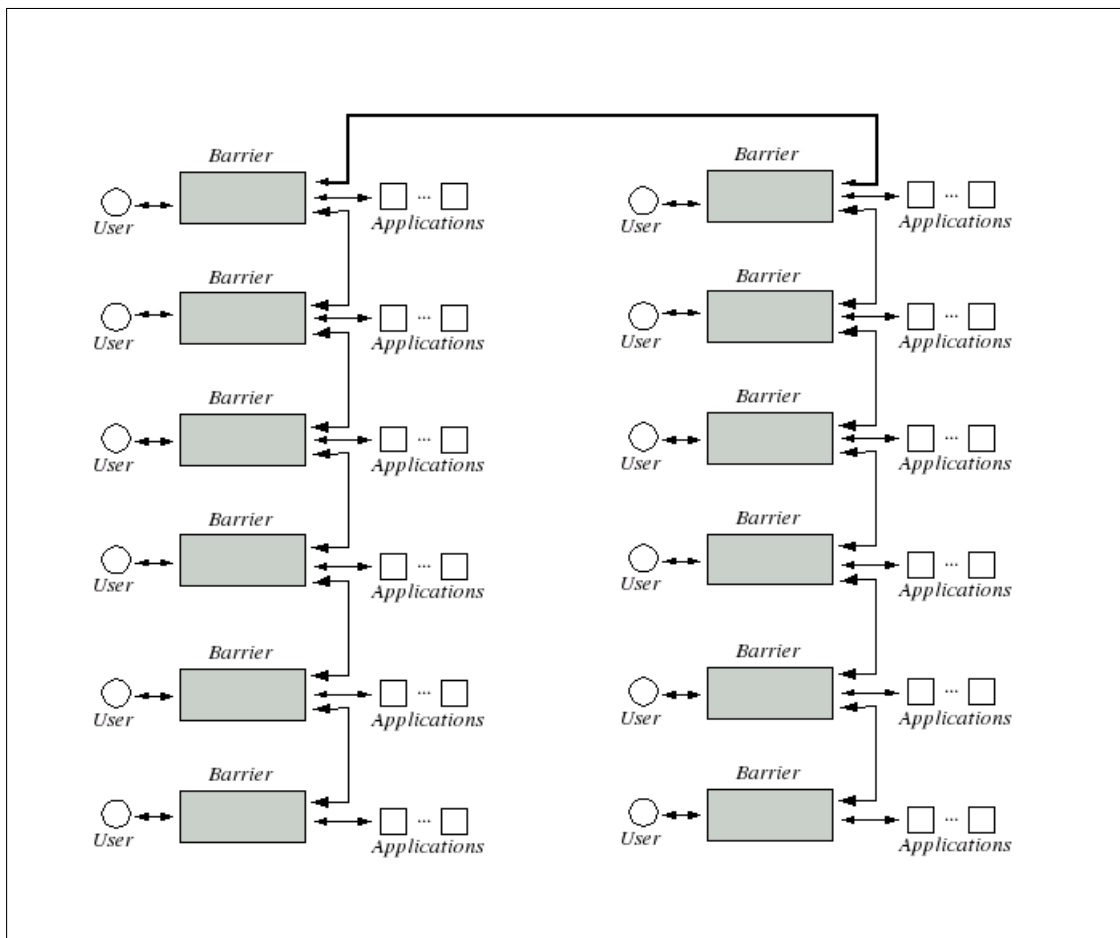


Figure 32.3.: Many Barriers.

distribute it. Only in this case, the requester would be the Barrier itself, and it would send the request to all other Barriers that are registered with it as applications. These could be located anywhere, just as applications can.

Once such a request has arrived at a fellow Barrier, it would again have the choice of forwarding it to other Barriers, not to answer it or to give it to one of its *own* applications for answering, and then forward the response. The initial Barrier that issued the request would then wait for

1. all other Barriers that it forwarded the request to have sent a negative answer, or
2. it receives a positive answer, which is the data it asked for (and could not have procured on its own), or
3. a timeout elapses and the initial Barrier must assume that no answer will be forthcoming.

This scheme would create a collaborative network between Barriers, and a Barrier receiving the answer to a request it could not have processed on its own (having no fitting applications registered) would not even know how many Barriers were involved in finally answering the request. Instead, one should imagine the call stack of a recursive function, where the recursive call is not to itself, but rather a ‘copy’ of itself (i.e. a Barrier on another system), which might eventually contain the ‘base case’ where no further recursive calls are necessary (i.e. having an application which can answer the request, or having the appropriate BSF in its storage), and pass the result back until it reaches the original calling level (the original Barrier).

Of course a user would be free, then, to configure the Barrier such that it will not answer requests from other Barriers (and hence ‘waste’ system resources for other systems), but it would not necessarily be wise to do so, since all things considered resources will be *saved* when other Barriers in turn return the favor and answer requests. In addition to the question of resources, this would enable many additional commands to be answerable even if the system itself does not have the application for answering it installed. It suffices that *another Barrier has them!*

This is similar to ‘semantic web’ philosophy,³ and applications running on a remote server. It seems, however, much more elegant, since the user does not need to explicitly seek out these other resources, but they are sought out by the Barrier, trying to fulfill the user’s request without giving her the trouble of ‘asking’, i.e. going to other servers, herself.

³Specifically, using online applications such as “Google Docs & Spreadsheets”, instead of offline applications such as Microsoft Word.

32.0.27. Scenario

A user, working on a system running Barrier₁, issues the command “show me the current stock market course”.

Unfortunately, while Barrier₁’s grammar can understand that command, it has no clue what to do with it (there is no such entry in its Catalog). Because it does not want to disappoint the user, it wraps the command in a Barrier-BSF and forwards it to the four of its twenty installed applications that it knows to be other Barrier systems running on other computers.

These systems are located in other countries, and running Barrier₂ to Barrier₅. Barrier₂ decides not to answer at all. Barrier₃ is running in China, and its Chinese grammar is not even able to *parse* the command, let alone match it. However, it is kind enough as to send a ‘negative’ back to Barrier₁.

Barrier₄ and Barrier₅ are both able to parse the command, but cannot match it either. (It seems to be the user’s unlucky day.) They do, however, each forward it to other Barriers that *they* are connected to, but which Barrier₁ does not know of.

Eventually, a couple of seconds later (we are talking several GHz systems, after all), Barrier₂₀ finds that it has recently received the answer to just that command, which was issued by its *own* user. Since, luckily, it has that response still cached, it forwards it—not to Barrier₁, since that is not the previous caller on the ‘stack’, but to the Barrier that forwarded Barrier₂₀ that request, which was in this example, say, Barrier₈. Barrier₈ then forwards that answer to Barrier₄, which finally returns the answer to Barrier₁.

Barrier₁, miraculously, does not know of that barrage of communications, but simply is glad that Barrier₄ got it its answer. It then displays that answer to the user.

The user, then, has given the computer a command, and might even think that the system itself has an application that checks current stock exchange courses.

However, it has not.

Note that in such a setup the different computing systems (i.e. the Barriers) need not know any API specifications or the like of each other—they simply communicate in *natural language!* This is in itself a conspicuous advantage, but it has one further implication: When one computing system is unable to process a certain command it can pass the job on to ‘fellow systems’ without even knowing what is required in order to fulfill it! In the above scenario, Barrier₁ does not know what a stock exchange is and that retrieving the course implies connecting to a certain online service; it simply shouts the command out into the outside world *verbatim*, hoping that someone else *does* know how to interpret it.

This emulates the human way of dealing with such problems much more closely than could be possible without the use of natural language.

32.0.28. Technical description

As alluded to, the changes necessary for networking Barriers as described above, are as follows:

- Pass on user utterances that have been parsed successfully, but not matched (contained) in the Catalog (which contains all the locally available commands).
- Same as point one, except also for requests incoming from applications (which could then also be other Barriers).
- Have Barriers that connect to another Barrier as an application also register for a certain command or send a certain message, to tell the Barrier to which they are connecting that they are in fact also Barrier instances.
- Cause the Middle Automaton to forward requests that it has no application for to those applications it knows to be Barriers, using a new Barrier-BSF type, containing parsed user utterances, used for passing them (probably in logical form) to some select ‘applications’ (other Barriers).

The time frame for implementing said changes can be estimated at about five weeks, including QA (Quality Assurance, i.e. testing whether everything still works). However, to make sense, additional time and resources (meaning hardware resources) would have to be invested into creating an appropriate multi-system setup. Only experimentally could the actual validity and usefulness of the approach as outlined in this chapter be verified.

32. *Multi-Barrier setup*

33. Outlook and conclusion

We can only see a short distance ahead, but we can see plenty there that needs to be done.

—Alan Turing, 1950

In retrospect, the Barrier's initial design has proven to be reliable, and the different parts work together seamlessly. Expectations have been surpassed by far. Even the prototype, which, while being a complex endeavor of eleven thousand lines of code in Java alone, has been demonstrated to be quite powerful. Essentially, *all* mechanisms, data structures, algorithms and tools described in this paper have been incorporated and are working in the prototype, except when it was explicitly stated they have not been included.

Obviously, many modularly pluggable parts that we have only implemented in a rudimentary manner need to be exchanged for more sophisticated versions. For example, threshold learning, while implemented in a way that would allow for any kind of adaptation algorithm, uses simple static modifiers for each case. The task would, however, fit perfectly in the realm of machine learning, which offers a panoply of methods that could be tried to tackle the problem in a better way, such as neural or Bayes nets. Such a machine-learning approach could be taken in several other parts as well in order to improve on the Barrier's performance.

In closing

It is the profound wish that development into the Barrier continue. Not because the Barrier prototype at its current stage is not powerful—it is. Precisely because it has surpassed many reasonable expectations its concept should be further explored. Especially the elderly, disabled, but also novices in computer usage could work reasonably easier using this approach.

More than this, we believe that the Barrier could prove to be a starting point towards a

33. Outlook and conclusion



Figure 33.1.: If the gap is too wide for a bridge, build a Barrier as a stepping stone.

Source: [Nie93]

new paradigm: a paradigm in which entities no longer have to follow strict and artificial communication protocols but rather communicate in *natural language*. Such ‘entities’ are, e.g.,

- the users: they shall be able to simply ‘speak their minds’ instead of memorizing shell commands or lengthy paths through graphical menus.
- applications: they shall not be forced to minutiously adhere to inflexible API specifications when using each other’s services. *Applications, too, shall use natural language for interaction!*
- entire computing systems: when a user’s wish cannot be fulfilled by the local computing system, it shall simply pass it on to other, remote computing systems—again in natural language! (Confer the ideas in chapter 32 about a multi-Barrier setup.)

Last but not least, apart from such futuristic visions, there is the pragmatic boon that application programmers need not be bothered by designing any more windows, GUIs, or supporting speech or facial recognition.

This project has been amazing in scope, and without the full support of Prof. Knoll and his chair it would never have reached realization. For that, we owe the group our appreciation.

The prototype is available for download at its Sourceforge location¹ and will hopefully also soon be from the chair’s website.

So let us end the discussion and start what it is all about:

```
public static void main(String[] args) {
    LeftAutomaton la = new LeftAutomaton();
    MiddleAutomaton ma = new MiddleAutomaton();
    RightAutomaton ra = new RightAutomaton();

    la.setMiddleAutomaton(ma);
    ma.setLeftAutomaton(la);
    ma.setRightAutomaton(ra);
    ra.setMiddleAutomaton(ma);

    la.start();
    ma.start();
    ra.start();
}
```

¹<http://sourceforge.net/projects/the-barrier/>

33. *Outlook and conclusion*

User guide

Overview

IMPORTANT NOTICE: THE BARRIER COMES WITH NO WARRANTY. IT IS NOT SUITED FOR CHILDREN UNDER THE AGE OF 5. PARENTS ARE LIABLE FOR DAMAGE THEIR CHILDREN CAUSE. THE BARRIER IS NOT MEANT FOR SWALLOWING, THROWING OR THOSE FAINT OF HEART.

—Excerpt from the *Barrier Terms & Conditions*, page 1324 (forthcoming²)

This is a short guide on how to use the Barrier prototype that this report described. For more information, cf. the Sourceforge project.³

Starting the Barrier prototype

The Barrier distribution contains the Barrier, the two example applications and each of the modules as separate Java archives (jar files, cf. picture 33.2). When the distribution is unpacked, a directory tree containing all of the Barrier's workspace is created. (That folder we shall call the Barrier's home directory.)

To start the Barrier and/or any modules/applications, it suffices to execute the respective jar file; e.g. to start the Barrier itself either type

```
java -jar barrier.jar
```

or double-click the corresponding symbol in a file manager.

It is important to *start the Barrier first*, then all modules can be started at any time to the liking of the user. To start the example application, it suffices to trigger an action

²... not!

³<http://sourceforge.net/projects/the-barrier/>



Figure 33.2.: A jar.

that those will execute. The user will not actually see the example application, but only the Barrier interface that is interacted with.

Installing the Barrier prototype

Before the Barrier can be started, it has to be configured. This serves to make the installation directory known to the Barrier and the applications/modules. Also, the IP addresses of the computer on which the Barrier runs (this may be another, remote machine) and of the local computer must be available.

Configuration works as follows:

1. Customize the file `params.xml` in the Barrier's installation directory by setting the correct values for `BARRIER_HOME` (the directory created upon unpacking of the distribution), `BARRIER_IP`, and `LOCAL_IP`. A sample is shown in figure 33.3.
2. Execute

```
java -jar barrier.jar config <PARAMS_FILE>
```

once in the directory of `barrier.jar`, where `<PARAMS_FILE>` is the location of the `params.xml` file mentioned above.

Using the Barrier prototype

The distribution `jar` file contains several different files. The following lists denote which of these are fit to run on which systems.

The following parts of the distribution are working on *any* operating system:⁴

⁴Referring to either Windows or Unix systems.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
3 <preferences EXTERNAL_XML_VERSION="1.0">
4   <root type="user">
5     <map/>
6     <node name="de">
7       <map/>
8       <node name="tum">
9         <map/>
10        <node name="in">
11          <map/>
12          <node name="barrier">
13            <map>
14              <entry key="BARRIER_HOME" value="/home/fkrull/barrier"/>
15              <entry key="BARRIER_IP" value="65.206.60.120"/>
16              <entry key="LOCAL_IP" value="131.159.77.90"/>
17            </map>
18          </node>
19        </node>
20      </node>
21    </node>
22  </root>
23 </preferences>

```

Figure 33.3.: An example of a configuration file.

- Barrier,
- Bobertino,
- Bmail,
- Clumsyboard input module,
- Vision input module,
- Keyboard input module,
- Screen output module,
- Speech output module.

The following parts of the distribution are *only* working on *Windows* systems and are resource intense, plus requiring at least a free testing license:⁵

- Speech input module.

The following parts of the distribution only make sense on systems with Internet access:

- Bmail.

The following parts of the distribution only make sense on systems connected to a Robertino mobile platform:

- Bobertino.

To *quit* the Barrier correctly, enter the command “exit Barrier” over either the speech input or the keyboard input module. Most modules and applications can be shut down by entering “quit” one of these ways.

Example commands that will trigger an action in the Barrier:

- get new e-mails
- get new e-mails every ten minutes
- write an e-mail
- write to the professor
- drive north
- drive wave
- make a picture

⁵Since they make use of proprietary software, confer section 10.1.

- send a picture to the professor

Bibliography

- [Ake06] D. Akers. Wizard of Oz for participatory design: inventing a gestural interface for 3D selection of neural pathway estimates. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems (Montreal)*, page 454, 2006.
- [All01] J. Allen. *Natural Language Semantics*. Blackwell Publishers, 2001.
- [Bal02] J. Baldridge. *Lexically Specified Derivational Control in Combinatory Categorical Grammar*. PhD thesis, University of Edinburgh, 2002.
- [Bea82] G. Bealer. *Quality and Concept*. Oxford University Press, 1982.
- [Ber96] P. Bernstein. Middleware: A model for distributed services. *Communications of the ACM*, 39(2):86, 1996.
- [BKMW05] H. Bunt, M. Kipp, M. Maybury, and W. Wahlster. Fusion and coordination for multimodal interactive information presentation. In O. Stock, editor, *Multimodal Intelligent Information Presentation*, pages 325–340. Springer, 2005.
- [BKW06] C. Bozşahin, G.-J. M. Kruijff, and M. White. *Specifying Grammars for OpenCCG: A Rough Guide*. Included in the OpenCCG distribution, 2006.
- [CMBT02] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. Gate: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.
- [CMN83] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.
- [CR01] J. M. Carroll and M. B. Rosson. *Usability Engineering: Scenario-Based Development of Human-Computer Interaction*. Morgan Kaufmann, 2001.
- [DFAB93] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human Computer Interaction*. Prentice Hall, 1993.

Bibliography

- [DG02] J. Dorn and G. Gottlob. Künstliche Intelligenz. In P. Rechenberg and G. Pomberger, editors, *Informatik-Handbuch*, pages 983–1008. Hanser, 2002.
- [FBRK06] M. Foster, T. By, M. Rickert, and A. Knoll. Human-robot dialogue for joint construction tasks. *Proc. Eighth International Conference on Multimodal Interfaces (ICMI 2006), Banff, Alberta, 2006*.
- [Fel98] C. Fellbaum. *Wordnet: An Electronic Lexical Database*. Bradford Books, 1998.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [Gre06] A. Greenfield. *Everyware: The Dawning Age of Ubiquitous Computing*. New Riders Press, 2006.
- [Hau01] R. Hausser. *Foundations of Computational Linguistics: Human-Computer Communication in Natural Language*. Springer, 2001.
- [Hic52] W. E. Hick. On the rate of gain of information. *Quarterly Journal of Experimental Psychology*, (4):11–26, 1952.
- [JMK⁺99] D. Jurafsky, J. Martin, A. Kehler, K. Linden, and N. Ward. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, 1999.
- [Kel83] J. F. Kelley. An empirical methodology for writing user-friendly natural language computer applications. *Proceedings of ACM SIG-CHI '83 Human Factors in Computing Systems (Boston)*, page 193, 1983.
- [KG01] A. Knoll and I. Glöckner. A basic system for multimodal robot instruction. *Proc. 5th SIG-Dial Workshop on Formal Semantics and Pragmatics of Dialogue, Bielefeld*, pages 241–250, 2001.
- [KRSR03] A. Kay, D. Reed, D. Smith, and A. Raab. Croquet: A collaboration system architecture. *First Conference on Creating, Connecting, and Collaborating through Computing (Stanford)*, page 2, 2003.
- [KvGR] H. Kamp, J. van Genabith, and U. Reyle. *Draft of an article for the new edition of the Handbook of Philosophical Logic*. <http://www.ims.uni-stuttgart.de/~hans/hpl-drt.ps>.
- [LS02] U. Lang and R. Schreiner. *Developing Secure Distributed Systems with CORBA*. Artech House, 2002.
- [Min06] M. Minsky. *The Emotion Machine*. Simon & Schuster, 2006.

- [MW98] M. Maybury and W. Wahlster, editors. *Readings in Intelligent User Interfaces*. Morgan Kaufman, 1998.
- [Nie93] J. Nielsen. *Usability Engineering*. Academic Press, 1993.
- [Ols98] D. Olsen. *Developing User Interfaces*. Morgan Kaufmann, 1998.
- [Pal99] S. E. Palmer. *Vision Science: From Photons to Phenomenology*. Bradford Books/MIT Press, 1999.
- [Pap93] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.
- [Ras00] J. Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison Wesley, 2000.
- [RN03] S. Russell and P. Norvig. *Artificial Intelligence. A Modern Approach*. Pearson Education, second edition, 2003.
- [Shn80] B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop, 1980.
- [Sie00] J. Siegel. *CORBA 3: Fundamentals and Programming*. Wiley, 2000.
- [SP04] B. Shneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison Wesley, 2004.
- [Ste87] M. Steedman. Combinatory grammars and parasitic gaps. *Natural Language & Linguistic Theory*, pages 404–439, 1987.
- [Ste96] M. Steedman. *A Very Short Introduction to CCG*. <ftp://ftp.cis.upenn.edu/pub/steedman/ccg/ccgintro.ps.gz>, 1996.

Bibliography

Glossary

Application	A program that uses the Barrier for interaction with the user and among each other. It is connected to the Right Automaton, 205
Arbitration	The process of choosing one application for executing a user command when several applications capable of doing the job exist, 193
Automaton	Each of the three spheres of the Barrier is organized into an 'Automaton'. Note: 'Automaton' is not to be understood literally in the meaning of 'defined as a finite state machine or Turing machine etc.', 19
Barrier imperative	A special imperative that is directed not at an application, but at the Barrier itself, 71
Barrier-BSF	Special kinds of BSF which are not meant for user nor application, but are meant solely for the Barrier itself. Alternatively, special BSFs which the Barrier created and sends to applications or modules, 144
BSF	Short for 'Barrier Structured Format'. The data format that the Barrier uses for all communication with the application, and frequently also within it, 139
BSF storage	Data structure in the Middle Automaton saving BSFs that did not pass to the user. Used as material for unification of BSFs, 161
Catalog (left)	The Left Automaton's registry of what commands are valid, i.e. of what services are offered by the overall computing systems. Does not contain information about which applications are registered for them, 74

Catalog (right)	The Right Automaton's registry of which applications are registered for which commands, 183
Goal Test	An algorithm that takes a user utterance and a Catalog entry as input to check whether the former matches the latter. The two need not be identical because the Catalog entry may contain variables, 74
Grouper	The part of the Left Automaton that decides on which input snippets (arriving from different input modules) are considered to form one coherent user command. The Grouper comes first in the chain of multimodal input processing, 37
Imperative	Once input from the user is parsed, this is the data structure holding that information. Used only within the Barrier. We call an imperative that is produced from a user utterance a 'concrete imperative', 69
Input module	Module which specializes on transforming user input into a string, which is then passed to the Left Automaton. Normally controls one specific media, 117
Left Automaton	The Automaton connected to the input modules, and therefore closest to the environment, i.e. the user, 27
Matcher	The algorithm that checks whether a user command can be executed by using the services stored in the Catalog. If so, it returns the script that will do the job, 77
Merger	The part of the Left Automaton that comes second in the input processing chain and that operates on the Grouper's output. It puts the unordered input snippets that form one user command into order so they resemble a natural-language sentence, 41
Middle Automaton	The Automaton that is in between the Left and the Right Automaton. Responsible for most of the Barrier's 'state', 153

Mode	There are two modes of inputting data into the Barrier: command mode and data mode. In data mode raw (i.e. unparsed) data is entered that serves as a parameter to the command previously entered in command mode. Switching between the two is done automatically, 105
Module	Special kind of application which can also communicate with the Left Automaton. There are two kinds: input and output modules, 117
Output module	Module which specializes on outputting certain types of BSFs on the media it controls, 117
Periodic command	A user command that is to be executed not only once but rather in periodic intervals; e.g. “air the room every two hours”, 235
Realizer	Produces a natural-language sentence from a logical form. It inverts the parser’s work of producing a logical form from a natural-language sentence, 49
Request	A user command; thus equal to an imperative. We speak of an ‘open request’ when it has not been processed by an application yet. The ‘request history’ is an all-time storage of what requests have been answered by what applications with what BSFs; it helps the Barrier learn how to answer requests automatically, 157
Right Automaton	The Automaton that is connected to the applications and handles all interactions with them, 179
Rulebook	The set of rules used by the Matcher algorithm to transform an imperative into another one with the same semantics but different syntax. The goal of such transformations is to make a user utterance match a Catalog entry, 88
Script	An ordered list of imperatives, 71

Glossary

Threshold	Value which denotes how likely a specific BSF is to be output by a specific media. Dependent on technical affordances and user preferences. When a BSF's priority is beyond its threshold for a media, it may legally be displayed on it, 169
Unification	Process in which several BSFs are recombined into a new BSF, 165

Index

- A* search, 86
- acceptGroupedWords, 37, **38**, 108, 109, 237
- alternating Turing machine, 81
- ambiguity, **46**, 52
- anaphora, 47, 51
- application, 21, 89, 111, 180, 183, 193, 197, **205**
 - Bmail, **211**
 - Bobertino, 131, **209**
 - capacity, 188
 - communication, **197**
 - failure, 201
 - installing, 189
 - management, **187**
 - name, 187
 - registering, 184, 189, 205
 - registry, 188
 - start command, 187
 - starting, 190
 - uninstalling, 189
 - unregistering, 184, 189
- apply, 103
- applyRec, 103
- arbitration, 180, 188, 191, **193**
- Archy, 13
- attenuator theory, 40
- Automaton, **19**
- Barrier
 - configuration, **217**
 - general setup, **19**
 - performance, **217**
- Barrier imperative, *see* imperative
- Barrier Structured Format, *see* BSF
- Barrier-BSF, *see* BSF
- Base 64, 135, 143, 211
- Bayes net, 122, 175
- beneficiary, 64
- Bmail, *see* application
- Bobertino, *see* application
- BSF, 21, 136, **139**, 157, 169, 197, 241
 - Barrier-BSF, 71, **144**, 189, 197, 234, 241
 - composite, 165
 - data, 141
 - header, 139
 - priority, 169
 - storage, 154, **161**
 - supertype, 170
 - unification, 155, 161, **165**
- camera
 - latency, 128
 - module, *see* module
- capacity, 193
- casemark, 64
- Catalog, 48, 71, **76**, 79, 88
 - broadcast, 113, 131, 234
 - entry, 76

Index

- Right Catalog, 76, 180, **183**
- category, 56, 63
 - atomic, 57, 67
 - complex, 57, 67
- CCG, **56**
- Chomsky hierarchy, 56
- Cloudgarden TalkingJava, 120
- Clumsyboard, *see* module
- color space, 129
- Combinatory Categorical Grammar, *see* CCG
- common sense, 52
- communication
 - among applications, **241**
 - channels, 29
 - protocol, 208, 224
 - setup, **221**
 - with applications, **197**
- complex category, 67
- contextualization, 51
- CORBA, 13, 221
- Croquet Project, 15
- cycle, 79
- depth-first search, 80, 86
- derivability, 74, 91
- design goals, 4
- Discourse Representation Theory, 51
- ditransitive verb, 63
- DOM, 72
- Dragon NaturallySpeaking, 119
- DRT, *see* Discourse Representation Theory
- DTD, 144
- e-mail client, 211
- entropy, 42
- feature structure, 58, 63, 67
- finite state machine, 127, 212
- FreeTTS, 136
- front-end approach, 31
- generalization, 74, 75, 78
- gesture, 29
- Goal Test, 48, **74**, 79, 91
- GOMS, 16
- grammar, 47, **52**
 - formal, 56, 91
- graph, 80, 88
- Grouper, 34, 39
- GUI, *see* user interface
- gulf of evaluation, 13
- heuristic
 - admissible, 86
- Hick's law, 15
- histogram, 129
- HSI, 129
- hypernym, 92
- imperative, 48, 59, **70**, 74, 105, 157, 180, 183, 193, 198, 205, 235, 241
 - Barrier imperative, **71**
 - ontology, 185
- ImperativeParser, 65
- inconsistency, 85
- incremental approach, 31
- input snippet, 33
- JAST project, 31
- Java, 72, 84, 120, 190, 221, 227
 - JavaMail, 213
 - naming convention, 187
 - Native Interface, 211
 - Speech API, 120
- JDOM, 72, 77
- JPEG, 131, 211
- keyboard module, *see* module
- label table, 75, 84, 91, 199
- lambda calculus, 56

- learning, 40, 43, 86, 123, 172, 194
- Left Automaton, 19, **27**
- lexical family, 58, 61, 66
- logical form, 58, 63, 67, 72, 74, 91, 183, 199
- Markov model, 122
- Matcher, 35, 49, 70, **78**, 101, 227
 - match condition, 36
- maximum likelihood, 122
- Merger, 34, 41
- Middle Automaton, **153**
- middleware, 13
- mild context sensitivity, 56
- mimics, 29
- mode
 - command mode, 105
 - data mode, 105
- module, **117**
 - input, 19, 105, 112, **119**
 - camera, *see* vision
 - Clumsyboard, **121**
 - keyboard, 33, **126**
 - speech, **119**
 - vision, 33, **127**
 - management, 111
 - output, 19, 72, 112, **131**, 169, 231
 - screen, **131**, 211
 - speech, **135**
 - starting, 112
 - transfer of output, 114
- multi-Barrier system, 163, 243, **245**
- multimodal communication, **29**
- natural language, 77, 183, 199, 241
 - processing, **45**
 - software, 53
- neural net, 86, 172, 194
- noun, 66
- ontology, 60, 68
- OpenCCG, 47, **57**, 65, 185, 199
- OpenNLP, 57
- parse tree, 58
- Parser, 35, 47, 52, 65, 183
- part of speech, 58, 63, 66
- path cost, 79
- patient, 64
- pattern, 122
- periodic command, 64, 71, **235**, 243
- permutation, 37, 42
- permute, 43
- processFromLeft, 22, 37, 224, 237
- processFromRight, 22, 224
- Project Looking Glass, 14
- quantifier alternation, 81
- question, 59
- raw data, 70, 105
- realizer, 49, 52, 73, 113, 199, 237
- relative clause, 69
- request
 - history, 158
 - management, **157**
- RGB, 129
- Right Automaton, 21, **179**
- Robertino, 209
- Robotino, 209
- rule, **90**, *see* Rulebook
 - anti-decomposition, 94
 - application, 91
 - decomposition, 90, 93, 107
 - translation, 92
 - inter-language, 94
- Rulebook, 49, 70, 78, **88**, 227
 - dynamic extension, 93
- SAX, 72
- screen module, *see* module
- script, 48, 71

Index

- matching script, 82
- search problem, 78
- secretary metaphor, **6**, 245
- semantics, 35, 63, 68
- semaphore, 190
- sender, 222
- serialization, **227**
- server, 23, 222
- slash, 67
- socket, 22, 188, 221
 - handler, 224
- speech input module, *see* module
- speech output module, *see* module
- speech recognition, 119, 218
- speech synthesis, 135
- Sphinx, 120
- state
 - in search, 78
 - repeated, 80, 84
- stem, 58
- stylesheet, *see* XSL
- synonym, 92
- syntax, 35, 63, 68
- task-oriented functional design, 14
- thread
 - architecture, 23
 - server thread, 222
 - synchronization, 107
- threshold, 153, **169**
 - adaptation, 173
 - initial, 113
 - management, **173**
- Timer, 237
- transitive verbs, 67
- transposition, 42
- Turing completeness, 56
- type, 58, 60, 66, 75
 - conversion, 63
 - hierarchy, 60
- ubiquitous computing, 222
- UML, 118, 134, 188, 207, 223
- user interface, 13
 - automatic generation, 4, **231**
 - command, 72
 - graphical, 131
 - semantic, 133, 232
- user preferences, 153, 173
- variable, 59, 64, 74, 75, 77, 90, 199
 - label, 60, 74
 - table, *see* label table
 - type, 60
- vision module, *see* module
- visitor, 77
- vocabulary, 66, 68, 191
- webcam, 127, 209, 242
- Where is Godot?, *He's on page 59*
- Wizard of Oz experiment, 17
- word, 34
- XML, 58, 65, 72, 136, 143, 167, 229
 - Schema, *see* XSD
 - validation, 181
- XSD, 143, 181
- XSL, 72, 85, 167, 200