Universität des Saarlandes

Master's Thesis

A System for Rapid Development of Large-Scale Rule Bases for Template-Based NLG for Conversational Agents

Submitted in partial fulfilment of the requirements for the degree
Master of Science (MSc) in Language Science and Technology

Author:
Tim Philipp Jeydon Krones
t.krones@coli.uni-saarland.de

Supervisors:
Prof. Dr. Stephan Busemann
Dr. ing. Ivana Kruijff-Korbayová

November 27, 2014

Declaration

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Declaration

I hereby confirm that the thesis presented here is my own work, with all assistance acknowledged.

Tim Krones
Saarbrücken, November 27, 2014

Acknowledgments

I'd like to thank my supervisors, Dr. ing. Ivana Kruijff-Korbayová and Prof. Dr. Stephan Busemann, for giving me the opportunity to work on a very interesting project, for providing guidance and invaluable feedback along the way, and for giving me freedom to explore and implement many of my own ideas.

I am grateful to Bobbye Pernice for providing non-technical advice and guidance at various points throughout the time I was working on this project.

My sincerest thanks go to my family for their unconditional support, their words of encouragement, and for their unshakable belief in my abilities.

I am also grateful to Brigitte Krones and Nic Kramer for sharing stories about the final stages of their studies. You helped me push on at a time when I was ready to quit.

Finally, I'd like to thank Anna Schmidt for feedback and fruitful discussions, and for being by my side throughout this journey.

Contents

Part I

Introduction

1 Motivation

Long-term social interactions between conversational agents and users should feel as natural as possible to users. This requires adaptivity and variation of system output [34, 35]: When communicating with humans using spoken language, agents should produce natural language output that is appropriate for and relevant to the current situation once it is their turn to speak. Thus, the choice of output to produce depends on what an agent knows about the current situation and the dialogue context. For instance, when formulating a greeting at the beginning of an encounter, an agent needs to take into account if it is meeting a user for the first time or if it has interacted with that particular user before. Additionally, agents should not repeat themselves in recurring situations involving the same user(s), which means that they need to be able to choose from a range of different verbalizations for the content to deliver. Sticking with the example of greeting a user, on the first encounter with an unfamiliar user an agent might say:

Hello, I am pleased to meet you.

On subsequent encounters, the agent might choose from any of the following verbalizations to greet the now-familiar user:

Hello, good to see you again.
Hi, good to see you again.
Hello, I am happy to see you again.
Hi, I am happy to see you again.
Hello, it's so good to see you again!
Hi, it's so good to see you again!
...

ALIZ-E[1] is one example of a project that focuses on developing conversational agents for long-term social interactions [44]. In the context of this project, which was carried out jointly by the German Research Center for Artificial Intelligence (DFKI)[2] and a number of European partners, conversational agents were used to provide companionship and support to diabetic children who needed to learn how to manage their condition themselves [36].

In the following, we will describe how the task of designing natural language output is currently handled in the ALIZ-E project. We will then discuss a number of factors that make the present approach challenging. Finally, we will outline how the system presented in this work addresses these challenges in order to facilitate development of natural language output for conversational agents in the context of the ALIZ-E project.

In ALIZ-E, the task of designing natural language output involves writing *rules* which match abstract representations of knowledge that a conversational agent has about a given situation to appropriate verbal responses. More specifically, abstract representations of knowledge consist of *features* and associated *values* that capture the dialogue situation and the content to communicate. Rules modify these representations and assign appropriate verbalizations to them. During on-line processing, a specialized *rewriting engine* [34, 31] chooses rules to apply by comparing collections of features and values representing an agent's current knowledge against the situational knowledge that each rule presumes. After

[1] `http://www.aliz-e.org`
[2] `http://www.dfki.de/`

determining and applying matching rules, a Text-to-Speech system takes care of realizing natural language output defined by these rules. Rules are stored in plain text files and must be written in a specialized syntax that the rewriting engine can understand. For example, the following rule produces one of the outputs listed above every time it is applied:

```
:dvp ^ <SpeechAct>greeting
     ^ <Context>(<Familiarity>yes ^ <Encounter>notfirst)
->
###greeting = random("Hello, ", "Hi, "),
###content = random("good to see you again.",
                    "I am happy to see you again.",
                    "it’s so good to see you again"),
# ^ :canned ^ <stringOutput>concatenate(###greet, ###content)
  ^ <SpeechModus>indicative.
```

Although moving to grammar-based natural language generation (NLG) (cf. [52]) is a long-term goal in ALIZ-E, the generation approach that is currently in use is template-based: Sets of alternative verbalizations associated with individual rules are specified in the form of *canned text*. If necessary, canned text may contain variables that are replaced with appropriate, context-dependent values at run-time to produce the final output that will be uttered by an agent. With this approach, creating large amounts of variation for individual rules can be time-consuming, as alternative verbalizations belonging to individual rules need to be specified manually. Aside from being time-consuming, this process can also be quite tedious because within rules, verbalizations might be similar to a large extent, requiring rule developers to type the same content over and over again. Another issue is that as the number of alternative verbalizations available for a given rule increases, keeping track of ways to combine parts of existing verbalizations to create even more output becomes increasingly difficult. As a result, rule developers might accidentally introduce duplicate output or omit verbalizations that would have been appropriate for the situation covered by a given rule.

Issues related to creating large numbers of verbalizations are mitigated to a certain degree by the fact that the syntax for defining rules provides constructs for reducing the amount of duplicate content that needs to be specified manually for individual rules (cf. Chapter 8). However, without support for visualizing full sets of alternative verbalizations for individual rules, using these constructs can lead to errors resulting from invalid combinations of substrings. Additionally, while using these constructs can reduce the number of full strings that have to be specified manually when defining the output of a single rule, the problem of having to specify the same or very similar content from scratch for the output of other rules addressing similar situations remains. Since rules are specified using plain text, it is of course possible to transfer content to other rules by copying it. Depending on the amount of content being transferred, however, this approach can be error-prone (in the sense that it can lead to unwanted duplication across rules).

Aside from challenges related to creating large amounts of variation, there are a number of additional issues concerning creation, maintenance, and evolution of large-scale rule bases that could benefit from specialized support. For instance, locating specific rules and checking rule coverage for specific scenarios in order to identify what to work on next are common tasks that become increasingly difficult to perform as rule bases grow. Additionally, if rule bases are large, errors resulting from inconsistent use of features and values for representing situational knowledge are more likely to occur without support for detecting them or preventing them from occurring in the first place. Lastly, the process of defining rules in native rule syntax lacks a standardized way of documenting the purpose of specific rules. This can complicate the process of locating relevant information both for original authors of specific rules and for potential collaborators.

Another aspect that is important for rapid and successful development of large-scale rule bases is efficient collaboration. To a certain extent, the process of collaboratively editing a set of rule files can be streamlined by using a version control system. But even then a certain amount of overhead is necessary to ensure that changes are propagated to other developers as soon as possible, and to resolve potential conflicts.

In addition to the challenges concerning specific aspects of rule design described above, there is another issue we would like to mention: Creating rules in native rule syntax can be difficult for people who lack background knowledge in relevant subject areas such as (computational) linguistics and computer science. In order to be able to productively create and edit rules, rule developers must be aware of the kinds of knowledge agents can have about their environment, and how this knowledge is represented internally. Furthermore, they must have at least a basic understanding of how rules operate, and how appropriate rules are chosen during on-line processing. As a result, enabling people without any relevant background knowledge to productively contribute to rule development might take a substantial amount of training[3]. This is a problem because natural language output for intelligent agents might be needed in a variety of domains. On the one hand, we can not expect people who are experts in these domains to also have background knowledge in the subject areas mentioned above. On the other hand, people who are qualified to write rules might lack knowledge relevant to domains in which intelligent agents are to be used. Reducing the amount of background knowledge required for working with rules (so as to enable domain experts to contribute more easily to rule development) would therefore be desirable.

In order to address the issues described above, we developed a graphical system for designing natural language output for conversational agents that supports creation, maintenance, and long-term evolution of large rule bases in the following way: First of all, specialized editing features for rule output allow for quick creation of large numbers of alternative verbalizations for individual rules. By facilitating reuse of existing material, the amount of typing involved in creating output alternatives is reduced. Secondly, by abstracting away from native rule syntax as much as possible, the system makes the process of creating rules more accessible to people without necessary background knowledge. Minimally, the system allows tasks involved in creating rules to be distributed according to areas of expertise of individual collaborators. Results from a first study involving subjects without a background in (computational) linguistics or computer science provides support for this claim. By having users work with graphical representations of rules, the system also reduces the potential for errors that are likely to happen when creating rules manually, and eliminates some classes of errors entirely. Third, our system aims to facilitate long-term evolution and maintenance of large rule bases by allowing users to store meta-information about rules, which enables them to, e.g., communicate the purpose of a specific rule to their collaborators. Users can also get a compact overview of all existing rules, and filter existing rules in various ways to locate specific rules or check scenario coverage. Lastly, in order to facilitate collaboration, we chose to implement our system as a web application instead of a desktop application. This allows for real-time collaboration on a given rule base and eliminates the overhead involved in exchanging rule files between developers.

[3]In environments that make use of a version control system to share rule files between developers, non-experts would additionally have to be trained in using that particular technology.

2 Structure

The remainder of this thesis is structured as follows: Part II discusses related work. In particular, Chapter 3 presents graphical development environments for general-purpose NLG, and Chapter 4 describes a number of tools supporting output design for Dialogue systems. Part II ends with a short discussion of how our system relates to the tools presented in chapters 3 and 4. As mentioned before, our system was implemented as a web application to facilitate collaboration between rule developers. Part III describes server-side and client-side architectures of the application and provides information about the technologies that were used to implement different components of the system. It also discusses how our system integrates with existing tools developed in the context of the ALIZ-E project. In Part IV we present the data models that were designed for the purpose of storing and operating on rules: We start by providing information about relevant aspects of native rule syntax in Chapter 8. Chapter 9 then describes the server-side data model in detail, and Chapter 10 provides basic information about client-side data models. (An in-depth discussion of client-side data models is deferred to Appendix A). Part V provides detailed information about user-facing functionality of our system. A specialized interface for defining building blocks for representing situational knowledge is presented in Chapter 12. Functionality for creating and editing rules is described in Chapter 13. Chapter 14 presents functionality for browsing existing rules and filtering them in various ways. It also describes how users can export sets of rules to native rule syntax. Chapter 15 explains how the system can be used to conduct fine-grained searches for rules based on situational knowledge to which they apply and natural output they produce. As mentioned above, we conducted a small study to evaluate our system with respect to making the process of writing rules more accessible to people lacking necessary background knowledge. This study is described in Part VI. Chapter 16 describes the setup of the evaluation experiments. Chapters 17 and 18 present and discuss the results we obtained, respectively. We conclude our discussion in Part VII: Chapter 19 summarizes the main points from the previous chapters and Chapter 20 gives an overview of challenges that will have to be addressed in future work.

Part II

Related Work

3 Development Environments for General-Purpose NLG

In the past, a number of projects in the area of natural language generation have yielded graphical tools for creating, editing, and managing output corpora. In this chapter we describe two full-featured systems developed in the context of general-purpose Natural Language Generation. Tools supporting output design for Dialogue Systems are presented in Chapter 4. Lastly, we highlight how existing applications differ from the system we developed.

3.1 (J)YAG and IDEY

YAG (Yet Another Generator) [10, 11] is a "real-time, general-purpose template-based generation system" [39, 12] written in Lisp. JYAG is a Java implementation of YAG. Templates for YAG must be written in a custom declarative template specification language. JYAG is designed to work with templates in XML format. Each template consists of a number of *template slots* and a number of *template rules*. Template slots are parameters that applications or users can fill with values at run-time. Template rules define how inputs to templates should be translated into text [39, 56].

IDEY (Integrated Development Environment for YAG) [12, 56] is a graphical development environment for (J)YAG templates. It was created for a similar purpose as the system we present in this work: When distributing their system to other researchers, the creators of YAG found that these researchers were able to install and use YAG from other applications successfully, but had difficulty defining new templates. These experiences led them to design and implement a graphical editing environment for YAG templates [56].

In order to facilitate tasks commonly involved in dealing with templates, IDEY provides functionality for authoring, testing, and managing them [12]. Individual templates can be visualized in different ways (see below). According to the authors, template visualization reduces the amount of time required for users to become familiar with template syntax. By constraining template construction and modification appropriately, IDEY also aims to prevent errors which are likely to be introduced when editing plain text representations of templates.

IDEY's user interface consists of two main components called *Project View* and *Template View* [56]. The Project View allows users to browse *resources* available to the current project. Resources include lexicons, morphological functions for inflecting verbs according to features such as tense, person, and aspect, and template libraries. When a resource is selected, its contents are shown below the list of available resources. The Template View consists of two tabbed subsections providing functionality for creating and editing templates, as well as for navigating and visualizing them in different ways. For instance, template rules can be manipulated via the *Rule tab* which shows individual rules in a hierarchical, tree-like structure whose nodes can be expanded and collapsed. The *View tab* shows plain text representations of rules that are currently displayed in the Rule tab. To test a given template, users can specify input values by entering them into a feature structure displayed in the *Test tab*. The Test tab then shows the text that would be generated from the current input [56].

As mentioned above, YAG and JYAG use different formats for templates. IDEY understands both of these formats and can be used to convert between them, which makes it possible to use templates created for YAG with JYAG and vice versa [56].

3.2 TG/2, XtraGen, and eGram

TG/2 [5, 8] is an NLG system written in Common Lisp that has been incorporated into a number of NLG applications ranging from shallow template systems to in-depth realization engines [7]. Generation grammars for TG/2 are created by defining sets of *condition-action rules* with a context-free categorial backbone [7]. These rules define input conditions in the form of test predicates and are used to map content representations matching these conditions onto chains of terminal elements – which may consist of canned text – to generate desired output. Agreement relations between different elements of the derivation tree are established by means of a constraint propagation mechanism [7]: Rules can be annotated with equations asserting equality of feature values at different constituents [5]. XtraGen [50] is a Java implementation of TG/2 that uses XML to encode grammar objects [6, 7].

eGram [6], a full-featured graphical environment for grammar development, supports both TG/2 and XtraGen. It was introduced to enable development of large-scale generation grammars, and abstracts away from different formats used by the TG/2 and XtraGen NLG systems by exposing a universal, developer-friendly grammar format to end users [7]. Consistency issues which arise from creating and editing grammars manually[4] are addressed in eGram by enforcing that basic building blocks for rules – such as test predicates for input conditions and constraints determining agreement relations – be defined before new rules using them can be added to the grammar. Additionally, for some elements the eGram GUI offers context-sensitive editing menus that are created dynamically to include only those options that are appropriate for the element currently being edited. For instance, menus for defining constraints list existing features (such as `CASE`, `NUMBER`, and `PERSON`), and menus for selected features list appropriate values (such as `nom` and `acc`). According to [7], by using dynamically created menus and moving from basic to more complex elements when defining rules, errors are minimized and definitions of individual elements are guaranteed to by syntactically complete.

To speed up creation of similar elements, eGram makes it possible to reuse existing elements for editing by providing options for saving their definitions without closing associated windows. Additionally, eGram provides functionality for deriving additional grammar rules by means of *meta-rules* [46]. The meta-rule mechanism is convenient if grammar coverage needs to be extended to include more specialized linguistic phenomena: Due to the fact that grammars defined using eGram have a context-free backbone, some phenomena such as word order variation, pronominalization, and voice require a large number of additional rules for handling them. If it were not for the option to derive additional rules automatically, these rules would have to be defined manually by the grammar developer [6].

Lastly, for the purpose of testing generation grammars, eGram integrates with TG/2 via a client-server interface [6, 7], and communicates with XtraGen via a Java API [7]. This allows users to issue calls to running generation systems. With each call, eGram sends an abstract content representation as input to the generator, and also transfers any modifications that have been made to the grammar since the last call. Input structures representing content to generate can also be defined via the eGram GUI.

[4]These include, e.g., use of undefined features and insufficient restrictions for feature values.

4 Development Environments for Dialogue Systems

4.1 A Graphical Editor for TAG-based Templates

Becker [2] briefly describes a tool for managing and editing TAG-based [26] templates that was developed in the context of the SmartKom project[5] [55]. SmartKom is a multi-modal dialogue system which is controlled by speech and gestures and also interprets facial expressions of users. It is capable of presenting graphical output, interacting with users via an animated talking agent, and carrying out a variety of other tasks such as controlling VCRs, sending e-mail, and querying databases. Instead of using strings, the generation component of the SmartKom system represents parts of sentences that make up a given template as a partial TAG derivation tree, thus bridging the gap between template-based and fully lexicalized generation [2].

The tool for managing and editing templates described in [2] allows users to organize related templates into *tree families*. Tree families are represented in the editor in the form of a directory structure that can be expanded and collapsed as necessary. Users can view graphical representations of individual trees and edit them in-place via a context menu. Supported operations include cutting, copying, and pasting subtrees, as well as changing the order of nodes by moving them horizontally. Users can also create new nodes, remove nodes along with their children, and turn individual nodes into root nodes [2].

4.2 NLG Template Authoring Environment

Caropreso et al. [9] describe an NLG Template Authoring Environment developed in the context of designing textual information and user interactions for *Serious Games*. A Serious Game is an interactive simulation game with the main goal of teaching players about a specific subject matter. In particular, the goal of the project described in [9] was to make generation functionality provided by the SimpleNLG library[6] [24] accessible to subject matter experts and game content designers lacking programming experience and advanced linguistic knowledge.

Compared to the systems described above, the graphical interface of the authoring environment is quite minimalistic. To create a template, users have to enter a natural language sentence, mark variable elements, and provide types and possible values of these elements. Additionally, they must specify syntactic dependencies between elements that are subject to variation. The system then displays the set of sentences that could be generated from the current template by determining all possible combinations of values for variable elements that respect the given dependencies. If necessary, templates can subsequently be refined by adjusting canned and variable portions of sentences and modifying dependencies between variable elements until they meet the needs of the target application[7] [9].

[5] `http://www.smartkom.org/`

[6] `https://code.google.com/p/simplenlg/`

[7] Note that functionality for refining templates was missing from the prototype implementation available at the time of publication.

4.3 DEXTOR

DEXTOR (Dynamic tEXt generaTOR) [40] is another tool geared towards supporting specification of natural language output for dialogue systems. Its background is similar to that of the template authoring environment developed by Caropreso et al. [9]: The target audience is users who need to create ongoing dialogue for interactive games. With DEXTOR, output is specified in the form of *dynamic text* which consists of nested templates that expand to predefined strings [40]. For instance, the authors state that one possible output corresponding to

```
inform(glados, chell, onsale(ties))
```

would be:

```
Glados says to Chell, 'There is a sale on ties.'
```

Starting from a *root template* with empty slots, DEXTOR's graphical interface allows users to fill slots with *subtemplates* by either typing them in or by locating them in a list of available templates and double-clicking them. In the previous example, the root template is `inform`. In addition to a list containing all available templates, the user interface also provides a *suggestion box* showing, in real-time, only those templates that would be appropriate to add to the dynamic text next. To come up with the list of suggestions, the system takes into account both the root template and subtemplates that have already been added [40].

While creating dynamic text using DEXTOR does not require extensive linguistic knowledge, the functionality described above does depend on the availability of template libraries and information about acceptable slot fillers for individual templates. No graphical interface seems to be available that facilitates creating these resources, and according to the authors, the task of creating template libraries in particular "may require non-trivial planning and organization" [40].

5 Summary

The tools described above share similar goals. Like the system described in this thesis, they aim to facilitate the process of engineering output for natural language generation systems. There is also a certain amount of overlap between these tools in terms of concrete features. For instance, several of them implement support for project management, testing, and context-sensitive editing. As will become apparent in later chapters that describe user-facing functionality provided by our system in detail, we adapted a number of ideas present in these tools for the system we implemented.

However, there are also a number of ways in which the systems described above differ from the system we present in this work: Full-featured development environments like IDEY and eGram are designed to minimize problems that arise from editing templates and rules manually. They also provide uniform interfaces to generation systems using different textual formats to represent templates and rules. These aspects arguably reduce development effort for template libraries and generation grammars. At the same time, however, these systems were not specifically designed to be usable by non-linguists or people without a background in computer science. Unsurprisingly, then, they still require different types of specialized knowledge which we can not assume people from unrelated backgrounds to have, and which take a while to acquire. For instance, using eGram requires good knowledge of rule-based systems and their derivational power, and using the editor developed for the SmartKom project requires familiarity with Tree-Adjoining Grammars. By contrast, we aim to make our system accessible to users who do not have a background in (computational) linguistics or computer science by reducing the amount of necessary background knowledge from these areas as much as possible.

On the other hand, systems like DEXTOR and the template authoring environment presented in [9] are specifically geared towards being usable by non-linguists, but lack even the most basic features for navigating and managing template corpora. As described in chapters 14 and 15, our system supports multiple ways of navigating and filtering rule bases to locate relevant content. Lastly, none of the tools described in the previous chapters are particularly suited for collaborative editing of content, as they require each end user to install and run their own copy of the software. The following chapter describes how our system addresses this particular concern.

Part III

System Architecture and Technologies

6 Architecture

6.1 Overview

As mentioned in Chapter 1, one of our main goals is to facilitate collaborative development of rule bases. Having rule developers access and modify a *single* copy of a rule base is a prerequisite for successful collaboration. In the context of the ALIZ-E project, this concern has so far been addressed by using a version control system to track changes made to plain text rule files. With respect to developing a dedicated system for creating and editing rules, however, this meant choosing an architecture that would allow rules to be stored centrally and accessed remotely. This is why we chose to adopt a cloud-based *Software-as-a-Service* (SaaS) approach for our system: A single instance of the application runs on a central host that multiple clients can connect to using a web browser (cf. Figure 1) [22]. All interaction with the system happens inside the browser. There is no need for end users to go through complex, operating system-specific install procedures, as the sole requirement for using the application is a modern web browser with JavaScript enabled.

The following two sections describe server-side and client-side architectures of our system in more detail. Integration with existing tools is addressed in Section 6.4. Chapter 7 provides information about technologies that were used to develop server-side and client-side components of our system.

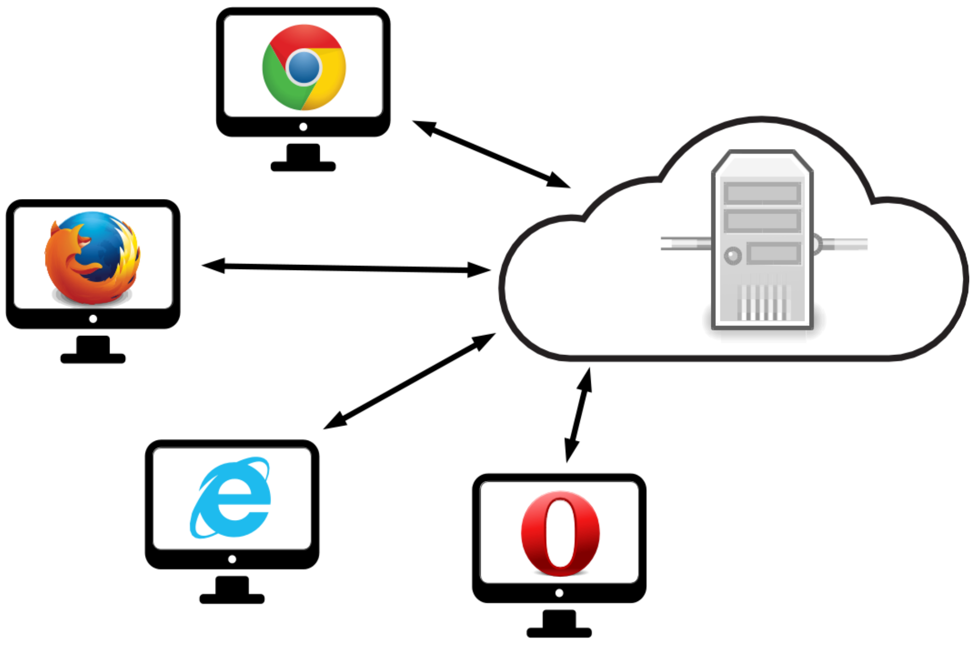


Figure 1: Software-as-a-Service architecture

6.2 Server-Side Architecture

The server-side architecture of our system is shown in Figure 2. It roughly follows the *Model-View-Controller* (MVC) pattern, which was first introduced by Krasner and Pope [32] for the purpose of building user interfaces in Smalltalk-80 [23]. One of the main benefits of this pattern is that it supports separating data from presentation [42]. Implementations of MVC vary in their interpretations of the MVC pattern, but in the context of web application development, responsibilities are generally divided between individual layers as follows [42]: *Models* represent application data and encapsulate logic for storing and operating on that data. *Controllers* are responsible for handling user requests, and the *view layer* is concerned with presentation of data. In particular, a *router* maps URLs requested by clients to appropriate controllers. Controllers communicate with the model layer to effect necessary changes to data stored in a persistence back-end. Additionally, controllers are responsible for rendering appropriate views by injecting relevant data into them, and sending them back to the client in an HTTP response.

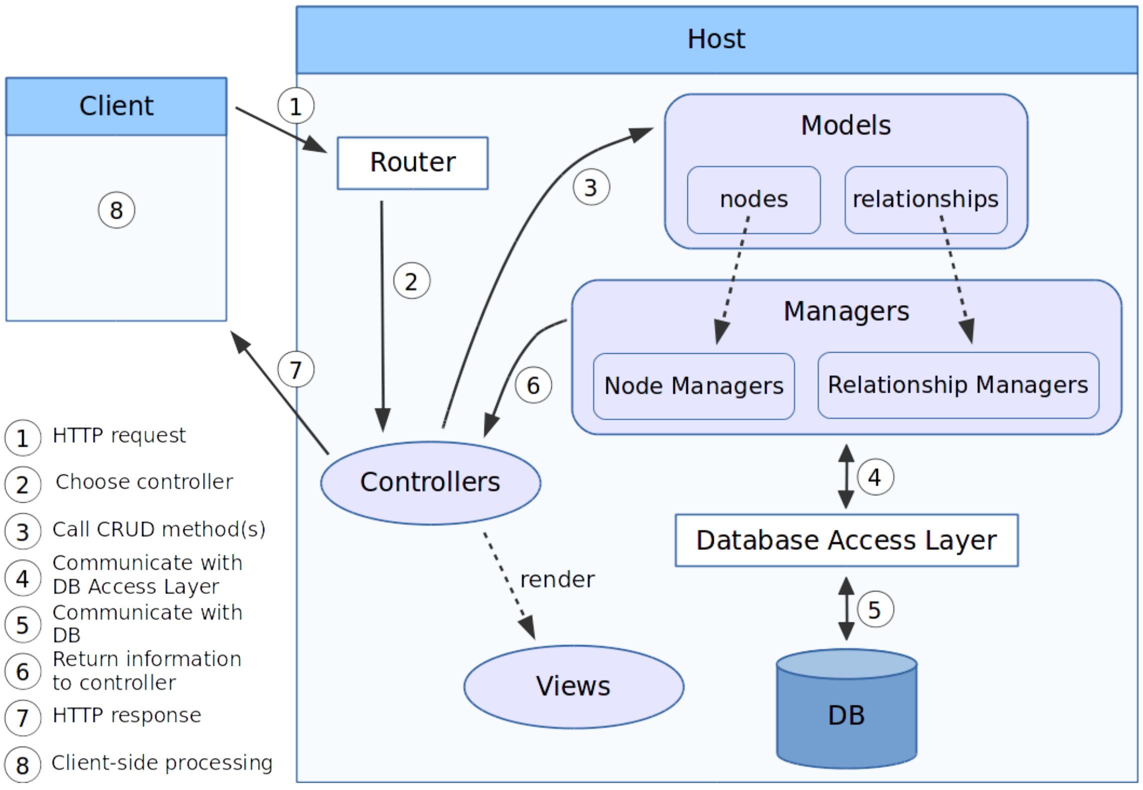


Figure 2: Flow of information on the server side. HTTP requests from clients are delegated to appropriate controllers by the router. Controllers operate on model objects by calling CRUD methods provided by `Manager` classes. Instead of communicating directly with the database, managers make use of the API provided by the database access layer to request information or effect changes to existing data. Results from operations involving the database are post-processed by managers and then returned to controllers, which send HTTP responses containing relevant data back to the client.

For reasons that will be explained shortly, we had to use a non-standard persistence back-end for storing rule data. From an architectural standpoint, using a custom persistence back-end necessitated the introduction of two additional components: First of all, functionality for communicating directly with the database was encapsulated in a separate *database access layer*. Secondly, a hierarchy of *managers* was added to handle communication with the database access layer in order to reduce coupling between models and the access layer, increase encapsulation, and obtain a sensible distribution of responsibilities. As shown in Figure 2, controllers communicate with managers via model classes: Each concept that is represented as a model has a static `nodes` field (or, in case of *relationship* models, a static `relationships` field) that stores a reference to an appropriate `Manager` object. Each manager implements appropriate CRUD (Create, Read, Update, Delete) methods for obtaining and operating on model data. Controllers call these methods via the `nodes` and `relationships` fields of relevant model classes. This design was inspired by the way application data is managed in *Django*[8], which is an MVC web framework for Python[9].

As a more concrete example of how individual server-side components of our system collaborate to make functionality available to end users, consider the following scenario: A user clicks a button that represents a link to an interface which lists all rules that are currently defined. (This interface is described in detail in Chapter 14.) This causes the browser to send an HTTP `GET` request to the URL associated with the link. At this point, server-side processing takes over: The router forwards the request to the controller that it knows to be responsible for handling this specific type of request. In order to be able to return an appropriate response to the client, the controller first needs to request a list of all rules from the `Manager` object responsible for handling rules. The manager contacts the database access layer with this request, which translates the request to an appropriate query and sends that query to the database. Upon receiving the results of the query from the database access layer, the manager generates a list of rule objects with fields set to appropriate values and returns that list to the controller. As a last step, the controller injects rule data into an appropriate view to generate the HTML page that the user requested by clicking the link in the browser, and sends the page back to the client in an HTTP response. JavaScript code that is necessary for further processing on the client side is transmitted in this step as well.

Having established how different server-side components interact to handle client requests, we now come back to the issue of persistence. The framework that was chosen to provide the core MVC architecture for our application supports a variety of *Relational Database Management Systems* (RDBMS) via the *Ebean* ORM[10]. Relational databases organize data in terms of the *relational model* [30], which was first introduced by E. F. Codd [14, 13]. It is based on predicate logic and set theory and uses the concept of a (mathematical) relation as its main primitive for modeling data [47]. As a result, when using RDBMS any data to be stored has to be modeled using tables (or, in mathematical terms, n-ary relations), with columns corresponding to attributes of concepts being modeled. Although this form of representation might have been appropriate for a subset of the concepts we needed to model, we chose not to use it: In ALIZ-E, rules for generating natural language output contain collections of feature-value pairs [31], which are naturally modeled as *feature structures* [28] (cf. Section 9.2.1). Feature structures, in turn, can be represented as *directed acyclic graphs* (DAGs) for the purpose of operating on them [28]. While relational databases can be used to store DAGs, the data model that is required for this to work (cf. [17] and [16]) uses a number of indirections that unnecessarily obscure the inherent structure of the original data [48]. Both retrieving and altering data requires complex (and in some cases non-standard) SQL queries (cf. [48] and [17]), and as Partner et al. [43] show, retrieval of connected data from RDBMS does not scale well.

[8]`https://www.djangoproject.com/`

[9]`https://www.python.org/`

[10]ORM stands for *Object Relational Mapping*, which describes the process of mapping objects to and from a relational format [29]. Ebean (`http://www.avaje.org/`) is an open source ORM tool for Java.

To circumvent these problems we chose to use a schema-free *graph database* [25] for the persistence layer of our system. Graph databases facilitate modeling highly interconnected data by representing it in the form of *nodes* and *relationships* [48]. Sections 9.1 and 9.2 describe the data model we developed for the purpose of storing and operating on rules in this type of database in detail. Aside from being more suitable for representing interconnected data, the schema-free nature of graph databases also facilitates the task of evolving data models over time. For instance, adding new types of nodes and relationships to support additional concepts does not require complex database migrations, and leaves existing data uncompromised [48].

In the current setup, the database runs in *server mode*, which means that it runs in a process that is completely separate from the process of the main application [48]. As a result, it is fully decoupled from other components of the system. The main application treats it as a remote web service and communicates with it via a *RESTful* interface. REST (Representational State Transfer) [20] is an architectural style that was developed by Roy Fielding as an abstract model of web architecture [19]. According to Richardson and Ruby [45], web services can be considered "RESTful" if their level of adherence to the architectural constraints for REST set forth in [20] is fairly high. Providing a detailed treatment of REST is outside the scope of this thesis, but one of the main ideas shared by many RESTful services is to represent data as *resources* and expose them via associated URIs, allowing clients to operate on them using standard HTTP methods (such as `GET`, `POST`, `PUT`, and `DELETE`) [45][11]. As mentioned above, knowledge about how to communicate with the database is encapsulated in the database access layer. Other components of the main application are not aware of how data is sent to and requested from the database (and do not need to be).

One major implication of treating the persistence back-end as a web service is that the lifecycle of the database does not depend on the lifecycle of the main application. As a result, the database server can be accessed from and plugged into other applications easily. This could be useful for, e.g., independent analysis of rule data. Another consequence of treating the database as a remote service was that we had to build custom support for wrapping multi-step operations into *transactions* into the database access layer: While the graph database implementation we use (cf. Section 7.1) does run queries inside of transactions by default, these transactions are committed at the end of each HTTP request [51]. This means that operations involving multiple steps will not be rolled back automatically if an intermediate step fails, which is likely to happen with multiple users accessing and modifying the same rule base simultaneously. As a result, rule data stored in the database might be left in an inconsistent state. To ensure that multi-step operations can only succeed if each intermediate step is successful, the database access layer provides an API for opening and closing transactions, allowing managers to execute individual steps belonging to a given operation inside a single transaction. This functionality relies on the use of an alternative transactional HTTP endpoint provided by the database that makes it possible to keep transactions open across multiple HTTP requests.

6.3 Client-Side Architecture

To improve user experience, a large number of user interactions with the system were implemented as client-side operations [49]. Page reloads generally only happen when navigating to a different system component (cf. Chapter 11). Actions requiring interaction

[11]For instance, a new node can be created in the database by `POST`ing appropriate data to `http://<host>/db/data/node`. An existing node can be retrieved or deleted by sending a `GET` or a `DELETE` request to `http://<host>/db/data/node/<id>`, and updated by sending a `PUT` request to `http://<host>/db/data/node/<id>/properties` [51].

with the back-end for the purpose of reading or altering data are communicated to the server via AJAX requests.

The architecture underlying client-side functionality follows an MV* pattern (cf. Figure 3). We initially tried to build client-side functionality using native JavaScript and *jQuery*[12] only. Due to the complexity of the data that the client-side front-end needs to manipulate and render, this approach turned out to be infeasible fairly quickly, causing us to adopt an additional framework in order to be able to separate responsibilities in a sensible and maintainable way. MV* patterns also make use of models and views, but differ from the MVC pattern in that they merge responsibilities of controllers into views or introduce additional components [42].

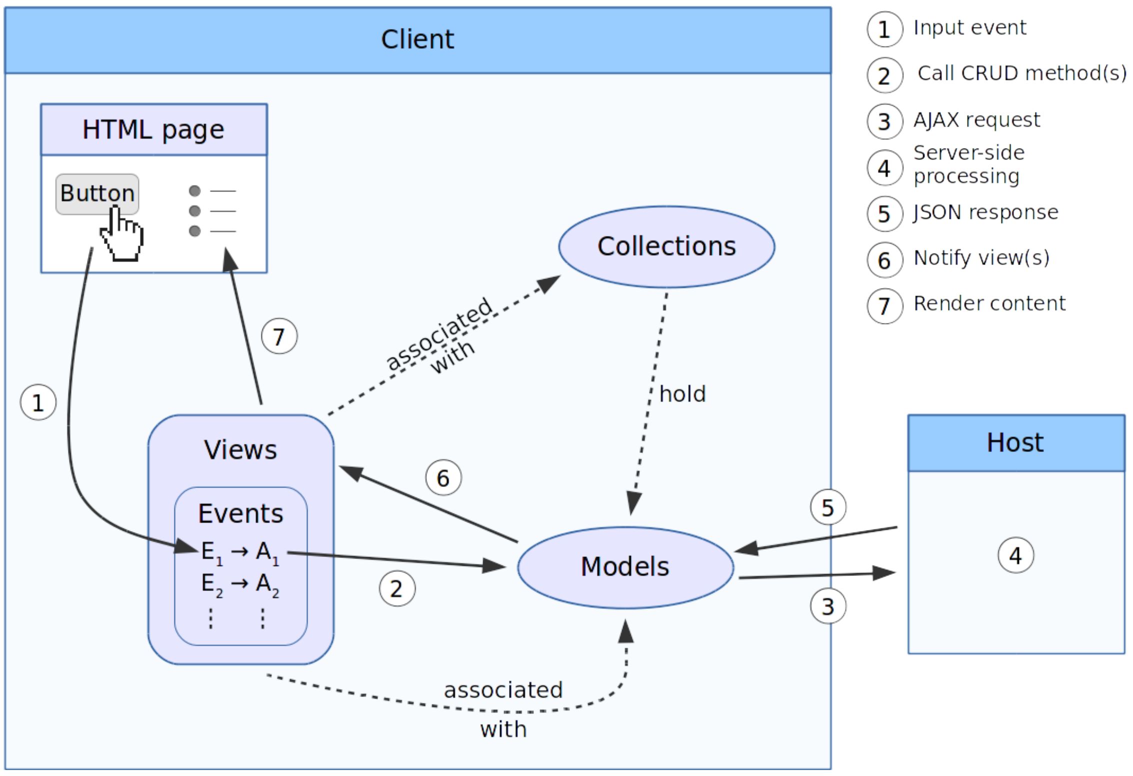


Figure 3: Flow of information on the client side. Input events such as button clicks cause views to call appropriate CRUD methods on associated model objects. CRUD methods issue AJAX requests to the server to send and/or receive data. Results of server-side processing are sent back to the client in a JSON response. Views are notified of any changes concerning model objects they are associated with. If necessary, views re-render updated information in the interface.

The framework that provides the basic architecture for client-side processing in our system is called *Backbone.js*. It provides two primitives for modeling application data, namely *models* and *collections*. Models serve an already-familiar purpose: They are used to represent domain entities and provide methods for operating on data that is associated with these entities. Collections are sets of models. Both models and collections provide CRUD methods for synchronizing the data they hold with the application server. These methods wrap AJAX functionality provided by jQuery, resulting in a large reduction in the amount of code required to perform RESTful synchronization for model objects [42].

[12]`https://jquery.com/`

Views contain logic for rendering data that has been stored using models and collections. Upon creation, view objects are associated with model or collection objects which enables them to access the data they are supposed to render. Backbone.js supports the use of client-side templating libraries such as *JsRender*[13], *Mustache*[14], and *Underscore.js* micro-templates[15] for rendering model and collection data to HTML, but also allows for custom rendering solutions [42]. Our system makes use of nested views and models for rendering and representing complex entities. This reduces the amount of HTML that must be rendered per view, allowing us to use functionality for creating HTML elements provided by jQuery (instead of a dedicated templating library).

Being an MV* framework, Backbone.js merges responsibilities that are usually associated with controllers into the view layer: On the client-side, requests correspond to *events* such as clicks on specific HTML elements or changes to models and collections [42]. Views can be set up to listen to both of these types of events. When a specific event occurs, views listening to it can respond by executing a number of appropriate actions. These actions usually involve making changes to rendered representations of associated models and collections.

Figure 3 illustrates the flow of information between different components on the client-side. Differences between server-side and client-side data models are described in Chapter 10. Appendix A provides a detailed treatment of client-side data models.

6.4 Integration with Existing Tools

Rules created with our system can be prepared for testing and on-line processing by *exporting* them to a format that the rewriting engine can understand. There are three main advantages to this approach: First, it maximizes reuse of existing functionality by allowing the system to integrate into the overall pipeline of development tools for rule bases without interfering with existing implementations[16]. Note that this advantage is specific to the ALIZ-E project, i.e., the context in which our system was developed. Secondly, it preserves the option of working with plain text representations of rules directly. At this stage, this is especially important because the system does not (yet) support editing rules whose output consists of instructions for manipulating abstract representations of situational knowledge. And lastly, it opens up the possibility to port the target system to projects using different rule formats by substituting the export module (cf. Section 20.1), while leaving other components of the system untouched. Section 14.3 describes how the export process can be triggered by end users. A detailed explanation of processing steps involved in exporting rules is presented in Appendix B.3.

[13]`https://github.com/borismoore/jsrender`

[14]`https://mustache.github.io/`

[15]`http://underscorejs.org/`

[16]Tools developed prior to the system presented here allow rule developers to debug individual rules by stepping through the processing steps that take place when a given rule is applied [31]. Given availability of appropriate test files, it is also possible to batch test an entire rule base. Syntax highlighting for rule and batch test files is available through integration with the Emacs editor (`https://www.gnu.org/software/emacs/`).

7 Technologies

7.1 Server-Side Technologies

As explained above, the server-side portion of the system consists of two major components, namely the main application and the persistence back-end. Both the graph database we use to store rule data and the MVC framework providing the basic architecture for the main application are written in Java[17].

7.1.1 MVC Framework

The framework that forms the basis for the main application is called *play*[18]. It belongs to a family of *evented* web frameworks enabling the use of *asynchronous* (or *non-blocking*) *I/O* when making calls to remote services [4]. This means that threads handling individual requests do not have to wait for network calls to complete. Instead, they can process other requests in the meantime and resume work when the response from the network call becomes available [4]. With respect to our system, asynchronicity on the server side is important because as mentioned above, the main application treats the persistence layer as a remote service.

Play uses Scala as its default templating language for the view layer. For our purposes, a less powerful templating engine would have sufficed: We only use server-side templates to provide basic scaffolding for individual pages. Complex data structures representing rule components are stored in HTML 5 `data-*` attributes [37], and final rendering of these data structures takes place on the client-side. However, in order to avoid making yet another addition to the set of technologies that were used to build the system we chose not to replace the default templating system.

7.1.2 Persistence Layer

We use an open source implementation of a graph database management system called *Neo4j*[19] to store rule data. The graph database exposes a *property graph model* that makes it possible to conceptualize domain knowledge using nodes and directed relationships. As the term *property graph* suggests, both nodes and relationships can store additional information about entities (nodes) and connections (relationships) between them in the form of *properties*. Properties are key-value pairs whose keys must be strings and whose values can be arbitrary data types [48]. In addition to properties, nodes can have one or more *labels*, and relationships always have a *type* [48, 51]. Labels make it possible to restrict operations to specific subsets of nodes. Relationship types provide enhanced possibilities for graph traversal. To illustrate, consider the example graph shown in Figure 4, which represents a small social network. Although structurally simple, using relationship direction and type this graph can be queried for the following information [51]:

[17]More precisely, the MVC framework is written in Java and Scala (`http://www.scala-lang.org/`), but since other tools developed in the context of the ALIZ-E project were implemented in Java we chose to use the Java version of the framework.

[18]`https://playframework.com/`

[19]`http://www.neo4j.org/`

Persons	**Obtained by following**
followed by a person	outgoing `FOLLOWS` relationships, depth 1
following a person	incoming `FOLLOWS` relationships, depth 1
blocked by a person	outgoing `BLOCKS` relationships, depth 1
blocking a person	incoming `BLOCKS` relationships, depth 1

Table 1: Information that can be obtained from the graph shown in Figure 4 [51]

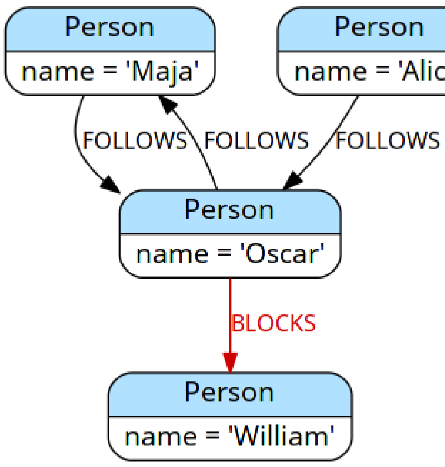


Figure 4: Example graph representing a small social network [51]

Sections 9.1 through 9.3 describe how we use these modeling primitives to represent relevant data for the purpose of storing it in the database.

Query Language

Neo4j defines a custom query language for retrieving and manipulating data stored in property graphs called *Cypher*. Its declarative syntax is designed to mimic visual representations of graphs. This makes it possible to describe data in the form of *patterns* when querying the database [48]. Discussing the Cypher query language in detail is outside the scope of this thesis, but as an example, consider the following query:

```
MATCH (x:Person)-[:FOLLOWS]->(y)
WHERE x.name = 'Alice'
RETURN y;
```

The first line contains a `MATCH` clause that defines the pattern to locate in the database. Nodes are surrounded by parentheses, so in this example, both `(x:Person)` and `(y)` are placeholders for nodes. `x` and `y` are *identifiers*. The purpose of identifiers is to make it possible to reference specific parts of patterns by naming them [51]. `Person` is a label. Including this label in the query reduces the set of nodes matching the pattern `(x:Person)` to all nodes `x` whose set of labels includes `Person`. In their simplest form, relationships are represented as arrows consisting of two dashes (`--`). If necessary, directionality of relationships can be indicated using `<` and `>`: Relationships matching the pattern specified in the query shown above must point from nodes matching the pattern `(x:Person)` to nodes matching the pattern `(y)`. Incoming relationships of `(x:Person)` nodes are filtered out.

Arrows representing relationships can be annotated with additional constraints enclosed in square brackets. In the example above, the string following the colon specifies that the entire pattern should only match nodes connected by relationships that are of type `FOLLOWS`.

The `WHERE` clause in the second line adds a constraint to the pattern described by the `MATCH` clause: It specifies that nodes matching the pattern `(x:Person)` must also satisfy the condition that their `name` property be set to the value `Alice`. Constraints can be negated using `NOT`, and it is possible to specify multiple constraints by chaining them with Boolean operators `AND`, `OR`, and `XOR` [51]. Lastly, `RETURN` clauses determine which nodes, relationships, and properties from the data matching a pattern should be returned by the query [48]. In the example above we are only interested in the end nodes of the pattern.

Summing up, the query shown above can be used to obtain the set of persons being followed by a person named `Alice`[20]. When run against the graph in Figure 4, it returns a single node, `Oscar`, who is the only person `Alice` is currently following.

7.2 Client-Side Technologies

Client-side functionality was implemented using HTML for structure, CSS for formatting, and JavaScript for user interactions. A number of different frameworks and libraries were employed to aid development of client-side functionality. Important characteristics of the MV* framework that provides the basic architecture for client-side processing were presented in Section 6.3, so we will not repeat them here. The following sections give a brief overview of technologies providing functionality for laying out UI components, making AJAX requests, and for manipulating structure and content of HTML pages representing individual interfaces of our system.

7.2.1 Layout

As mentioned at the beginning of this chapter, content displayed to users was formatted using *Cascading Style Sheets* (CSS)[21]. Aside from specifying fonts, colors, and spacing for HTML elements, CSS can also be used to create responsive, grid-based layouts for precise placement of interface components [15, 53]. When developing such layouts, special attention must be payed to concerns of cross-browser compatibility in order to avoid forcing end users to use specific browsers [54]. Since our focus was on designing and implementing functionality for working with rewriting rules, we chose to circumvent these issues by using an existing framework with built-in support for grid-based layouts called *Twitter Bootstrap*[22]. Aside from providing CSS for grid-based layouts, this framework also implements a large variety of reusable interface components such as drop-down menus, tabs, and button groups. Most notably, our application uses a Bootstrap navigation bar[23] for providing quick access to different types of functionality (cf. Chapter 11), and incorporates a number of buttons that are rendered using *glyphicons*[24].

[20]Note that for the purpose of illustrating different aspects of Cypher syntax, we omitted the `Person` label from the end node of the pattern to match. Therefore, when running this query against data sets allowing other types of entities to be followed, the result set might include nodes with different labels.

[21]`http://www.w3.org/Style/CSS/`

[22]`http://getbootstrap.com/`

[23]`http://getbootstrap.com/components/#navbar`

[24]`http://getbootstrap.com/components/#glyphicons`

7.2.2 AJAX Functionality and User Interactions

Backbone.js, the MV* framework that provides the basic architecture for client-side functionality of our system, does not include support for making AJAX requests to the application server. However, as mentioned in Section 6.3 above, methods for synchronizing model data to the server do require AJAX functionality to be available. Furthermore, Backbone.js views rely on external support for DOM[25] manipulation. The jQuery library addresses both of these concerns, and although it is possible to use other libraries exposing similar APIs (such as *Zepto*[26]) with Backbone.js, jQuery provides the highest level of compatibility. For these reasons we chose to add jQuery to the set of tools we used to build our system.

Lastly, a number of features for working with rule data were implemented as drag-and-drop operations (cf. chapters 12 and 13). Support for drag-and-drop is enabled through the use of *jQuery UI*[27] which is built on top of jQuery and provides "a curated set of user interface interactions, effects, widgets, and themes" [27].

This concludes our discussion of system architecture and the technologies that were used to build the system. The following chapters describe the native format of rewriting rules, as well as the server-side and client-side data models we developed for storing and processing them.

[25]The *Document Object Model* (DOM) is an API for representing and manipulating contents of HTML and XML documents [21].

[26]`http://zeptojs.com/`

[27]`http://jqueryui.com/`

Part IV

Data Models

8 Native Rule Format

Rewriting rules for on-line generation of natural language output consist of two components, a *left-hand side* (LHS) and a *right-hand side* (RHS)[28]. In native rule syntax, these components are separated by an arrow (->):

```
:dvp ^ <SpeechAct>greeting
^ <Context>(<RobotName>#robot ^ <Encounter>first)
->
###x = concatenate("ciao, mi chiamo ", #robot),
###y = concatenate("ciao, sono ", #robot),
# ^ :canned ^ <stringOutput>random(###x, ###y) ^ <SpeechModus>indicative.
```

LHS of rules contain a set of *matching conditions* which must be fulfilled for a rule to apply: During on-line processing, knowledge that a conversational agent has about the current situation is represented as a *proto-logical form* (PLF) [31, 41]. PLFs are instances of feature structures [41], i.e., they are collections of *features* and associated *values* [28]. When it is the agent's turn to speak, decisions about what to say are made by matching the PLF against conditions specified by LHS of individual rules. If a PLF fulfills the conditions established by a given rule, that rule is applied to the PLF (cf. below). Matching conditions specify a set of features that must be present in the PLF. For each feature they can also specify the value to which it should be set. In native rule syntax, features are enclosed in angular brackets. To match the LHS of the rule shown above, a PLF must contain four features: `SpeechAct`, `Context`, `RobotName`, and `Encounter`.

Features differ in the types of values that they take. If a feature is set to a simple string value, we call it *atomic*. In the example above, `SpeechAct`, `RobotName`, and `Encounter` are instances of atomic features. On the other hand, if the value of a feature consists of a number of feature-value pairs enclosed in parentheses, we call it *complex*. The LHS of the rule shown above contains a single feature that is complex, namely `Context`. Values starting with a hash symbol (#) are *variables*. They are used to capture concrete values associated with features in a PLF during on-line processing. Among other things (cf. [31]), this makes it possible to reference feature values that only become known at run-time from other locations within the same rule. To give a concrete example, the rule shown above stores the name of the agent in a variable called `#robot`. It then references that variable on the RHS to incorporate the name of the agent that is currently involved in a dialogue with a user into the natural language output it produces.

In addition to features and values, every PLF also has a *type* [41]. Types are represented using a colon followed by a sequence of non-whitespace characters in native rule syntax. The presence of a type in the LHS of a rule places further constraints on the input PLF. For instance, the rule shown above only applies to feature structures of type `:dvp`. The ^ operator is used to combine individual match conditions to form more complex match expressions [31]. Specifically, the presence of this operator requires that a given PLF match all of its operands.

RHS of rules provide explicit instructions about how the rewriting engine should modify a matching PLF. Additionally, they can specify one or more *output strings* that are appropriate for the conversational agent to produce if their parent rules apply. For example, the

[28]Note that from this point on we will be using "LHS" and "RHS" as shorthand for both singular and plural forms of the terms they represent.

rule shown above defines `ciao, mi chiamo #robot` and `ciao, sono #robot` as possible outputs. This is achieved by using a function called `concatenate`[29] to combine two pieces of canned text (`ciao, mi chiamo` and `ciao, sono`) with the value of the `#robot` variable at run-time. The resulting output strings are stored in two separate variables `###x` and `###y`[30]. Instructions for modifying the input PLF are given in the last line of the rule definition: In the context of an RHS, a single hash character represents the PLF against which a rule is matched, and the `^` operator signifies *addition*. In its entirety, the last line of the rule definition instructs the rewriting to change the type of the PLF to `:canned`[31], and to add two features (`stringOutput` and `SpeechModus`) with appropriate values to it. It also specifies that the rewriting engine should make use of a function called `random`[32] to compute the value of the `stringOutput` feature from the values of `###x` and `###y`. As the name suggests, this function randomly selects and returns one of the arguments that are passed to it at each application of a rule[33]. During on-line processing, actual output is produced by reading the value of the `stringOutput` feature from the altered feature structure.

Since one of our goals is to make the process of working with rules more accessible to people without background knowledge in (computational) linguistics and computer science, our system currently hides the fact that RHS of rules provide instructions for manipulating PLFs from end users completely. Editing functionality for RHS of rules (described in detail in Section 13.3) allows users to focus solely on designing appropriate verbalizations. Instructions for choosing among available output strings and adding the `stringOutput` feature to input PLFs are automatically incorporated into rules when exporting them to plain text as described in Section 14.3.

Finally, it must be noted that the example above only shows a subset of the syntax for rewriting rules that has been developed in the context of the ALIZ-E project. We omit additional details about native rule syntax here because the server-side and client-side data models that we developed for the purpose of storing and operating on rules only take into account the concepts presented above. Please refer to Kiefer [31] for a formal specification and an in-depth treatment of rewriting rule syntax.

9 Server-Side Data Model

This chapter describes the server-side data model that we developed for the purpose of storing rules created with our system. Section 9.1 describes how features and associated values are represented, and Section 9.2 describes the data model for rules. In the last section of this chapter we briefly touch on a basic model for representing registered users.

Note that the following sections do not describe a new formalism. Rather, they describe how we make use of the property graph model to translate rule data (augmented with specific kinds of useful meta-information) to a format that is serializable using the Neo4j graph database implementation. The process of designing the server-side data model was

[29]The `concatenate` function is part of a set of built-in functions provided by the rewriting engine. In native rule syntax, built-in functions can be used from both LHS and RHS of rules [31].

[30]Variables prefixed with three hash symbols (`###`) are called *right-hand side local variables* [31]. As their name suggests, they are local to RHS of rules.

[31]Feature structures can only have one type. As a result, "adding" a new type to a feature structure causes the existing type to be replaced.

[32]This function also belongs to the set of built-in functions provided by the rewriting engine [31].

[33]Note that if there is no variable content to be interpolated into individual output strings, they can also be passed directly to the `random` function; it is not necessary to use the `concatenate` function in the definition of a rule in this case.

greatly influenced by practical concerns: We wanted to represent rule data in a way that would keep the amount of duplicate information we needed to store to a minimum. To address this goal, we decided to introduce different types of nodes to represent complex entities, and to connect them using appropriately-typed relationships. This allows the system to store components shared by many rules (such as features) only once, which would not have been possible if we had opted for storing information about rule components as properties of a single rule node. In addition to minimizing redundancy, we also wanted to keep the number of processing steps involved in performing relevant operations minimal. This was achieved by tailoring both the server-side and the client-side data models to the different types of editing operations that we wanted to support.

9.1 Features

Features are represented as nodes in the database. Each feature node stores the *name* and *type* of the feature it represents, as well as a *description*. Additionally, feature nodes have a property called `uuid` whose value is a *Version 4* UUID[34]. The `uuid` property is used for internal purposes and becomes relevant when operating on stored representations of LHS. Section 9.2.1 below explains why it was added to the set of properties we store for feature nodes. By introducing a `description` property we enable users to document and communicate to others how individual features should be used[35]. Note, however, that we do not force users to provide descriptions for features: If they fail to specify a description for a feature when first creating it, the system will assign a default value of "`...`" to the `description` property that can be modified at any time (cf. Chapter 12). The `type` of a feature determines whether is atomic or complex. As described in Chapter 8, atomic features take string values and complex features embed sets of feature-value pairs. Feature nodes are uniquely identified by a combination of their label (`Feature`) and the value of the `name` property: No two features are allowed to have the same `name`.

In addition to `name`, `type`, and `description` properties, each feature is associated with a number of possible *targets*. Complex features target other features: If a feature A targets another feature B, this means that the targeted feature (B) is allowed to appear in the set of feature-value pairs embedded by A. Atomic features, on the other hand, target values: In the context of any given rule, the value of an atomic feature can only be set to a targeted value. Associations between features and appropriate target nodes are represented by relationships of type `ALLOWS`. Relationships of this type currently do not store any additional information in the form of properties[36].

As hinted at above, values of atomic features are represented as nodes as well. They have a single property called `name` which – in conjunction with the `Value` node label – uniquely identifies them. Value names are thus subject to the same uniqueness constraint that applies to feature names.

By default, each atomic feature is associated with a special value node whose `name` is set to `underspecified`. The purpose of this value is explained in Section 9.2.1 below.

[34]UUID stands for *Universally Unique Identifier*. A UUID is 128 bits long and can be generated using a number of different algorithms. Version 4 UUIDs are generated from truly random or pseudo-random numbers [38].

[35]Since feature descriptions constitute meta-information that is irrelevant for the purpose of generating output during on-line processing, they have no counterpart in native rule syntax.

[36]The purpose of associating features with specific targets via `ALLOWS` relationships is similar to the role of types in typed feature structure formalisms: It restricts the set of features that are allowed to appear in feature structures embedded by specific features. The main difference is that in the context of typed feature structure formalisms, the type of a given feature structure can determine, for each feature that belongs to it, the exact value the feature is supposed to take (cf., e.g., [33]). By contrast, the data model described here only allows users to specify general restrictions for sets of values specific features are allowed to take (by associating atomic features with one or more values).

With the exception of the `underspecified` value, the lifespan of values is controlled by the atomic features that target them. That is, a given value will continue to exist as long as it is referenced by at least one feature. The associated node will be removed from the database as soon as it becomes *orphaned*.

Figure 5 shows an example subgraph of features and associated values.

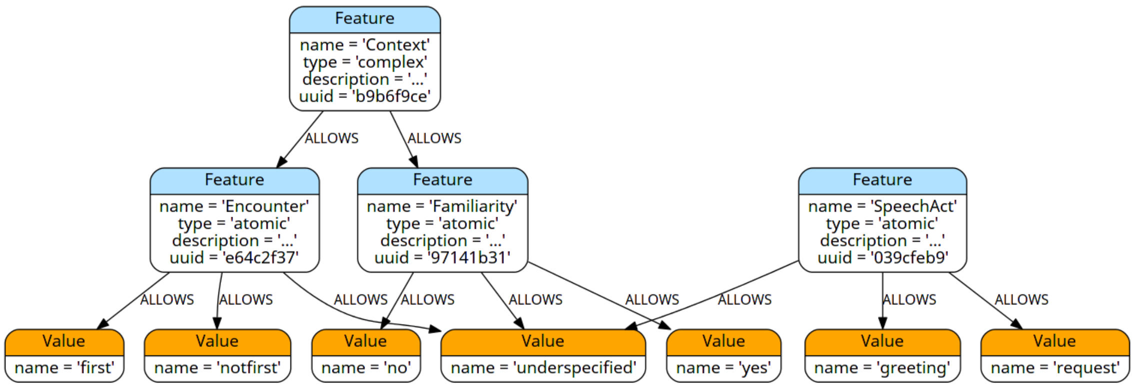


Figure 5: Example subgraph of features and associated values. Descriptions have been truncated for brevity.

9.2 Rules

Rules are represented as nodes with `name`, `uuid`, and `description` properties[37]. Just like feature descriptions, rule descriptions can be omitted when creating new rules: The `description` property will be set to a default value of "`...`" in this case. The `uuid` property is set to a Version 4 UUID that is generated upon rule creation and used for internal purposes. Section 9.2.1 below provides more detailed information about what these purposes are. Rule nodes are uniquely identified by a combination of their label (`Rule`) and the value of the `name` property. No two rules are allowed to have the same `name`.

In addition to the `name`, `uuid`, and `description` properties, each rule is associated with a subgraph representing its LHS via a relationship of type `LHS`, and another subgraph representing its RHS via a relationship of type `RHS`. The following two sections describe the data model for these rule components in detail.

9.2.1 LHS

As mentioned in Chapter 8 above, LHS of rules contain pairs of features and values that are matched against input structures to determine whether or not a given rule should be applied during on-line processing. We therefore chose to represent them as *Attribute-Value Matrices* (AVMs)[38], both for the purpose of graphical editing (cf. Figure 12 in Section 13.2) and for the purpose of storing them in the database. Each AVM is represented as a node with a single property called `uuid`. In conjunction with the `AVM` node label, this property identifies AVM nodes uniquely. For each AVM node, the value of the `uuid` property is set to a *Version 3* UUID[39]. If an AVM node represents an LHS, this UUID is derived from the UUID of the parent rule. On the other hand, if an AVM node represents a nested

[37]Note that none of these properties have counterparts in native rule syntax.
[38]AVMs are graphical representations of feature structures [28, 3].
[39]Version 3 UUIDs are generated from "names" that are unique within some namespace [38].

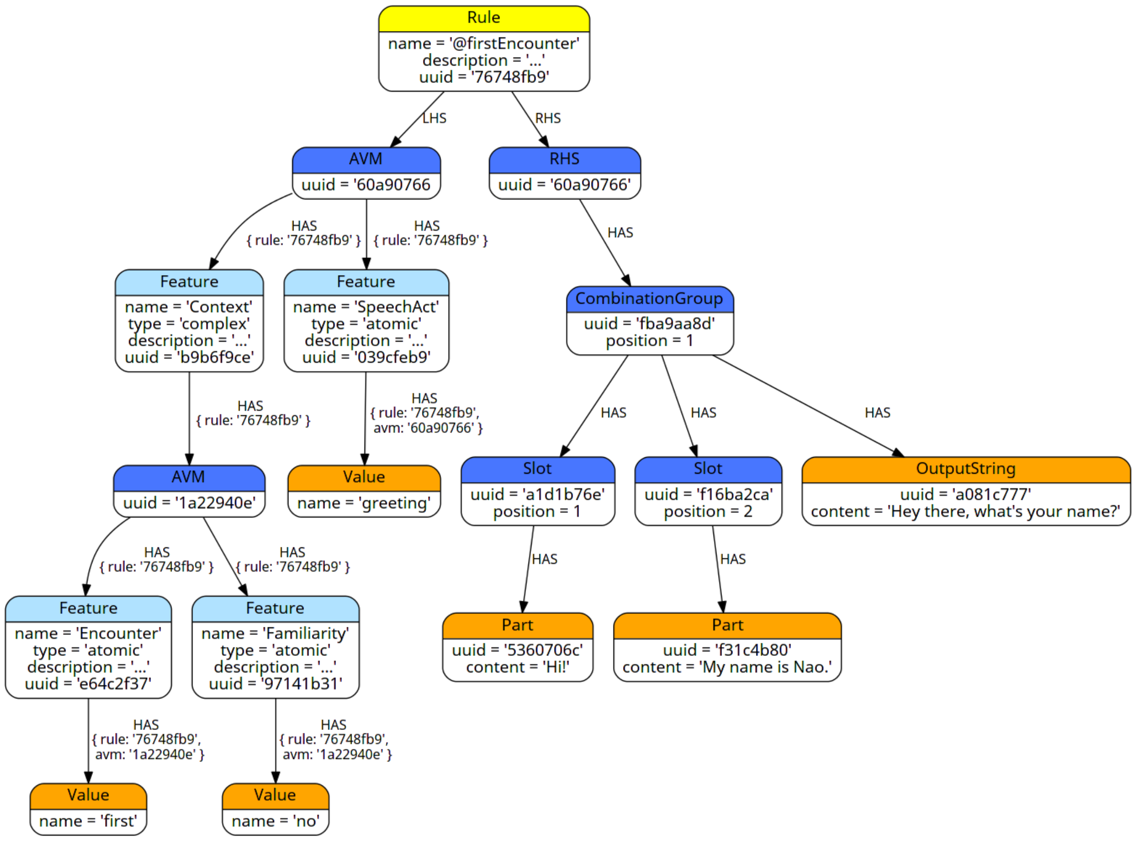


Figure 6: Example subgraph of a rule. UUIDs and descriptions have been truncated for brevity.

substructure of an LHS (i.e., a set of feature-value pairs embedded by a complex feature), the UUID is derived from a combination of the UUID of the parent AVM and the UUID of the complex feature embedding the AVM. The main benefit of using derived UUIDs is that UUIDs of AVM nodes belonging to a given rule can be determined programmatically without having to query the database for this information, allowing write operations to target appropriate nodes directly. Since names of rules and features are subject to change, we would risk losing this ability if we were to derive UUIDs for AVMs from values of `name` properties of rule and feature nodes. This was the main reason for introducing `uuid` properties for rule and feature nodes as well. Deriving UUIDs of embedded AVMs from UUIDs of parent AVMs *and* UUIDs of embedding features is necessary because UUIDs of embedded AVMs would not be unique if they were derived from UUIDs of parent AVMs or embedding features *alone*: In the former case, every substructure of a specific AVM would have the same UUID, and in the latter case, substructures embedded by the same feature would have the same UUID across rules.

In addition to the `uuid` property, each AVM is associated with zero or more features. Connections between AVMs and features are represented by relationships of type `HAS` with a single property called `rule`. The value of this property is set to the UUID of the parent rule. Features, in turn, are associated with substructures (if they are complex) or values (if they are atomic) in the context of AVMs. Relationships connecting complex features to substructures have the same type and set of properties as relationships connecting AVMs to features. Relationships representing associations between atomic features and values store an additional property called `avm`. For each relationship, this property is set to the UUID of the parent AVM of the start node, that is, the AVM in which the start node `HAS` the end node of the relationship as a value. This allows us to precisely identify the context in which a feature is associated with a given value.

We mentioned in Section 9.1 above that every atomic feature `ALLOWS` the `underspecified` value. In the context of rules, each atomic feature `HAS` the value `underspecified` by default. This covers cases in which the LHS of a rule requires the *presence* of a certain feature, but makes no assumptions about the associated value. Each complex feature is associated with (`HAS`) an empty substructure, i.e., an AVM with no outgoing `HAS` relationships.

9.2.2 RHS

Just like LHS of rules, RHS of rules are represented as nodes with a single property called `uuid`. For any given rule, the value of the `uuid` property of the RHS node it is associated with is identical to the value of the corresponding LHS node. That is, the value of the `uuid` property of an RHS node is a Version 3 UUID that is derived from the UUID of the parent rule. RHS nodes are uniquely identified by a combination of their label (`RHS`) and the value of the `uuid` property.

In addition to the `uuid` property, each RHS is associated with one or more *combination groups* via relationships of type `HAS`. Unlike `HAS` relationships connecting AVMs to features and features to embedded AVMs or values, these relationships do not have any properties. The following sections describe the data model for combination groups and their components in detail.

Combination Groups

Combination groups do not have a counterpart in native rule syntax. Their exact purpose is described in Section 13.3. For now, it is enough to mention that they are used to group sets of output strings belonging to a given rule. Each combination group is represented as a node with two properties called `uuid` and `position`. The value of the `uuid` property is a Version 4 UUID that is generated upon group creation. The `position` property is used to represent the (1-based index) position of a group in the list of combination groups associated with a given RHS. By imposing an artificial order on the set of combination groups associated with a given RHS, the system is able to keep the arrangement of visual representations of these groups consistent between editing sessions. Nodes representing combination groups are uniquely identified by a combination of their label (`CombinationGroup`) and the value of the `uuid` property.

In addition to the `uuid` and `position` properties, each combination group is associated with zero or more *output strings* and zero, or at least two *slots*. Relationships connecting combination group nodes to output strings and slots are of type `HAS`.

Output Strings

We explained in Chapter 8 above that RHS of rules contain specifications of output strings, and that one of these strings is chosen randomly every time a given rule applies. In our system, output strings are represented as nodes with two properties called `uuid` and `content`. As its name suggests, the `content` property stores the content of a given output string. The value of the `uuid` property is a Version 3 UUID that is derived from the value of the `content` property. As a result, output string nodes are uniquely identified by a combination of their label (`OutputString`) and the values of the `uuid` and `content` properties.

Combination groups control the lifespans of output strings. That is, a given output string will continue to exist as long as it is referenced by at least one combination group.

It will be removed from the database as soon as it becomes orphaned, i.e., as soon as it loses the last incoming `HAS` relationship.

Slots

As mentioned above, combination groups are not only associated with a number of output strings; they can also embed two or more slots. Like combination groups, slots are structural entities. They are used to group strings that can be used as building blocks for full output strings. We call these strings *parts*. They are described in more detail below. Slots are represented as nodes with two properties: `uuid` and `position`. The value of the `uuid` property is a Version 4 UUID that is generated upon slot creation. In conjunction with the `Slot` node label, this property identifies slot nodes uniquely. The `position` property records the (1-based index) position of a slot in the list of slots associated with a given combination group. Unlike combination groups, slots must be ordered not only for consistency across editing sessions, but also because parts stored in slots are combined in a linear fashion when generating full output strings from them (cf. Section 13.3.2). Without a fixed ordering of slots, we would not be able to guarantee that generated output strings would be syntactically valid.

In addition to the `uuid` and `position` properties, each slot is associated with zero or more of the aforementioned *parts* and can *cross-reference* zero or more rules. Relationships connecting slots to parts and rule nodes are of type `HAS`.

Parts

Parts represent non-overlapping portions of output strings. As will be described in detail in Section 13.3, they are created by *splitting* output strings at word boundaries. With respect to the data model, each part is represented as a node with two properties, `uuid` and `content`, which serve the same purpose as the identically-named properties of output string nodes: The `content` property stores the content of a given part, and the `uuid` property is a Version 3 UUID derived from the value of the `content` property. Part nodes are uniquely identified by a combination of their label (`Part`) and the values of the `uuid` and `content` properties. Lifespans of parts are controlled by slots. That is, a given part exists as long as it is referenced by at least one slot. The associated node will be removed from the database as soon as it becomes orphaned.

Cross-References

As mentioned above, cross-references between slots and rules are represented by relationships of type `HAS`. These relationships currently do not have any properties: Unlike features, slot nodes are unique to their parent rules. As a consequence, individual slots can not reference the same rule twice, making it unnecessary to record information about parent rules for the purpose of re-establishing rule context later on. Adding the concept of cross-references to the data model is a basic requirement for allowing modularization of rule bases. By establishing a connection between a slot S and a rule R (cf. Section 13.3.2), users can import the full output of R into S: Each output string associated with R will be treated as a part of S when exporting rules to native formats. This is explained in detail in Section 14.3.

9.3 Users

Going beyond representations of features and rules, the data model also includes a basic user model. At the time of this writing, the sole purpose of storing information about users is to be able to restrict general access to the system to registered users. However, by incorporating users into the data model we have also laid the groundwork for adding more sophisticated access control, and for attributing modifications to features and rules to specific users when recording version histories of these entities.

Users are represented as nodes with a label of `User` in the database. Each user node has two properties for storing a user's `email` address and `password`, respectively. Information for both of these properties must be supplied upon registration. No two users are allowed to have the same `email` address. Attempts to register new accounts using an email address that is associated with an existing user are rejected by the system.

10 Client-Side Data Models

User-facing functionality of our system – which will be described in detail in Part V – is spread out over a number of different interfaces. These interfaces differ with respect to how much information needs to be available to them about entities such as features and rules. As a result, there is no uniform data model on the client side that is reused across all interfaces. Instead, each interface defines a custom data model that omits any information that is irrelevant to the functionality it provides. For instance, as it simply lists rule names and descriptions, the interface for browsing rules (described in Chapter 14) does not need access to detailed information about LHS and RHS of rules. On the other hand, interfaces for editing rule input (described in Section 13.2) and rule output (described in Section 13.3) allow users to modify LHS and RHS, respectively, so they do need information about these entities. Since rule data is much more complex than feature data, client-side models for rules vary even more across interfaces than client-side models for features. In some cases, the client-side data model incorporates a small number of models that do not have a direct counterpart on the server side.

On a more technical level, properties of nodes stored in the graph database generally translate to *attributes* of model objects on the client-side. Associations between different types of entities which are modeled using relationships on the server side are recreated through the use of nested models and collections on the client-side.

Ideas presented in the following chapters do not rely on a detailed understanding of data models used by individual interfaces. Therefore, we do not provide more information about these models here. Please refer to Appendix A for an in-depth treatment of client-side data models.

Part V

User-Facing Functionality

11 Overview

The following chapters describe user-facing functionality provided by our system. There are four main components: a feature editor, a rule browser, a search interface, and a rule editor. The latter is split up into a number of subcomponents for creating new rules, editing rule input, and editing rule output. A navigation bar at the top of the application window allows users to switch between individual components of the system. As the contents they display are specific to individual rules, subcomponents of the rule editor are not listed in the navigation bar. They are accessed via the rule browser (cf. Chapter 14), which serves as an entry point for viewing details about individual rules and modifying them. Except for the interface for creating new rules, each of the components mentioned above has been implemented as a *single-page application* (SPA). This means that unless a user switches to a different interface using the navigation bar, pages associated with individual components will not be reloaded. As mentioned in Chapter 6.3, user interactions requiring access to the server are carried out using AJAX requests.

We begin by describing the feature editor in Chapter 12 below. Functionality for editing rules is discussed in Chapter 13. Chapters 14 and 15 provide details on rule browser and rule search, respectively. Where relevant, we present detailed information on how certain functionality is implemented.

12 Feature Editor

To reduce complexity for the end user, our system separates the task of editing rules from the task of defining features and appropriate values that are used to build LHS of rules. As described in more detail below, when editing rules users have at their disposal a read-only version of all features that are currently defined[40]. To introduce new features or modify existing ones, users must navigate to the feature editor which is available on the Features tab. Figure 7 shows the feature editor in its initial state.

A list of all features that are currently defined is displayed on the left. It can be filtered dynamically by typing into the text input field at the top. Filtering is case-insensitive: In order to locate, e.g., the `Encounter` feature, both `Enc` and `enc` can be used as match strings. When trying to find matches for the current contents of the input field, the system does not anchor the search to the beginning of feature names. Using the string `en` to filter the list of features shown in Figure 7 would result in two matches, `Encounter` and `ChildGender`. To go back to the full list of features the current match string must be erased from the input field, or shortened so as to match all features that are currently defined.

A list of all values that are currently defined is displayed on the right. Just like the list of features, this list can be filtered by typing into the associated input field. However, filtering the list of values also affects the list of features, which is simultaneously narrowed down to show only those features that define one or more of the values that are currently

[40]In addition to allowing users to focus on one task at a time (which reduces the complexity of each individual task), keeping features and associated values read-only for the purpose of editing rules has another advantage: By doing away with the need for typing them in it eliminates the possibility of introducing errors through misspellings.

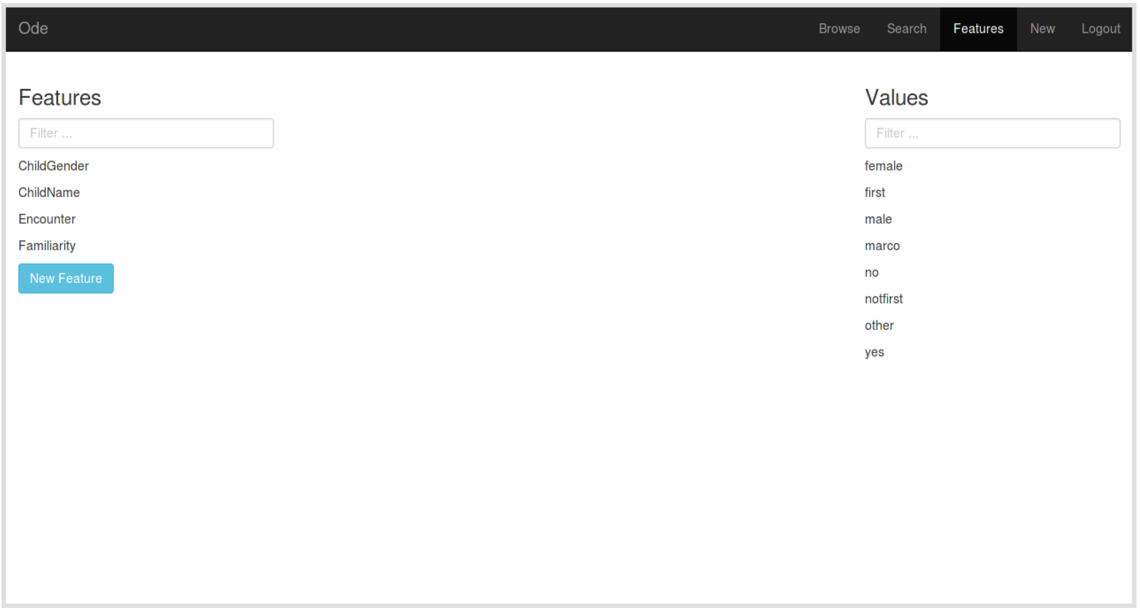


Figure 7: Feature editor in its initial state

displayed as possible targets. For example, upon entering the string `ma` the list of values shown in Figure 7 would be narrowed down to the values `female`, `male`, and `marco`. At the same time, the list of features would be narrowed down to the features `ChildGender` (whose list of possible values contains `female` and `male`) and `ChildName` (whose list of possible values contains `marco`).

12.1 Creating Features

New features can be created by filling in an appropriate form which is displayed between the list of features and the list of values. The form becomes available after clicking a button that reads New Feature. This button is located at the bottom of the list of features. Figure 8 shows the feature editor with the form for adding a new feature embedded in the center.

Submitting this form creates a new `Feature` node in the database. Values for `name`, `description`, and `type` properties are derived from corresponding fields of the form. As mentioned in Section 9.1, if no description is provided, the value of the `description` property will be set to a default value of "`...`". On the client-side, a new entry representing the newly created feature will be inserted into the global list of features to the left of the form[41]. The form itself will stay visible to allow for quick creation of additional features.

12.2 Modifying Existing Features

Description, type, and targets of a given feature can be viewed by clicking on its name in the list of features. The name of the feature that is currently selected is highlighted in the list of features. Figure 9 shows the feature editor with the `Familiarity` feature selected.

This view also allows users to rename features, change feature descriptions and types, and to add and remove targets from individual features.

[41]Feature and value lists are sorted alphabetically. This sorting is maintained when inserting new items.

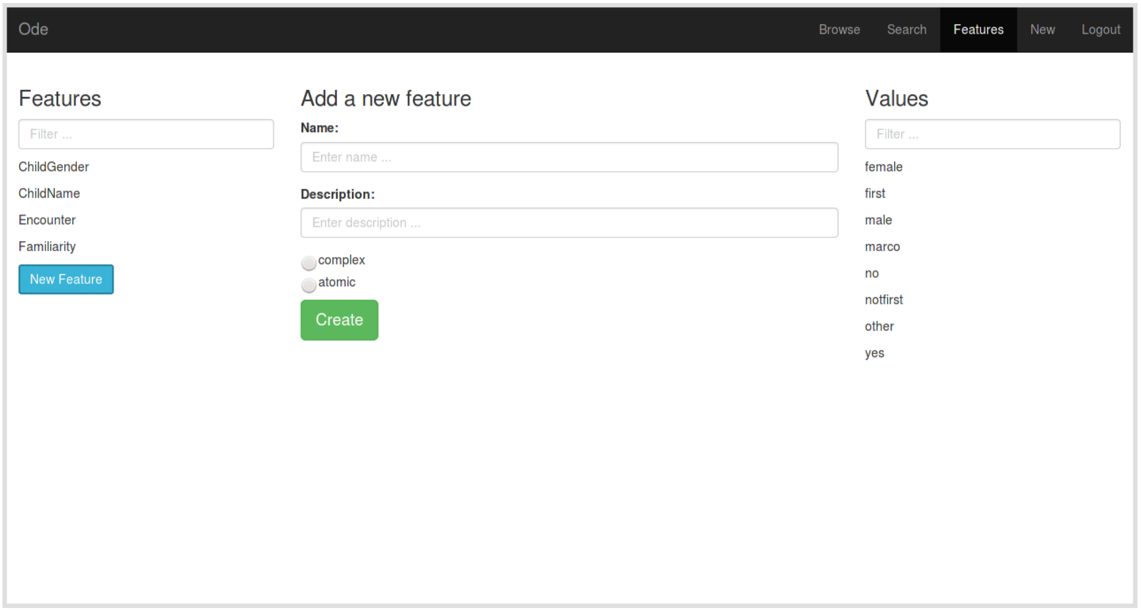


Figure 8: Form for adding a new feature

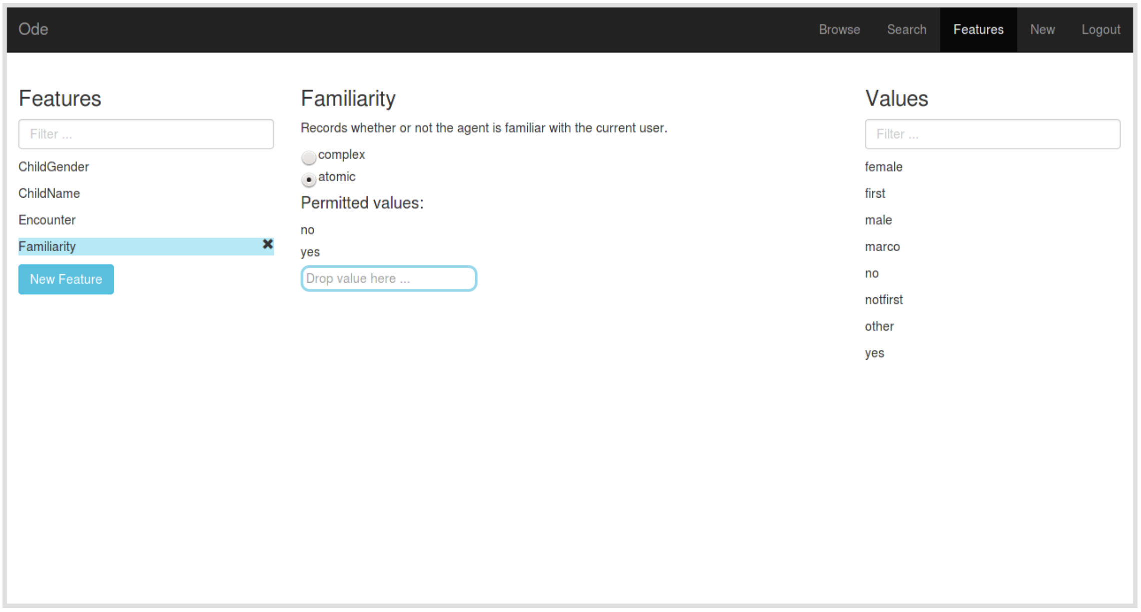
Renaming a feature is accomplished by double-clicking its name (at the top of the view), entering the new name into the text input field that replaces the read-only version of the current name, and submitting the changes by clicking an OK button that is displayed below the input field. No calls are made to the server if the input field is empty or contains the current name of the feature being edited when the OK button is clicked: In this case, the editing controls are simply replaced with the current name of the feature. The process for changing feature descriptions is analogous. To increase discoverability, contextual help is provided in the form of tooltips: When hovering over name or description of a feature with the mouse, the system informs users that they can start editing the corresponding property by double-clicking it.

The type of a feature can be changed by selecting the type that is currently unselected and confirming the change by clicking the OK button that replaces the list of possible targets. On the server-side, changing the type of a feature causes all outgoing `ALLOWS` relationships to be deleted from the database. Additionally, if the type of a feature changed from `complex` to `atomic`, an `ALLOWS` relationship is created between the feature and the `underspecified` value[42]. If a feature is in use, i.e., if it is referenced by one or more rules, the system will prevent users from changing its type.

Targets can be added to the feature that is currently displayed by dragging them to the placeholder at the bottom of the list of current targets and clicking the Add button that appears when dropping them. Placeholders for targets of *complex* features will only accept items from the feature list (and reject values), whereas placeholders for targets of *atomic* features will only accept values (and reject items from the feature list). This ensures that feature definitions conform to the data model defined in Section 9.1. If a given value to be added as a target to an atomic feature does not exist yet, it can be entered manually by clicking the placeholder and typing it in. This will cause a new value node to be created in the database.

Individual targets can be removed from a given feature by clicking the × button that appears next to them when hovering over them with the mouse. This will only work if the

[42]This is done because as explained in Section 9.1, each atomic feature is associated with this value by default.



Records whether or not the agent is familiar with the current user.

complex
[✓ atomic]

Permitted values:

no
yes

Drop value here ...

Figure 9: Viewing details of the `Familiarity` feature

feature currently being edited and the value to remove are not associated with each other in any rule. More specifically, if the feature is *atomic*, removing a target will fail if there is a `HAS` relationship between it and the target. If the feature is *complex*, removing a target will fail if the property graph stored in the database contains a path of length 2 (via an `AVM` node) between it and the target.

12.3 Deleting Features

As can be seen in Figure 9, when a feature is selected, the system highlights it in the list of features and displays an × button to the right of it. This button can be used to delete the feature that is currently selected. If a feature is referenced by one or more rules, the system will refuse to delete it.

12.4 Working with Feature Values

With respect to the data model, values are second-class citizens. As explained in Section 9.1, their lifespan is determined by the atomic features that reference them, and as mentioned above, the only way to create new values is by adding them to an existing feature. There is only one operation that can be carried out on values themselves: They can be renamed by double-clicking on them in the list of values, entering the new name into the text input field that appears, and clicking the OK button below the text field. As is the case for features, if the input field is empty or contains the current name of the value being edited when the OK button is clicked, the editing controls are simply replaced with the current name of the value, and no calls are made to the server. Changes made to value names become visible immediately in the feature editor. For instance, if the list of targets for the feature that is currently displayed contains a value named `x`, renaming that value to `y` causes the list of targets to be updated immediately.

12.5 Summary

The feature editor allows users to create, rename, and delete features, and to edit their descriptions and types. It also provides functionality for defining acceptable targets for complex and atomic features. On the other hand, the feature editor prevents any operations that would compromise the integrity of the rule base: Targets can not be removed from features if the features reference them in any rules. If a feature is in use, its type can not be changed, and it also can not be deleted. Lastly, it is not possible for users to accidentally add targets to features if they are inappropriate for the features' types.

Due to the fact that features and values are modeled as nodes and incorporated into rules by establishing appropriate relationships (instead of storing representations of LHS as properties on rule nodes), renaming features and values causes all rules to be updated at once. This ensures that features and values are used consistently across rules. Furthermore, it eliminates the need to find and manually update rules referencing specific features or values when the decision is made to rename them – a task that is cumbersome at best, and becomes increasingly time-consuming and error-prone when rule bases grow in size.

13 Rule Editor

The rule editor provides functionality for creating new rules, renaming rules and changing rule descriptions, and for editing LHS and RHS of rules. In order to provide a distraction-free editing experience, the interfaces for editing rule input and output are kept separate. Section 13.2 describes the interface for editing LHS of rules, and Section 13.3 discusses the interface for editing RHS of rules. We call these interfaces *InputBuilder* and *OutputBuilder*, respectively. Both of them provide means to rename rules and to change rule descriptions. As described in Section 13.1 below, new rules can be created from any interface.

13.1 Creating Rules

In order to create new rules, users do not have to switch to a specific interface: No matter what part of the system they are currently looking at they can click a button labeled New in the navigation bar to bring up a *modal window*[43] containing a form for creating a new rule. The form consists of two text input fields that allow users to specify a name and a description for the rule to create. Figure 10 shows the modal window its initial state.

When the Create rule button is clicked the system checks the form for errors. For instance, the **Name** field might have been left empty, or its contents might match the name of an existing rule. In these cases the modal window stays visible and the system displays an appropriate error message. By rejecting creation of rules with inappropriate values for the `name` property we ensure that newly created rules conform to the data model described in Section 9.2. If there are no errors, the modal window is closed and the user is redirected to the interface for editing the LHS of the newly created rule.

[43]In the context of web application development, modal windows provide a way to display or prompt users for information without having to use pop-up windows or page reloads. They are also known as *modal dialogs* or *modal boxes*. After invoking a modal window, control is transferred to it, i.e., the parent window from which the modal was invoked can not be interacted with anymore [1]. Control returns to the parent window as soon as the user performs an action that causes the modal to be closed. Our system uses the Twitter Bootstrap implementation of modal windows (cf. `http://getbootstrap.com/javascript/#modals`).

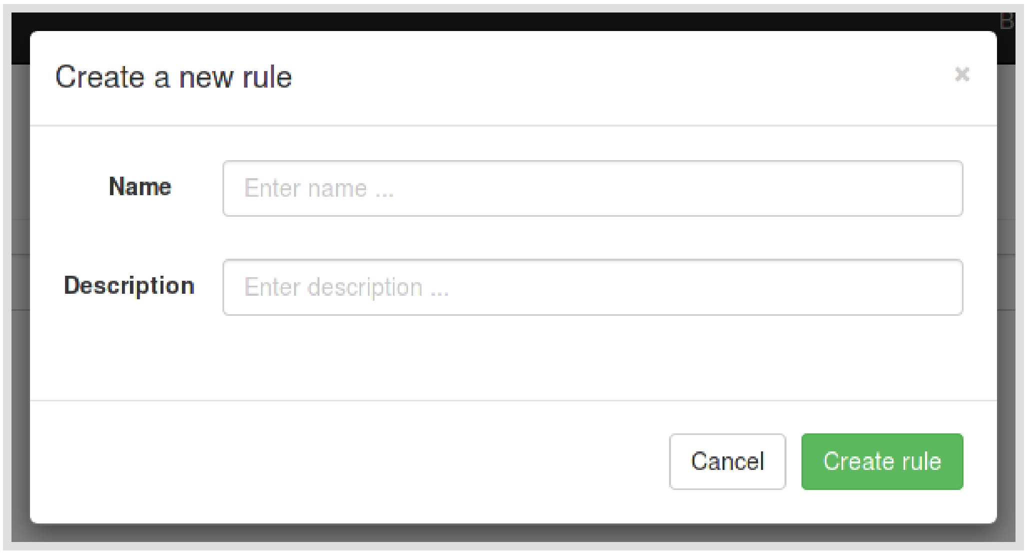


Figure 10: Modal window for creating a new rule

Rule creation can be canceled by clicking Cancel or by clicking the × button in the upper right corner of the modal window. This causes the modal window to disappear, returning control to the interface that was visible when the New button was clicked.

13.2 InputBuilder

The interface for editing rule input provides functionality for adding features, setting features to specific values, and for removing features from the LHS of the current rule. Figure 11 shows the InputBuilder in its initial state.

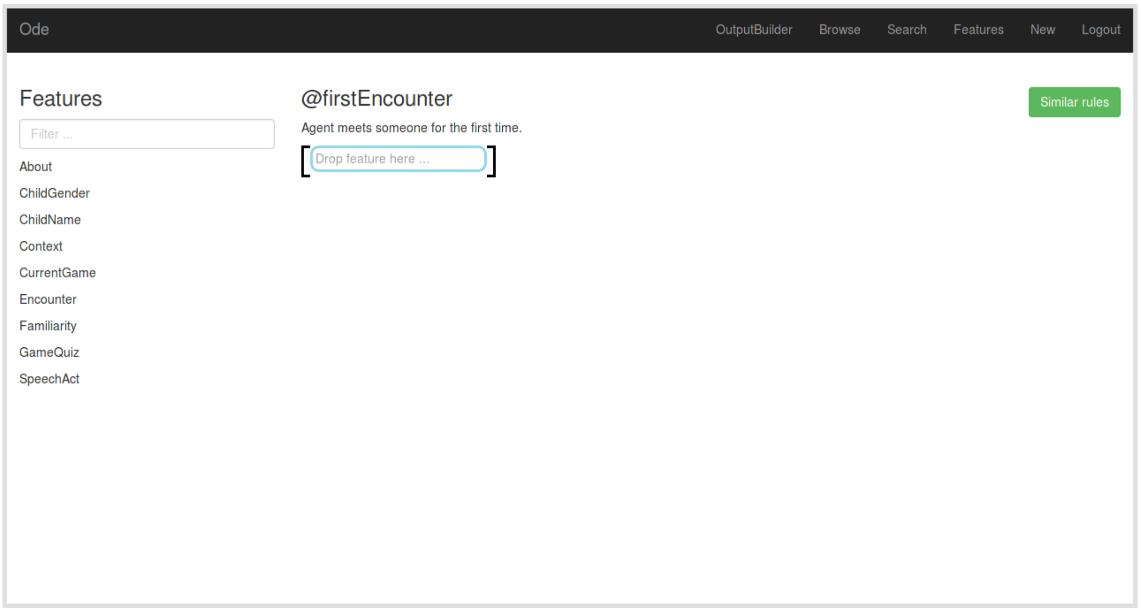


Figure 11: InputBuilder in its initial state, showing an empty AVM

In the context of the InputBuilder we call the left column under the **Features** heading *feature inventory*. It lists all features that are currently defined in alphabetical order, and exhibits the same behavior as the list of features displayed in the feature editor when filtered via the text input field at the top (cf. Chapter 12). When hovering over a specific feature with the mouse, a tooltip containing the associated description is displayed after a short delay. Provided that a proper definition exists for the feature, this keeps users from having to look it up in the feature editor if they need more information to decide whether the feature is appropriate for the current rule.

Name and description of the current rule as well as its LHS are displayed to the right of the feature inventory[44]. If the LHS of a rule is empty because no features have been added to it, its visual representation consists of two square brackets and a single placeholder for adding features (cf. Figure 11).

Editing the LHS of a rule does not involve any typing. To add a feature to an LHS, users need to click on it in the inventory, drag it to the AVM and drop it on a placeholder. As can be seen in Figure 12 below, nested AVMs are identical to the root AVM structurally and therefore have separate placeholders[45]. Top-level AVMs accept any feature from the inventory. Nested AVMs, on the other hand, will reject features if they have not been defined as acceptable targets for the features that embed them via the feature editor. To set an atomic feature to a specific value, users can select that value from the drop-down menu displayed to the right of the feature. The drop-down menu associated with a feature contains a separate entry for each value that has been defined as an acceptable target via the feature editor. Values that are not allowed are omitted from the menu. Because rules can demand the presence of specific features while leaving their values underspecified, the menu also contains an entry for the `underspecified` value. Additionally, when an atomic feature is first added to an AVM, the system sets its value to `underspecified` by default. This saves users an additional editing step if they are not interested in specifying an actual value for a given feature.

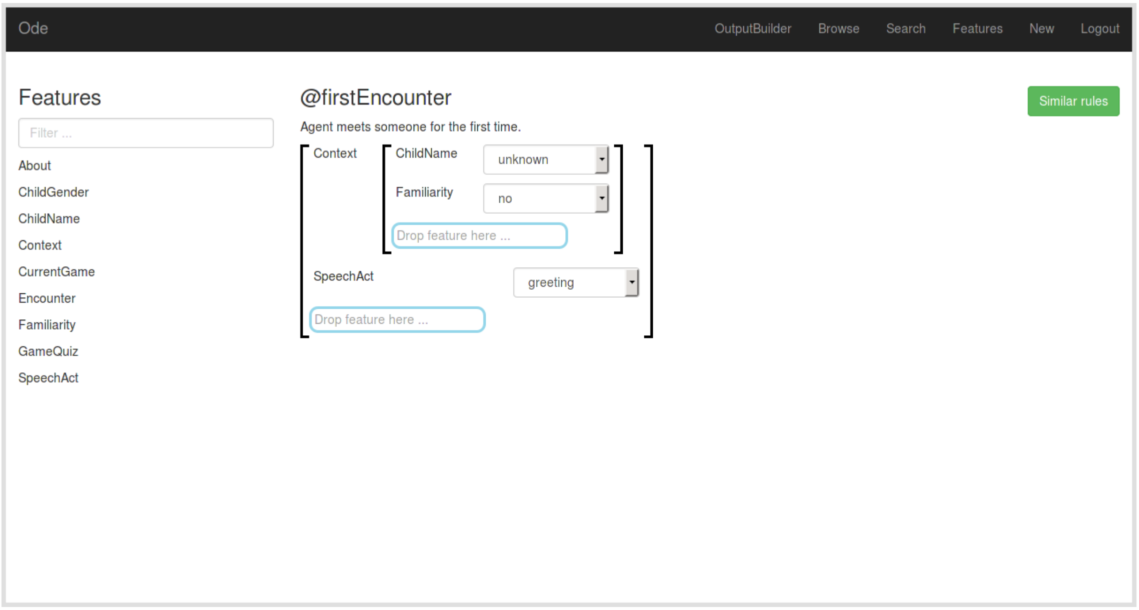


Figure 12: InputBuilder showing an LHS with a nested AVM and a total of four features

[44]The name of the rule displayed in Figure 11 is `@firstEncounter`, and its description is `Agent meets someone for the first time.`

[45]As of this writing, features can not be moved *within* an LHS. Consequently, if a user chooses the wrong AVM for a given feature, they must remove it from the LHS and re-add it to the correct one.

Features can be removed from the LHS of a rule one by one or in bulk: To remove a single feature from its parent AVM users can click the × button that appears when hovering over the name of the feature in the AVM. If users need to make more substantial changes to the structure of an LHS, they can also *empty* individual AVMs, i.e., they can remove all features from them at once. This is achieved by clicking a button shaped like an empty rectangle that appears when hovering over an AVM with the mouse. Figure 13 shows a close-up of the LHS displayed in Figure 12 in which both types of buttons for removing features are visible.

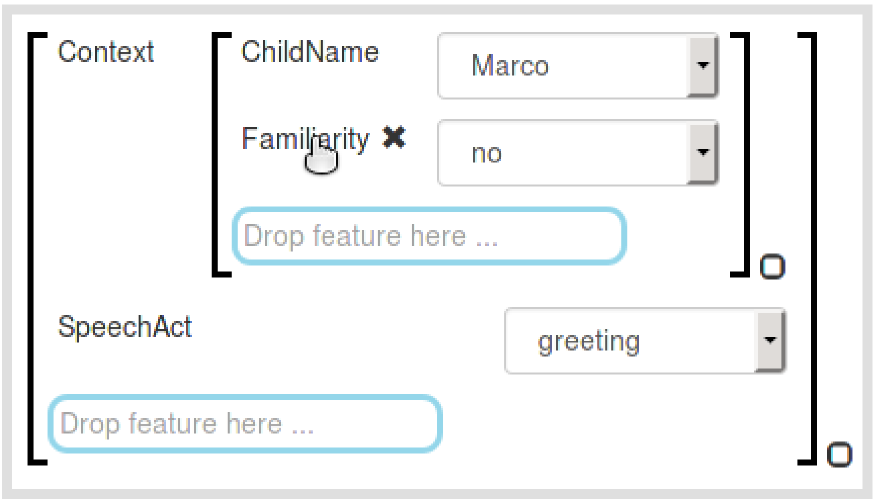


Figure 13: Visual representation of an LHS. The mouse cursor points to the name of a feature (`Familiarity`) belonging to a nested AVM. As a result, the system shows a button for removing this feature, as well as two buttons for emptying the nested and top-level AVMs.

In addition to the LHS-specific features described above, the InputBuilder provides means to change name and description of a rule. From a user's point of view, the process of renaming a rule is almost identical to the process of changing its description: In each case, the attribute to change must be double-clicked. This causes a text input field and an OK button to appear in place of the attribute. Users can then enter the new name or description into the field and click the OK button to make the change permanent[46]. To avoid unnecessary AJAX calls to the server, the contents of the input field are checked for two conditions on the client-side: If the input field is empty or contains a string that is identical to the current value of the attribute being edited when the OK button is clicked, the system simply replaces the editing controls with the current value of the attribute, and refrains from contacting the server. From the system's point of view the difference between renaming rules and changing their descriptions is that in the former case, the resource URL of the rule being edited changes, making it necessary to redirect users to the updated URL after a successful rename.

Aside from the interfaces that are always available in the navigation bar, the InputBuilder is connected to two additional interfaces: A button located in the upper right corner of the editing area allows users to view a list of rules that are similar to the current rule in the rule browser[47]. (Section 14.2 describes the notion of rule similarity we adopt in detail, and

[46]Note that the input field initially contains the current value of the attribute being edited. This is especially useful for situations in which users only want to make small changes to the value of an attribute: With an empty field they would either have to remember copying the original value before starting to edit (in order to be able to paste it into the input field), or be forced to manually re-enter portions of the original value which they would like to keep.

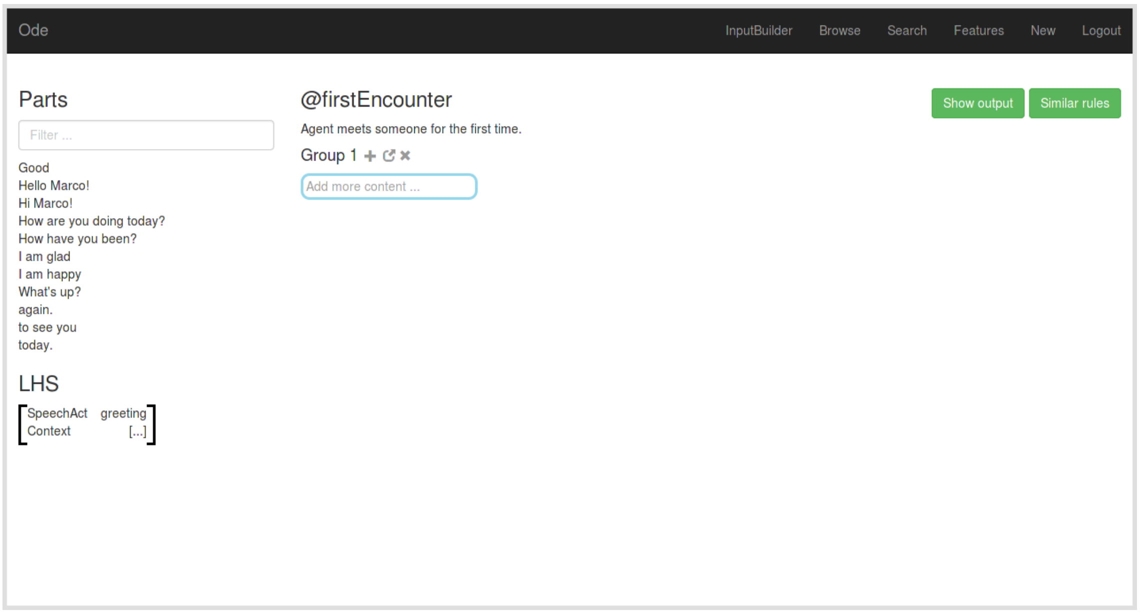
[47]This button is also present in the editing area for rule output (cf. Figure 14).

also presents the steps that the system takes to compute sets of similar rules.) Additionally and most importantly, while editing rule input users can switch to the interface for editing rule output using the OutputBuilder button made available in the navigation bar.

Finally, it is important to note that all changes made to the LHS of a rule in the InputBuilder are persisted to the database as they occur. There is no need to explicitly "save" a set of changes before switching to another interface. However, changes to rule names and descriptions are lost unless they are made permanent as described above. In other words, if a user modifies the value of one of these attributes and navigates away from the InputBuilder before clicking the OK button, the edit is aborted and no changes are made to the value stored in the database.

13.3 OutputBuilder

In terms of high-level organization of interface components, the OutputBuilder is similar but not identical to the interface for editing rule input described above. It consists of a *parts inventory*, an abbreviated, read-only view of the LHS of the current rule[48], and an editing area. The **LHS** view is included for reference purposes: Together with rule name and description, the presence of this view is supposed to minimize the need for switching back and forth between InputBuilder and OutputBuilder while designing rule output. The purpose of the parts inventory is to speed up the process of designing rule output by facilitating reuse of parts between rules. It displays all parts that are currently defined, and can be filtered dynamically by typing into the text input field at the top. Parts inventory and **LHS** view are located in the same area of the screen that is occupied by the feature inventory in the InputBuilder, and the editing area for rule output replaces the editing area for the LHS of the current rule. Figure 14 shows the OutputBuilder in its initial state.



Parts

Good
Hello Marco!
Hi Marco!
How are you doing today?
How have you been?
I am glad
I am happy
What's up?
again.
to see you
today.

LHS

SpeechAct	greeting
Context	[...]

@firstEncounter

Agent meets someone for the first time.

Group 1

Figure 14: OutputBuilder in its initial state, showing a single combination group

[48]Depending on the size and complexity of a given LHS, including a full representation of it might take up too much space in the OutputBuilder. In order to avoid this, only top-level features and associated string values are displayed in full. Immediate substructures of LHS are *ellipsized* to "`[...]`".

The OutputBuilder supports several different methods for building up the desired output for a given rule which can be combined as necessary. On the one hand, this allows for a gentler learning curve for new users, as they are not forced to learn how to use more advanced features for editing output right away. On the other hand, users who *are* aware of different options for working with rule output can flexibly choose a workflow that is most suitable for the content they need to create. In the following sections we describe in detail how our system can be used to add, modify, and remove content from RHS of rules. Features for dealing with output strings are described in Section 13.3.1. Section 13.3.2 details how to work with parts, and Section 13.3.3 describes the purpose of using multiple combination groups.

13.3.1 Working with Output Strings

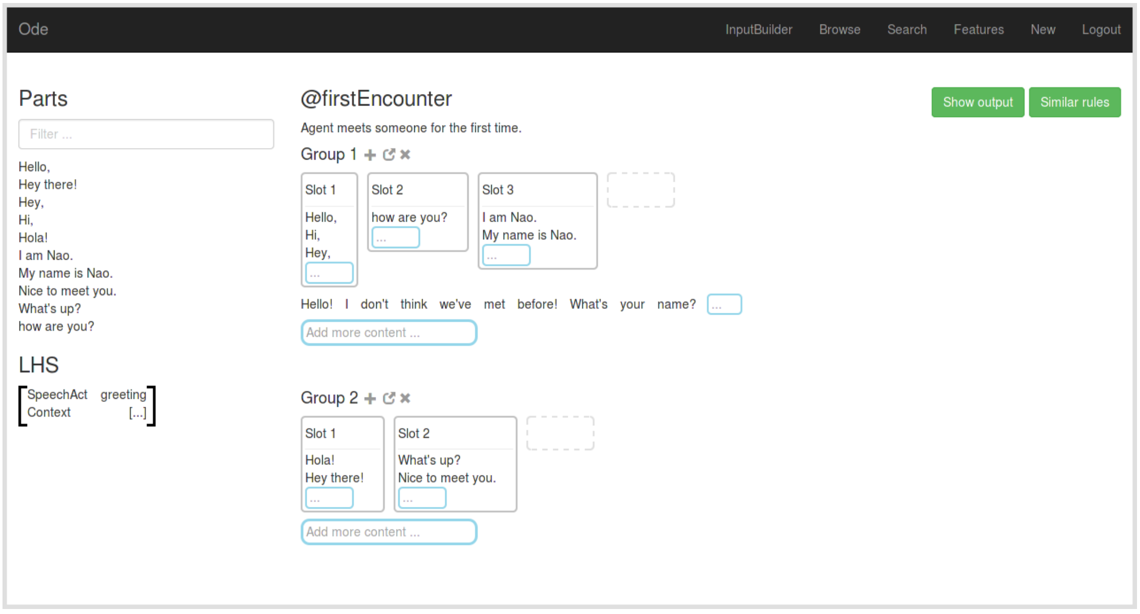
As can be seen in Figure 14, the OutputBuilder initially displays a single combination group without any slots. The most basic way to populate this group with content is to manually add one or more output strings. This is done by using the (multi-purpose) placeholder displayed at the bottom of the group: To add a single output string, users must click the placeholder, enter the desired contents, and click the Add button that becomes available below the placeholder after they start typing. If the operation is successful, the output string is placed above the placeholder, and the placeholder itself is reset to its initial state[49]. Sticking to this approach is most suitable for rules that introduce completely novel content (i.e., rules for which there is no or very little appropriate content to be drawn from the parts inventory) and/or require only a small number of output alternatives. Additionally, note that this method of adding content to RHS of rules benefits novice users in two ways: First of all, it does not require them to understand the concepts of parts and slots, enabling them to become productive with the system very quickly. Secondly, it allows them to focus exclusively on creating appropriate output for the situation covered by the current rule.

Once they have been added to a combination group, individual output strings can be modified by double-clicking them: This causes their contents to be displayed in an editable text input field, along with an OK button that users must click to finalize the changes they made to the original string. It is important to note that changing the contents of a given output string does *not* cause all combination groups referencing the same string to be updated. Instead, the node representing the parent group is connected to an output string node with the updated content. If no such node exists, it is created first. Individual output strings can be removed by clicking the × button that appears when hovering over them with the mouse.

In addition to adding output strings manually as described above, users can also assemble them using parts available in the parts inventory. To add a part from the parts inventory as an output string, they can drop the part on the same placeholder that accepts text input for manual specification of output strings[50]. If this results in an output string that is incomplete (after all, parts are *substrings* of full output strings and do not necessarily constitute grammatical sentences) or lacking in other ways, users can either modify the output string as described above or *extend* it. This is done by dropping individual parts from the parts inventory on the placeholder displayed to the right of the string in question. Depending on the availability of appropriate parts, assembling output strings in this manner can reduce the amount of typing involved in creating output for a given rule considerably.

[49]Placeholders for other types of operations behave identically; they also reset themselves to their initial state when the operation currently performed on them completes.

[50]Note that in this case, the system does not display an Add button for users to click: After dropping a part on a placeholder, it is added to the list of output strings for the current group automatically.



Parts

Hello,
Hey there!
Hey,
Hi,
Hola!
I am Nao.
My name is Nao.
Nice to meet you.
What's up?
how are you?

LHS

SpeechAct greeting
Context [...]

@firstEncounter

Agent meets someone for the first time.

Group 1

Slot 1	Slot 2	Slot 3
Hello,	how are you?	I am Nao.
Hi,	...	My name is Nao.
Hey,		...
...		

Hello! I don't think we've met before! What's your name? ...

Group 2

Slot 1	Slot 2
Hola!	What's up?
Hey there!	Nice to meet you.
...	...

Figure 15: OutputBuilder showing an RHS with two combination groups

13.3.2 Working with Parts

Since one of the main goals of the system presented here is to reduce the amount of work required for building rules with large numbers of alternative verbalizations, the Output-Builder supports another type of workflow for populating combination groups associated with RHS of rules. After adding one or more output strings using the methods described above, users can *split* individual strings into parts by clicking the area between the last word that is supposed to belong to the left part and the first word that is supposed to belong to the right part. Parts generated by splitting output strings are added to appropriate slots which are displayed above the list of output strings defined for the current group: The left part is added to **Slot 1** and the right part is added to **Slot 2** (cf. Figure 15). If no slots exist, i.e., if no other output string belonging to the same combination group has been split before, they are created and connected to the appropriate group first. The original output strings are deleted from the combination group that is currently being edited.

As soon as the two default slots have been created, users can add parts to them in several ways: The most basic way to add a part to a slot is by entering it into the placeholder at the bottom of the slot. The steps involved are very similar to the steps for manually adding output strings to combination groups: After clicking a placeholder, users can type in the contents of the part they wish to add to the corresponding slot. The operation is finalized by pressing Return. Alternatively, users can add existing parts by dragging them from the parts inventory and dropping them on the placeholder of the target slot[51]. Finally, as hinted at above it is also possible to add parts to existing slots by splitting additional output strings. Parts that are created manually or by splitting output strings are added to the parts inventory automatically. Note, however, that this only happens for parts that are actually *new*. As mentioned in Section 9.2, parts must be unique with respect to their content, and as a result, the parts inventory will never contain any duplicates. Another benefit of making the system responsible for eliminating redundancy is that this allows users to focus exclusively on *what* content to add instead of *how* to add it. If they prefer

[51]Note that in this case it is not necessary to press Return in order to finalize the operation: Parts are added to slots automatically after dropping them on corresponding placeholders.

to check the parts inventory for appropriate content first, they are welcome to do so, but if they deem it best to create content manually, adding an existing part (or output string) will not lead to increased storage consumption.

Just like output strings, parts can be modified after they have been added to a specific slot. The process, again, is similar to the process for modifying output strings: After double-clicking a part, its contents are placed in an editable text input field that can be used to make the desired changes. Pressing Return finalizes the operation. Note that changing the contents of a part does *not* cause all slots referencing this part to be updated. Instead, the node representing the parent slot is connected to a part node with the updated content, which is created first if it does not exist.

To remove a part from its parent slot, users can click the × button that appears when hovering over it with the mouse. If the parent slot was the only slot referencing this particular part, this operation turns the part into an orphan. In this case, the part will be removed from the database as well. Note, however, that the complete removal of a part from the database will *not* cause the parts inventory to be updated right away. That is, the orphaned part will stay available in the parts inventory until the current user either navigates to a different interface or refreshes the page. This type of *lazy* behavior takes into account the possibility that users might want to reuse deleted parts for other slots and/or output strings in the context of the same editing session.

If the default number of slots does not provide enough flexibility for adding content to the current combination group, users can add an arbitrary number of additional slots by clicking a placeholder for new slots that is displayed next to the rightmost slot. If a group contains more than two slots, any slot n can be removed by clicking the × button that appears when hovering over its header (**Slot** n) with the mouse. Slot removal is restricted to groups with more than two slots because with *less* than two slots the ability to combine parts to form full output strings is lost.

In addition to parts, slots can also contain cross-references to other rules, allowing entire sets of output strings to be reused as parts: When exporting a set of rules via the rule browser, each output string of a cross-referenced rule is treated as a part belonging to the slot that contains the cross-reference. Section 14.3 describes how users can initiate the export process, and explains individual steps of this process in detail. Cross-references between slots and rules are established by adding names of rules as parts. In order for the system to recognize a part as a cross-reference, it must be preceded by an `@` symbol. After a cross-reference has been added to a slot successfully it can be clicked to jump to the OutputBuilder for the corresponding rule, making it possible to quickly review the output that this rule generates[52]. In sum, cross-references allow users to modularize the rule base – at least to a certain extent – and to reduce explicit duplication of content across rules by factoring out entire sets of commonly used phrases into separate rules.

At any given point during the editing process, users can view a list of all output strings that can be generated from the contents of existing slots by clicking a button labeled Show output located in the top right corner of the interface. This list also contains any regular output strings belonging to the RHS of the current rule. It is identical to the list of output strings displayed on the *details page* of the rule being edited, which will be described in Section 14.1 below. In the context of editing RHS of rules, the main purpose of viewing lists of full output strings is to check output strings generated from slots for grammatical errors, but it also allows users to get a sense of the amount of variation that is available for a given rule without having to calculate the number of possible combinations. Note that in order to keep the system responsive, cross-references are not taken into account when generating the list of output strings associated with a given rule. A detailed description of the steps involved in generating lists of full output strings can be found in Appendix B.1.

[52]Note that it is currently not possible to *edit* cross-references. The only way to change a cross-reference is to remove it and add a new cross-reference pointing to a different rule.

Finally, as described in Chapter 8, native rule syntax supports the use of variables for capturing values associated with features and referencing them from other locations within the same rule. This is useful for cases in which the value of a feature needs to be incorporated into the verbal output that an agent should produce, but only becomes available at run-time. With the current version of our system, it *is* possible to create rules whose output incorporates values of variables introduced by corresponding LHS, but the process involves a number of steps: First, a string representing a variable has to be added as a target to the atomic feature whose value is to be captured at run-time. Since feature values must start with a hash symbol (`#`) to be treated as variables (cf. Chapter 8), it is important to prefix the string with this symbol in this step. Secondly, in order to specify that the value of the corresponding feature should be captured dynamically in the context of a given rule, users need to select the variable from the drop-down menu displayed next to the feature in the InputBuilder. The variable can then be referenced from the corresponding RHS by adding it as a part to one or more appropriate slots. After exporting a set of rules to plain text as described in Section 14.3, RHS of rules incorporating variables must be post-edited to remove quotation marks from parts representing variable references. This is necessary because when building plain text representations of rules, the exporter does not distinguish between literal content and variables. That is, all parts are enclosed in quotation marks when exporting rules to native rule syntax (cf. Appendix B.3), and will therefore be treated as regular strings by the rewriting engine. Furthermore, if variables are embedded into larger parts or output strings, additional processing steps are required to ensure that variable references can be resolved successfully by the rewriting engine. Clearly, this workflow is far from optimal: It presumes familiarity with the way rules are serialized to strings during export, and involves a number of steps that can easily lead to errors. As a result, specialized editing support will have to be implemented for this use case in future versions of the system.

13.3.3 Working with Multiple Combination Groups

Depending on the desired number of output strings and the amount of variety found in verbalizations that are appropriate for a given rule, a single combination group might not allow the output of the rule to be entered efficiently. As an example, consider the following set of strings:

```
I am glad to see you again.
I am glad to see you today.
I am happy to see you again.
I am happy to see you today.
Good to see you again.
Good to see you today.
```

In order to keep the amount of typing involved in adding these strings to the output of a rule to a minimum, they should ideally be split up as follows:

`I am happy`	`to see you`	`again.`
`I am glad`		`today.`
`Good`		

Table 2: Ideal split

Let us further assume that we wanted to add these strings to the output of a rule containing the following combination group:

`Hello Marco!`	`How have you been?`
`Hi Marco!`	`How are you doing today?`
	`What's up?`

Table 3: Group 1

While it would be possible to stick to using just this group by adding each string as a part to the second slot *as is*, there is no way to add the strings to this group that would preserve the ideal split shown above *and* result in error-free output. For instance, extending the group as follows preserves the ideal split:

`Hello Marco!`	`How have you been?`	`to see you`	`again.`
`Hi Marco!`	`How are you doing today?`		`today.`
	`What's up?`		
	`I am happy`		
	`I am glad`		
	`Good`		

Table 4: Group 1 (modified)

But the rule would be unusable because its output would include a number of ungrammatical strings:

```
Hello Marco! How have you been? to see you again.
Hello Marco! How have you been? to see you today.
Hello Marco! How are you doing today? to see you again.
Hello Marco! How are you doing today? to see you today.
...
```

Using an additional group, however, the strings can be combined with the contents of the first slot of the original group *and* be split as shown above:

`Hello Marco!`	`I am happy`	`to see you`	`again.`
`Hi Marco!`	`I am glad`		`today.`
	`Good`		

Table 5: Group 2

To support cases like this, the OutputBuilder provides functionality for adding an arbitrary number of additional combination groups to the RHS of a single rule. Each group can be populated using whatever workflow requires minimal effort. It is also possible to create new groups by *copying* existing groups[53]. This facilitates using the contents of an existing group as a starting point for a new group: In the example above, the first slot of the second group is identical to the first slot of the first group. Because of this, the most convenient way to create the second group would be to copy the first group and post-edit it to achieve the final result. Compared to this, the naive approach of creating an empty

[53]Note that this currently only works in the context of a single rule. It is not possible to import groups by copying them from other rules.

group, adding `Hi Marco! I am happy` as an output string, splitting it, and then filling in the remaining parts is less efficient.

Available controls for adding, copying, and removing groups are located in the header of each group: Clicking the leftmost, plus-shaped button inserts an empty group below the current group. The middle button creates a new group below the current group and copies the contents of the current group to the new group on click. Finally, the × button displayed to the right of the buttons for adding and copying groups allows users to delete the corresponding group. To avoid confusion, the order of groups is preserved between visits to the OutputBuilder. As mentioned in Section 9.2, each combination group has a `position` property which is used to determine its position in relation to other groups belonging to the same RHS. Every time a group is added or deleted, the positions of all groups following it are incremented or decremented, respectively.

13.4 Summary

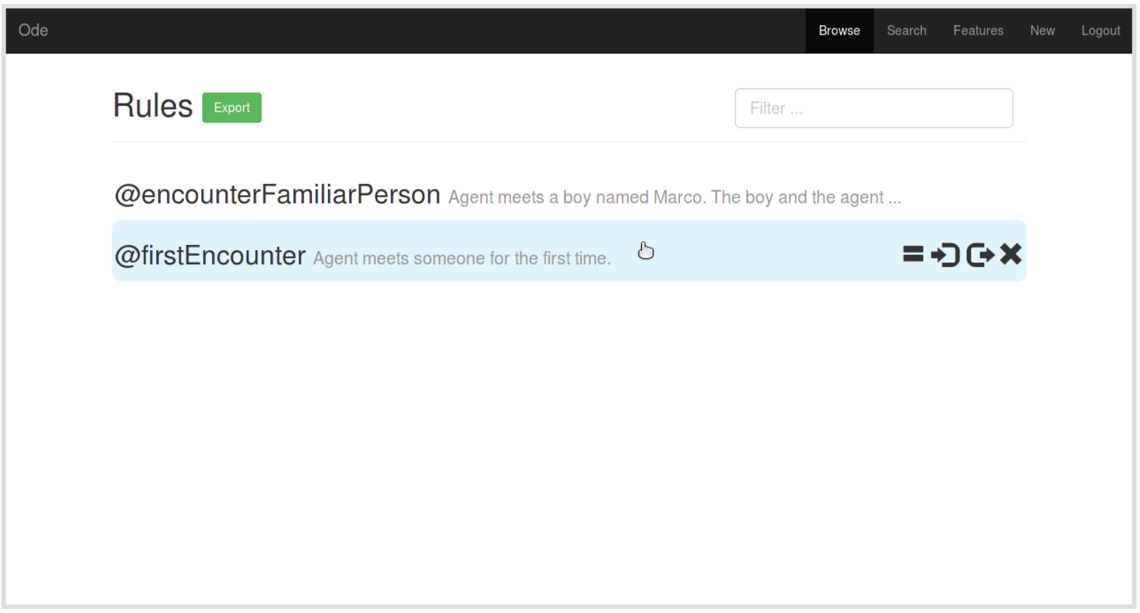
This chapter has described in detail how our system supports users in creating and editing rewriting rules. The InputBuilder allows users to modify LHS of rules without any typing. This eliminates the possibility of introducing errors through misspellings of feature and value names. Menus for setting values of atomic features only contain values that have been defined as appropriate via the feature editor, and nested AVMs only accept features that have been defined as acceptable targets for the features that embed them. By setting appropriate default values for features as they are added to a given AVM, the system ensures that atomic features can not be used as complex features and vice versa. Modifications to LHS of rules are persisted to the database as they occur, which means that users can safely navigate to another interface at any point without losing their work.

The OutputBuilder allows users to add, modify, and remove content from RHS of rules. It supports multiple workflows for defining rule output, allowing users to tailor their approach to the type and amount of content they intend to create. Novice users are not forced to learn how to use advanced editing features right away: They can stick to using output strings until they become familiar with parts, slots, and the purpose of using multiple combination groups. Furthermore, for the purpose of speeding up creation of rule output, the OutputBuilder allows content to be reused in various ways: In the context of a single rule, combination groups can be copied to reuse their content as a starting point for creating additional verbalizations. Secondly, the parts inventory facilitates reuse of content for parts and output strings across rules. Lastly, by establishing cross-references users can incorporate output defined by other rules into specific slots. Together with the possibility to split up output strings into parts (and to use multiple combination groups to enter output efficiently if necessary), these features can considerably reduce the amount of typing involved in creating rule output. Finally, functionality for viewing full sets of output strings associated with individual rules allows users to ensure that specific configurations of parts and slots produce valid output.

Both InputBuilder and OutputBuilder provide functionality for changing rule names and descriptions, and allow users to view a list of rules that are similar to the rule they are currently editing. Functionality for creating rules is available from any interface, and can be invoked with a single click. This eliminates the overhead of having to navigate to a specific interface (or traversing a complex menu) for the purpose of creating rules.

14 Rule Browser

The purpose of the rule browser is to provide a high-level overview of the rule base in order to make it easy to examine the current state of the rule base and identify rules to work on next. When accessing the rule browser via the Browse tab in the navigation bar, users are presented with a list of existing rules, sorted alphabetically by name. Each list entry consists of the name of the corresponding rule followed by its description, and can be clicked to jump to a page showing detailed, read-only information about the associated rule. Figure 16 shows the rule browser in its initial state.



@encounterFamiliarPerson Agent meets a boy named Marco. The boy and the agent ...

@firstEncounter Agent meets someone for the first time.

Figure 16: Rule browser in its initial state

The list of rules can be filtered dynamically by typing into the text input field located at the top. Both rule names and descriptions are taken into account when matching rules against search strings. Filtering behavior for rules is identical to filtering behavior for features and values provided by the feature editor: Case is ignored and matches can occur anywhere inside rule names and descriptions.

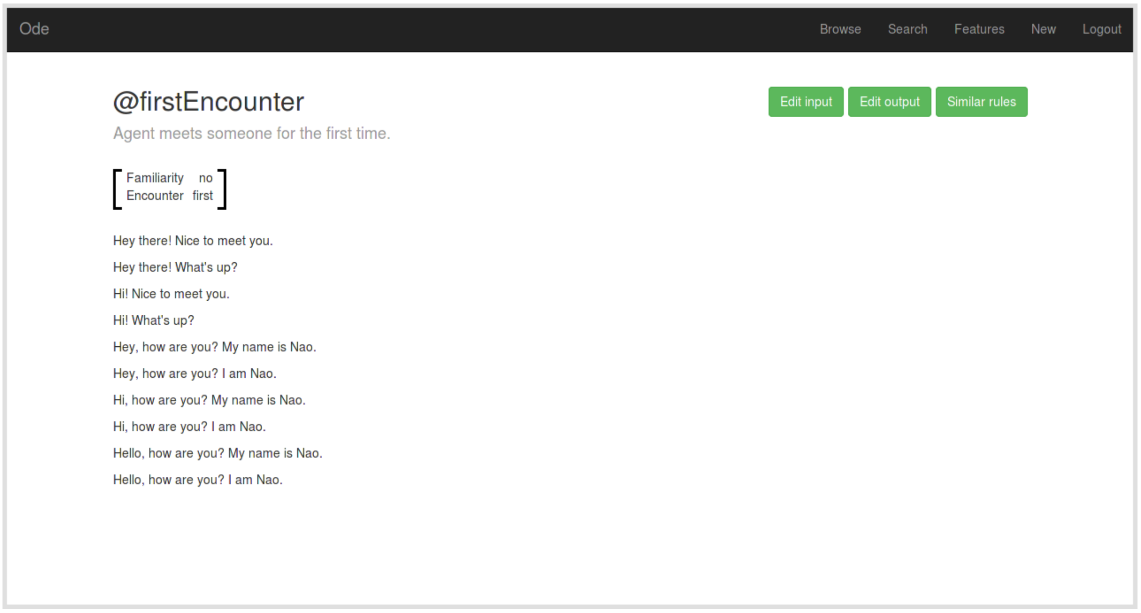
Each entry in the list of rules provides a set of controls that are shown when hovering over the entry with the mouse. From left to right, these controls allow users to

- show a list of similar rules (see Section 14.2 below)
- jump to the interface for editing rule *input*
- jump to the interface for editing rule *output*
- delete the corresponding rule.

A single click on a control causes the system to carry out the operation that is associated with it. If a rule is in use, i.e., if it is cross-referenced by the RHS of one or more rules, the system will refuse to delete it and display an appropriate error message. To improve discoverability of features provided by the rule browser, each control has a tooltip associated with it that explains its functionality and appears when hovering over the corresponding control with the mouse.

14.1 Viewing Detailed Information about Rules

As mentioned above, users can navigate to a page showing detailed information about a given rule by clicking on the corresponding entry in the global list of rules displayed in the rule browser. In addition to rule name and description, this page shows the AVM representing the LHS of the rule, as well as a complete list of output strings associated with the RHS of the rule. This list is generated as described in Appendix B.1 below. The details page also provides controls for showing similar rules and jumping to the editing interfaces for rule input and output, making it unnecessary for users to go back to the rule browser to carry out these operations. Figure 17 depicts the details page for a rule named `@firstEncounter`.



@firstEncounter

Agent meets someone for the first time.

$\begin{bmatrix} \text{Familiarity} & \text{no} \\ \text{Encounter} & \text{first} \end{bmatrix}$

Hey there! Nice to meet you.
Hey there! What's up?
Hi! Nice to meet you.
Hi! What's up?
Hey, how are you? My name is Nao.
Hey, how are you? I am Nao.
Hi, how are you? My name is Nao.
Hi, how are you? I am Nao.
Hello, how are you? My name is Nao.
Hello, how are you? I am Nao.

Figure 17: Details page for the `@firstEncounter` rule

Showing rule details in a separate view brings with it the advantage of being able to show exhaustive information without cluttering the interface of the rule browser, and avoids slowing down page load times[54].

14.2 Viewing Sets of Similar Rules

Based on experience collected in the context of creating rules for the ALIZ-E project, checking rule coverage in order to identify what to work on next is an integral part of the workflow. Provided that rules are aptly named and/or have meaningful descriptions, users should be able to identify whether or not a rule base provides support for a specific scenario by browsing and filtering the rule base as described above, even if the rule base is fairly large. However, starting from a specific rule users might also need to check rule

[54]Note that there are other approaches to ensuring short page load times for the rule browser such as pagination and lazy loading of rule details via AJAX requests. In the latter case, list entries could be set up to expand on click to show LHS and RHS of the rules they represent.

coverage for the scenario to which this rule belongs. For instance, given a rule covering a situation in which an agent is supposed to give a user positive feedback because they performed well on a quiz game, a rule developer might want to see what other rules for providing feedback (for, e.g., bad or mediocre performances) have already been defined. On a technical level, this means finding rules which are similar to the original rule in the sense that they incorporate the same set of features, or possibly a superset. When working with plain text representations of rewriting rules, this task grows progressively harder as rule bases increase in size, and fairly quickly becomes impossible to perform in a reasonable amount of time. Our system addresses this issue by allowing users to narrow down the list of rules displayed in the rule browser to show only those rules that are related to a specific rule according to the notion of similarity described above[55]. For any given rule, this list always includes the rule itself. As mentioned above, both the rule browser and the details page include controls to jump to the list of rules that are similar to the current rule. Additionally, users can access this functionality from both the InputBuilder and the OutputBuilder, which also display the necessary controls (cf. Chapter 13). Including these controls in a variety of interfaces is useful because the need to check rule coverage is not necessarily tied to carrying out a specific task. Appendix B.2 describes the algorithm our system uses to compute sets of similar rules.

14.3 Rule Export

As mentioned in Chapter 6.4, our system integrates with other tools developed in the context of the ALIZ-E project by providing functionality for exporting rules to native rule syntax. To trigger the export process, users can click the Export button at the top of the list of rules displayed in the rule browser (cf. Figure 16). The system then produces plain text representations of all rules that are currently displayed, and offers the results as a `.trf` file for download. More specifically, the default name for this file is `rules.trf`, but users can enter a different name before accepting the download via a standard dialog window for saving files. As this aspect of the export process is handled by the browser, however, the exact behavior will vary slightly depending on user-specific browser configurations. `.trf` files produced by the system include rule names and descriptions, which facilitates the task of locating specific rules using search functionality provided by standard text editors. Furthermore, since users might want to specify LHS and/or RHS of some rules manually, rules are included in `.trf` files even if their definitions are incomplete. To ensure that rule files produced by the exporter are immediately usable by the rewriting engine, the bodies of incomplete rules are commented out.

It is also possible to export subsets of rules: If the list of rules is narrowed to a subset of rules that are similar to a given rule when the Export button is clicked, only that subset will be exported. With the current implementation of the export algorithm (described in detail in Appendix B.3), there is only one condition that must be satisfied for this to work: Rules belonging to the subset to export must not cross-reference any rules that are not part of this subset. Support for exporting subsets of rules whose names and/or descriptions match strings entered into the input field at the top of the list has not been implemented yet. That is, the system will export all rules that were displayed *before* the list of rules was filtered according to the contents of the input field.

[55]Note that depending on long-term feedback, the current set of criteria determining what it means for two rules to be similar might change in future versions of the system.

14.4 Summary

The rule browser described in this chapter provides a compact overview of all rules that are currently defined, and allows users to filter rules by name and description. It also serves as an entry point for viewing and operating on rules: Clicking an entry in the list of rules takes users to a page showing detailed information about the corresponding rule. Furthermore, each entry has controls for jumping to the editing interfaces for rule input and output, and for deleting the rule it represents. If a rule is cross-referenced by one or more rules, the rule browser will prevent it from being deleted in order to preserve integrity of the rule base. To facilitate the task of checking rule coverage for specific scenarios, the rule browser provides functionality for viewing sets of rules that are similar to a given rule with a single click.

In addition to allowing users to examine a rule base in various ways, launch editing interfaces for rule input and output, and delete individual rules, the rule browser also exposes functionality for exporting rules to native rule syntax: Users can obtain a `.trf` file containing plain text representations of all rules that are currently defined by clicking a single button and saving the resulting file to a location of their choosing. `.trf` files contain rule names and descriptions, which makes them more easily searchable, and are immediately usable by the rewriting engine.

15 Rule Search

As the number of available rules increases, locating specific rules by simply browsing the rule base becomes increasingly difficult. At the same time, examining the rule base – to check coverage and identify missing rules – remains an integral part of the overall workflow. One could even argue that the larger the rule base, the more important being able to quickly "zone in" on specific rules becomes.

To facilitate the tasks of navigating large rule bases and locating specific rules, the system provides a search interface that can be accessed via the Search tab in the navigation bar. It has been designed to complement the functionality for filtering rules by name and description built into the rule browser, allowing users to conduct fine-grained searches for specific rules based on input features and/or output strings. Figure 18 shows the search interface in its initial state.

Search terms for matching features must be entered on the left, and search terms for matching output strings must be entered on the right. Each input field takes either a single feature or a single (output) string[56]. By default, the system displays two fields for entering search terms, but additional fields can be created as necessary by clicking a plus-shaped button that is displayed to the right of each input field[57]. Irrespective of the number of fields that are currently displayed, there is no need to fill in each one of them: As long as there is at least one non-empty field, the system will be able to perform a search. This saves users the trouble of having to remove empty input fields (cf. below). To

[56]Note that the system does not enforce this restriction in any way. Search queries based on input fields with multiple search terms (where "search term" denotes a feature, a feature-value pair, or a string) will simply be unsuccessful.

[57]Clicking + next to an input field in the left column adds another field for specifying features, and clicking + next to an input field in the right column adds another field for specifying strings. If a column is showing two or more input fields, only the field at the bottom has a + button associated with it. That is, additional fields are always *appended*; they currently can not be inserted between two existing fields.

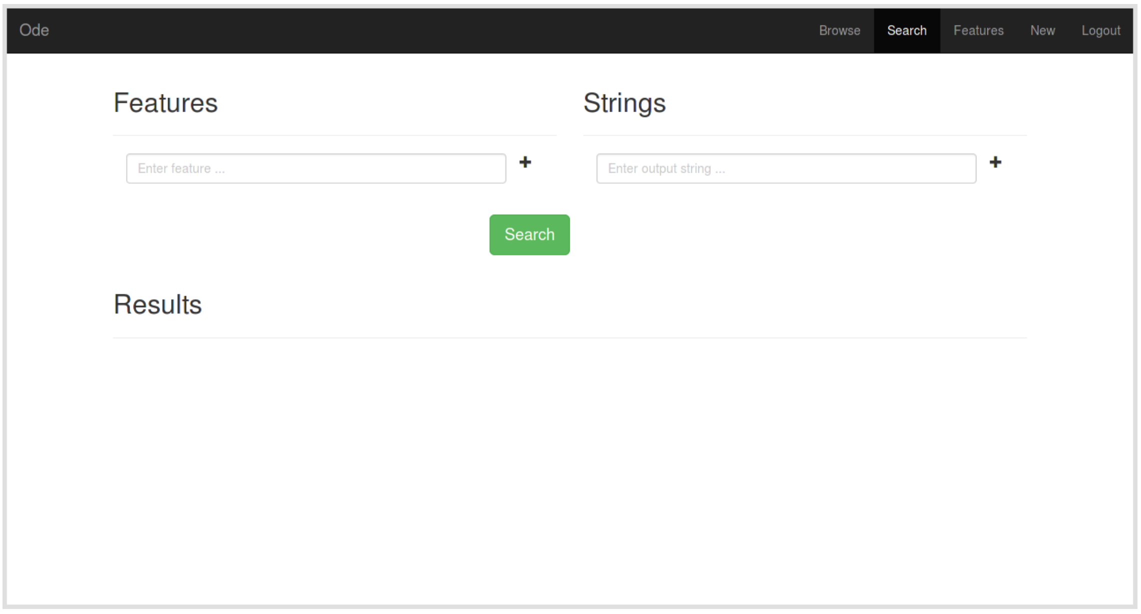


Figure 18: The search interface in its initial state

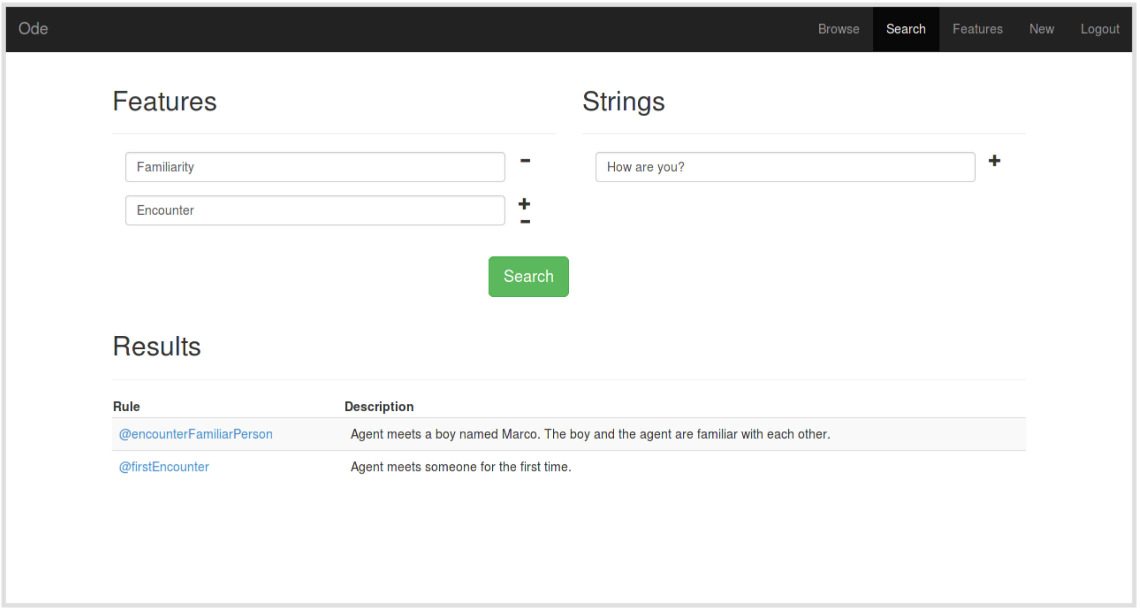
avoid unnecessary AJAX calls, input fields are checked for content on the client side before contacting the server. If all fields are empty, an appropriate warning message is displayed under the **Results** heading. If a column contains more than one search field, the system will display a minus-shaped button next to each field that allows users to remove individual fields in any order, which is useful for removing search terms from a query that turned out to be too specific. Search terms to be matched against input features or feature-value pairs are currently treated in a case-sensitive manner. This enables users to determine whether a rule base contains rules using features that differ only in their spelling without having to look at LHS of individual rules. Search terms representing (fragments of) output strings are treated in a case-*insensitive* manner.

With respect to matching rules against features, the search interface supports two levels of granularity: To search for all rules whose LHS incorporate a specific feature *irrespective of value*, the name of that feature (e.g., `Familiarity`) must be entered as a search term. The set of rules which associate a given feature with a specific value can be obtained by entering the name of the feature surrounded by angle brackets and followed by the name of the desired value (e.g., `<Familiarity>yes`). It is possible to mix these syntaxes in the context of a single search operation, meaning that the search component supports queries such as:

"List all rules that incorporate the `Encounter` feature
and the `Familiarity` feature with a value of `yes`."

Contents of individual input fields treated as conjunctive constraints when performing the search: For instance, if there are three non-empty fields in the **Features** column and two non-empty fields in the **Strings** column, rules that incorporate, e.g., two out of the three features specified will not be part of the results. The list of results will only include rules that match every single one of the search terms. Appendix B.4 describes the search algorithm in detail.

The system can be instructed to perform a search by clicking the Search button below the input fields for features and strings. Search results will be listed under the **Results** heading in the form of a table. For each rule matching the current set of search criteria, name and description will be shown. Rule names can be clicked to jump to the details page for the corresponding rule. Results can be sorted by name or description by clicking on corresponding column headers of the table, which might be useful if the number of rules matching a given query is large.. Multiple clicks on the same header toggle between ascending and descending order. By default, results are sorted by rule names in ascending order. Figure 19 shows the search interface after performing a search for rules matching two features (`Familiarity` and `Encounter`) and a single string (`How are you?`).



Rule	Description
@encounterFamiliarPerson	Agent meets a boy named Marco. The boy and the agent are familiar with each other.
@firstEncounter	Agent meets someone for the first time.

Figure 19: The search interface after performing a search

15.1 Summary

The search interface presented in this chapter complements functionality for filtering rules by name and description provided by the rule browser. It allows users to perform fine-grained searches for rules based on arbitrary combinations of features, feature-value pairs, and output strings. Search results are displayed in the form of a table that can be sorted alphabetically by rule name or description, which might be useful for large result sets. Each entry in the table links to the details page for the corresponding rule, which allows users to review exhaustive information about individual rules easily. Empty input fields are generally ignored: As long as at least one of the input fields that are currently displayed contains a search term, the system is able to perform a search. As a result, users do not have to clean up input fields they added but then decided not to use.

This concludes our discussion of user-facing functionality. The following chapters describe how the system presented in this thesis was evaluated and discuss the results that we obtained from the evaluation.

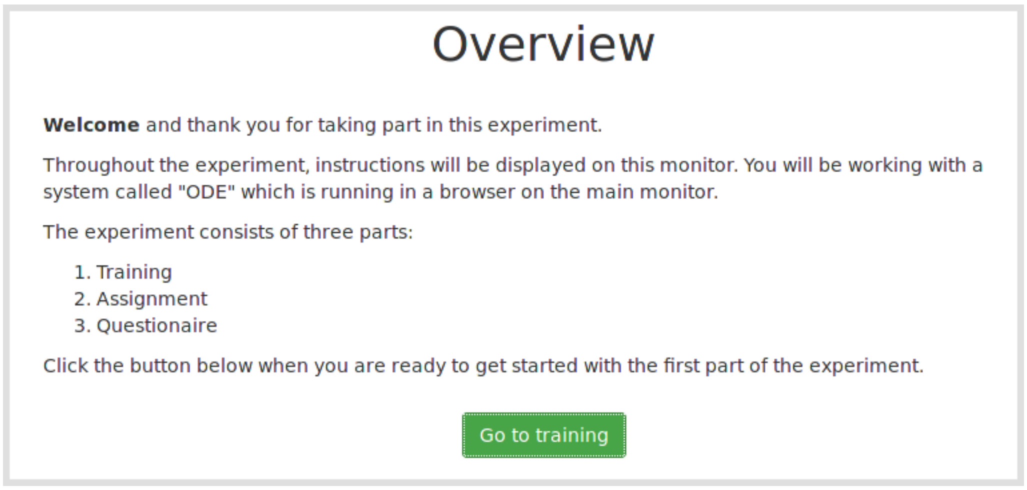
Part VI

Evaluation

16 Experimental Setup

We stated in Chapter 1 that one of our goals was to create a system that would be usable by non-experts, i.e., non-linguists and users without a background in computer science. It is this aspect that the evaluation experiments described in this Chapter focus on. In particular, we wanted to address the following questions: Is it possible for novice users of the system to produce rules that indicate a basic understanding of both the concept of rules and system functionality with only a small amount of training? And if so, how many alternative verbalizations do they produce per rule? To answer the first question, we designed a set of rule templates to act as a point of comparison for rules produced by participants of our experiments. Based on these templates we identified different types of errors that might affect LHS and RHS of rules created by participants. We then examined rule data collected over the course of the experiments for these specific types of errors and recorded how many rules were affected by them. Creation of rule templates is described in Section 17.1, which also presents different types of errors affecting LHS of rules. Errors affecting RHS of rules are covered in Section 17.2. To answer the second question, we collected total and average numbers of unique output strings created by participants, and also examined total and average numbers of output strings associated with individual rules subjects were asked to create. The purpose of collecting this data was two-fold: First, we wanted to see if rules produced by participants incorporated enough variation to (theoretically) be usable in a real world setting. Secondly, we wanted to see if the numbers would give any indication as to whether our system has the potential to enable users to create large amounts of variation in a short amount of time. Detailed results are presented in Section 17 below. In Chapter 18 we discuss the data we obtained with respect to the questions raised above.

Before we move on to describe the evaluation experiments we conducted in detail, we would like to mention a number of alternative ways in which our system could have been evaluated: For instance, since our system was developed from scratch, we could have tested it with respect to general usability and discoverability of features. However, despite the fact that the target audience of our system includes both expert and non-expert users, it is still a very specific audience and we expect new users to receive basic training and/or guidance in the beginning stages of using the system. (The design of the evaluation experiments described below reflects this assumption.) Secondly, as stated in Chapter 1, one of our aims is to facilitate maintenance and long-term evolution of potentially large rule bases. Unfortunately, designing and carrying out a long-term evaluation addressing this goal was outside of the scope of this work. Lastly, evaluating the benefits of the system we developed by comparing results from having participants create rules in native rule syntax and using our system seemed problematic for a number of reasons: On the one hand, this kind of evaluation strategy would have required availability of a sufficiently large number of experts in order to obtain reliable results. On the other hand, carrying out this kind of evaluation with non-experts would have required spreading out the experiments over two or more sessions for each participant in order to give subjects a chance to become sufficiently familiar with native rule syntax. Most importantly, even if we had been able to find a large number of experts or a large number of non-experts willing to participate in a multi-part evaluation experiment, comparing creation of plain text representations of rules to workflows supported by our system would have been unfair: Unlike our system, the process of writing rules in native rule syntax was not designed to be accessible to non-experts. Secondly, our system was specifically designed to address some of the challenges involved in working with plain text representations of rules, which would have likely skewed the results in its favor.


Overview

Welcome and thank you for taking part in this experiment.

Throughout the experiment, instructions will be displayed on this monitor. You will be working with a system called "ODE" which is running in a browser on the main monitor.

The experiment consists of three parts:

1. Training
2. Assignment
3. Questionaire

Click the button below when you are ready to get started with the first part of the experiment.

Figure 20: Overview page displayed at the beginning of each experiment

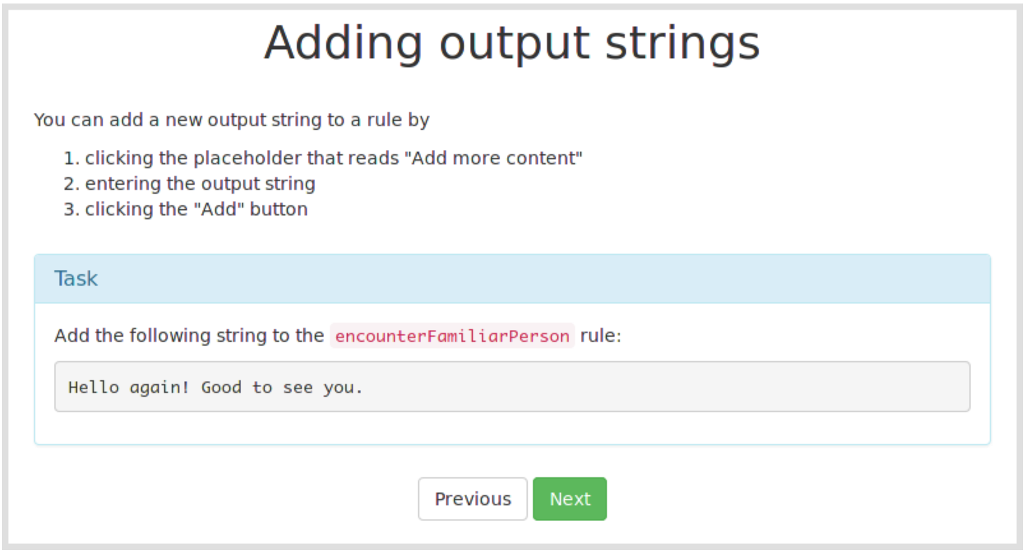
Experiments conducted to collect relevant data consisted of three parts: a training phase, an assignment, and a questionnaire. We will describe each of these in detail shortly. Subjects were recruited from the Psychology department of Saarland University. Throughout the experiment they were seated in front of a dual monitor setup. The system was running in a maximized browser window on the main monitor, and instructions for individual parts of the experiment were displayed in a full-screened browser window on a smaller monitor positioned to the right of the main monitor. Instructions were created in the form of HTML pages equipped with buttons for navigation. Aside from instructions specific to individual parts of the evaluation experiment, we also created an overview page that introduced participants to the monitor setup and listed the three phases of the experiment (cf. Figure 20). At the beginning of each experiment the overview page was shown on the secondary monitor, and the main monitor was set to the main page of our system[58]. The experimenter stayed in the room throughout the experiment but was seated at a desk facing away from the desk at which participants were working. Subjects were compensated with €12 for their participation.

16.1 Training

The training and assignment phases were designed to be similar to the kind of training actual users would be going through when first getting to know the system. For the purpose of familiarizing participants with relevant functionality (as well as the steps involved in designing rules) we designed a set of 20 *tasks* focusing on features for creating and editing rules. Each of these tasks included specific instructions about content to add, modify, or remove from LHS and RHS of rules. Due to time constraints, we did not introduce participants to functionality for searching the rule base during training. We also decided not to cover functionality provided by the feature editor in the training phase: Most importantly, we wanted subjects to focus on the core aspects of designing rules. Secondly, we did not consider the level of understanding about rule design that is required for creating appropriate features and values to be attainable with only 20 to 30 minutes of training. Note also that testing rules in the context of on-line processing was not part of the experiments.

[58]Under normal circumstances, this page is shown after a registered user successfully logs on to the system.

As a result, we did not introduce participants to functionality for exporting rules, and also did not show them how to use external testing tools.



Adding output strings

You can add a new output string to a rule by

1. clicking the placeholder that reads "Add more content"
2. entering the output string
3. clicking the "Add" button

Task

Add the following string to the `encounterFamiliarPerson` rule:

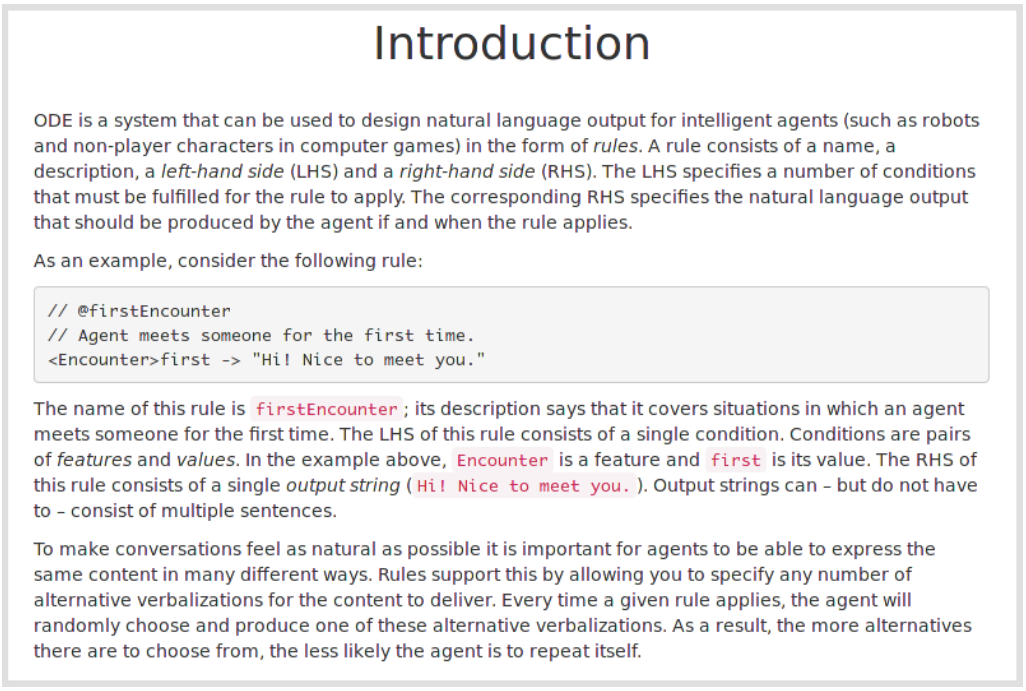
```
Hello again! Good to see you.
```

Figure 21: Page displaying task designed to teach subjects how to add output strings to RHS of rules

Each task was displayed on a separate HTML page and consisted of two parts, an introductory text explaining how to perform actions relevant to the task and a description of the task itself. Individual tasks addressing InputBuilder and OutputBuilder were designed to build on each other, making it necessary to complete them in order. Participants were allowed to navigate between task pages in order to re-read information about how to use specific features of the system. They were also explicitly encouraged to ask questions during training. Figure 21 shows one of the tasks that were part of the training material.

At the beginning of the training phase participants received basic information about rules and how they are used to generate natural language output for intelligent agents (cf. Figure 22). They were also introduced to the concept of tasks and instructed to complete them in order (cf. Figure 23). After reading this information, subjects were presented with the first task, which described functionality for creating new rules and asked participants to create a single rule. It was followed by a series of tasks addressing all aspects of editing LHS of rules: Subjects were instructed to add and remove features from the LHS of the rule created for the first task, and to practice changing feature values[59]. In order to enable subjects to correct potential errors in rule names and descriptions during the assignment, they were then asked to complete two separate tasks explaining how to rename rules and change rule descriptions. After learning how to switch between InputBuilder and OutputBuilder, participants had to complete another block of tasks focusing on features for editing rule output. Aside from introducing functionality for creating and operating on rule output, tasks belonging to this block also taught subjects how to check rule output for errors and duplicate strings. More specifically, one of the tasks taught participants how to view full sets of output strings associated with individual rules (cf. Appendix C.13). In a later task subjects were instructed to make use of this functionality to check their work for errors while making several edits to a combination group with multiple slots (cf. Section C.19.3). The final task introduced basic usage of the rule browser. Unlike functionality provided by the search interface and the feature editor, functionality for browsing rule bases had to be

[59]Note that tasks targeting functionality for working with LHS of rules did not introduce participants to complex features and nested AVMs. To eliminate potential sources of confusion, we did not add any complex features to the feature inventory they worked with during training and assignment.



Introduction

ODE is a system that can be used to design natural language output for intelligent agents (such as robots and non-player characters in computer games) in the form of *rules*. A rule consists of a name, a description, a *left-hand side* (LHS) and a *right-hand side* (RHS). The LHS specifies a number of conditions that must be fulfilled for the rule to apply. The corresponding RHS specifies the natural language output that should be produced by the agent if and when the rule applies.

As an example, consider the following rule:

```
// @firstEncounter
// Agent meets someone for the first time.
<Encounter>first -> "Hi! Nice to meet you."
```

The name of this rule is `firstEncounter`; its description says that it covers situations in which an agent meets someone for the first time. The LHS of this rule consists of a single condition. Conditions are pairs of *features* and *values*. In the example above, `Encounter` is a feature and `first` is its value. The RHS of this rule consists of a single *output string* (`Hi! Nice to meet you.`). Output strings can – but do not have to – consist of multiple sentences.

To make conversations feel as natural as possible it is important for agents to be able to express the same content in many different ways. Rules support this by allowing you to specify any number of alternative verbalizations for the content to deliver. Every time a given rule applies, the agent will randomly choose and produce one of these alternative verbalizations. As a result, the more alternatives there are to choose from, the less likely the agent is to repeat itself.

Figure 22: Introductory text providing basic information about rules. Displayed on the same HTML page as the text shown in Figure 23.

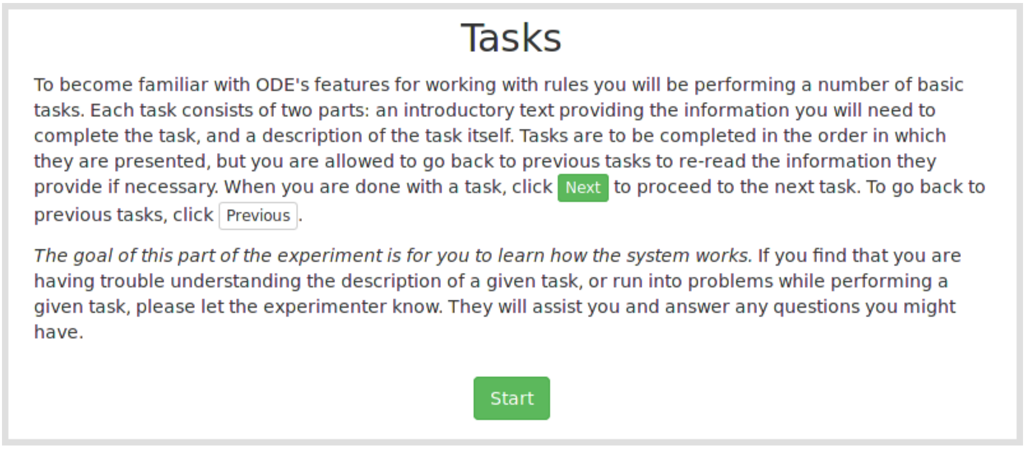
covered in the training phase because not being able to navigate between rules would have made completing the assignment unnecessarily difficult. A full list of tasks can be found in Appendix C. After completing the training phase, subjects were allowed a short break before starting the second part of the experiment.

16.2 Assignment

Participants were given 45 minutes to create a small corpus of five rules covering different situations belonging to a scenario in which a robot is playing a quiz game with a familiar user. We placed no restrictions on the order in which rules had to be created and worked on. Rule names and descriptions were provided in the assignment instructions, but subjects had to create the rules themselves and fill both LHS and RHS[60]. We chose to have participants work with predefined rule names and descriptions for several reasons: First of all, we did not want subjects to dedicate valuable time to coming up with rule names and descriptions themselves. Secondly, in order to obtain comparable results, giving participants a completely open-ended task[61] was not an option. As a result, assignment instructions had to include descriptions of specific rules subjects were supposed to create. With these descriptions in place, however, it did not seem sensible to ask participants to create original descriptions for the rules. Lastly, by providing rule names and descriptions we were honoring the assumption that in a real world setting, novice users would most likely continue to receive fairly concrete instructions about the kinds of rules they are expected to create after completing their very first training session. The decision to never-

[60] As mentioned above, each task participants had to complete during training included specific instructions about content to add, modify, or remove from LHS and RHS of rules. No such information was provided for any of the rules subjects had to create to complete the assignment.

[61] For instance, we could have specified a general scenario and left the task of coming up with appropriate rules completely up to participants.


Tasks

To become familiar with ODE's features for working with rules you will be performing a number of basic tasks. Each task consists of two parts: an introductory text providing the information you will need to complete the task, and a description of the task itself. Tasks are to be completed in the order in which they are presented, but you are allowed to go back to previous tasks to re-read the information they provide if necessary. When you are done with a task, click Next to proceed to the next task. To go back to previous tasks, click Previous.

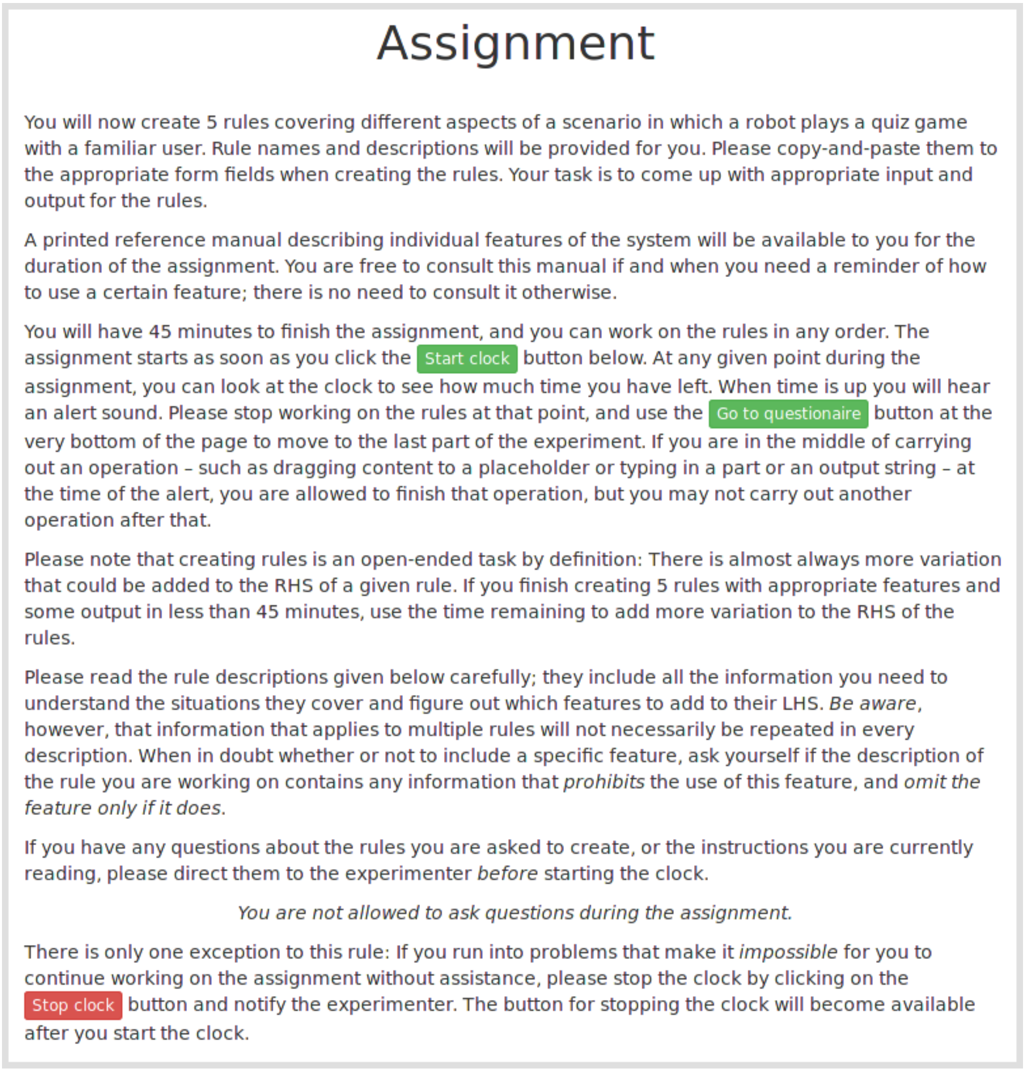
The goal of this part of the experiment is for you to learn how the system works. If you find that you are having trouble understanding the description of a given task, or run into problems while performing a given task, please let the experimenter know. They will assist you and answer any questions you might have.

Figure 23: Text describing the concept of tasks. Displayed on the same HTML page as the text shown in Figure 22.

theless have subjects *create* the rules themselves instead of making them fill in predefined rule stubs was made for two reasons: Most importantly, we wanted to avoid any thematic overlap between training and assignment. Aside from not having participants recreate the same rule they built during training, we also had to ensure subjects did not have access to any information about the rules they were supposed to create during the assignment in order to achieve this goal. Secondly, providing a set of predefined rules to fill in *and* keep participants from seeing these rules during training would have required us to exchange the database between training and assignment. This was undesirable because it most likely would have further increased the overall run-time of individual experiments.

In contrast to the training phase, subjects were not allowed to ask questions while completing the assignment. Instead, they were told to ask any questions they might have about the assignment before starting to work on it, and to only break the aforementioned rule if they encountered any problems that made it impossible for them to proceed without assistance. However, we did provide a printed reference manual describing relevant functionality that participants were allowed to consult in case they needed a reminder of how to perform a certain action. This manual listed and described system features in the same order in which they were introduced during training. We reused introductory texts from corresponding task pages to create individual entries of the manual. Task descriptions were omitted, and no additional information was provided about how to perform different types of actions. A copy of the reference manual can be found in Appendix E. We chose to provide a reference manual because as mentioned in the previous section, our goal was to create a scenario that was similar to how actual users might be introduced to the system. While we do expect that users will stop receiving human assistance once they are sufficiently familiar with the system, we have no reason to assume that they will not have any kind of reference material at their disposal when working on a rule base by themselves.

Assignment instructions (shown in Figure 24) and rule descriptions stayed visible for the whole duration of the assignment. Participants were made aware of the time limit for completing the assignment because we wanted to give them the opportunity to plan their time, so as to avoid situations in which subjects failed to finish the assignment because they spent too much time focusing on a subset of the rules we asked them to create. In order to make it easy for subjects to check how much time they had left to work on rules, we added a countdown timer (in minutes) to the HTML page displaying the assignment instructions. The HTML page also included functionality for stopping the timer in case of serious problems requiring intervention of the experimenter. Participants were told to start the timer after they had finished reading assignment instructions and asking questions


Assignment

You will now create 5 rules covering different aspects of a scenario in which a robot plays a quiz game with a familiar user. Rule names and descriptions will be provided for you. Please copy-and-paste them to the appropriate form fields when creating the rules. Your task is to come up with appropriate input and output for the rules.

A printed reference manual describing individual features of the system will be available to you for the duration of the assignment. You are free to consult this manual if and when you need a reminder of how to use a certain feature; there is no need to consult it otherwise.

You will have 45 minutes to finish the assignment, and you can work on the rules in any order. The assignment starts as soon as you click the Start clock button below. At any given point during the assignment, you can look at the clock to see how much time you have left. When time is up you will hear an alert sound. Please stop working on the rules at that point, and use the Go to questionaire button at the very bottom of the page to move to the last part of the experiment. If you are in the middle of carrying out an operation – such as dragging content to a placeholder or typing in a part or an output string – at the time of the alert, you are allowed to finish that operation, but you may not carry out another operation after that.

Please note that creating rules is an open-ended task by definition: There is almost always more variation that could be added to the RHS of a given rule. If you finish creating 5 rules with appropriate features and some output in less than 45 minutes, use the time remaining to add more variation to the RHS of the rules.

Please read the rule descriptions given below carefully; they include all the information you need to understand the situations they cover and figure out which features to add to their LHS. *Be aware*, however, that information that applies to multiple rules will not necessarily be repeated in every description. When in doubt whether or not to include a specific feature, ask yourself if the description of the rule you are working on contains any information that *prohibits* the use of this feature, and *omit the feature only if it does.*

If you have any questions about the rules you are asked to create, or the instructions you are currently reading, please direct them to the experimenter *before* starting the clock.

You are not allowed to ask questions during the assignment.

There is only one exception to this rule: If you run into problems that make it *impossible* for you to continue working on the assignment without assistance, please stop the clock by clicking on the Stop clock button and notify the experimenter. The button for stopping the clock will become available after you start the clock.

Figure 24: Assignment instructions

about them. If subjects were in the middle of carrying out an operation (such as dragging content to a placeholder or typing in a part or an output string) when the time limit was reached, they were allowed to finish that operation, but they were not permitted to carry out another operation afterwards.

As mentioned above, one aspect we wanted to address with the evaluation experiments was how much variation subjects would be able to generate within a certain time limit. To ensure that results produced by individual subjects would be comparable, we asked them to continue working on rule output even if they managed to create five rules with appropriate features and *some* output in less than 45 minutes.

Aside from a single example rule and the rule created during training, no other rules were present at the beginning of the assignment. The set of features and values from which LHS of individual rules had to be built was carried over from the training phase as well. Note that this set did not contain any features or values that were irrelevant with respect to completing the assignment. To provide further assistance for the task of choosing appropriate features, participants were told to check rule descriptions for information that would *prohibit* the use of a given feature, and to add the feature if no such information existed. Appendix D lists the rules subjects were asked to create.

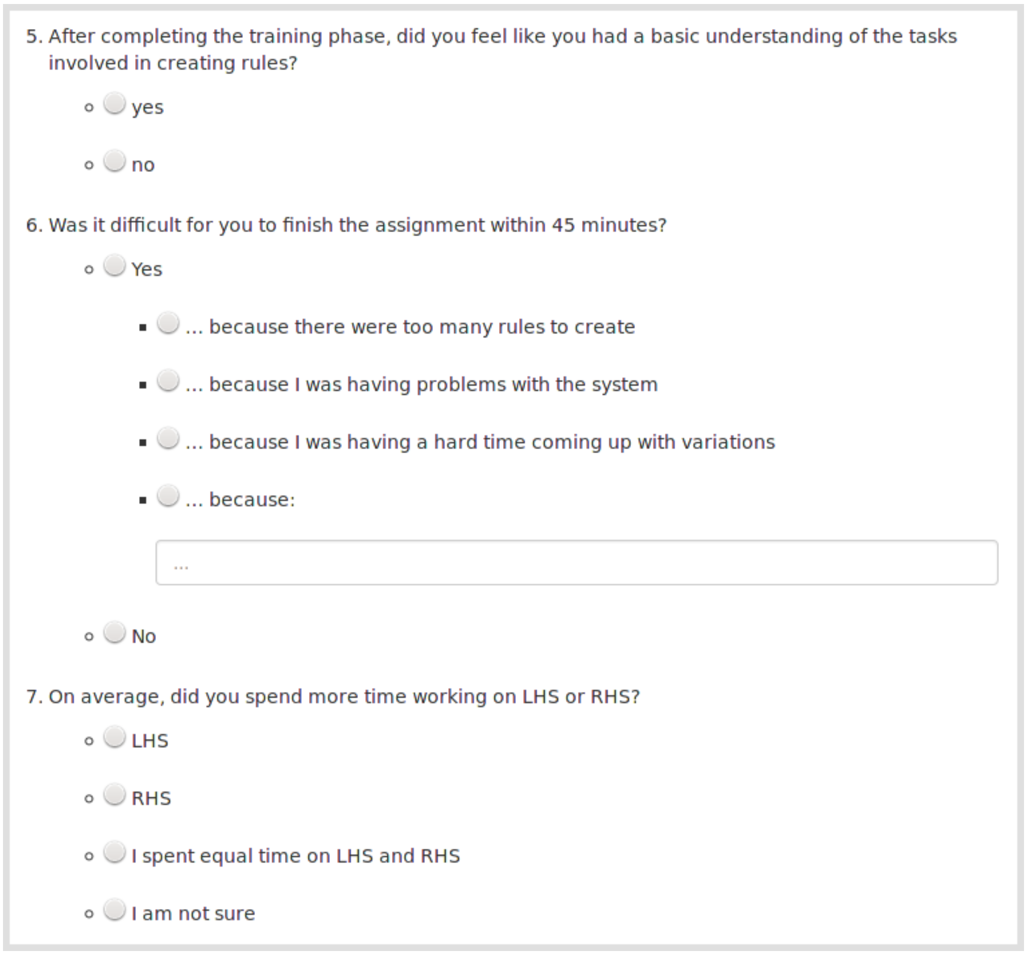
16.3 Questionnaire

The main purpose of including a questionnaire in the experiment was to collect information that would help us interpret results produced by individual participants and to elicit feedback about the system that would give us ideas for potential improvements. Subjects were asked to answer a total of twelve questions. The first four questions were designed to cover their academic background, native language, and level of computer literacy. These questions were followed by a set of three questions addressing training and assignment phases: We asked participants if they felt that they had a basic understanding of the tasks involved in creating rules after completing the training phase. With respect to the assignment, we asked them whether they were having difficulty completing it within the allocated time frame. In case they answered affirmatively, they were also asked to specify *why* they were having trouble. The questionnaire provided a set of predefined reasons to choose from for this purpose (which to us seemed to be the most likely causes for not being able to finish the assignment in 45 minutes), and also included a field that allowed subjects to specify a custom reason. In the last question addressing the assignment we asked participants to estimate whether they had spent more time editing LHS or RHS of rules. Predefined answers to this question also allowed subjects to specify that they had spent equal time on LHS and RHS, or that they did not know. The next three questions focused on system behavior: We first asked participants whether they had encountered any errors while working on the assignment, and if they did, they were told to describe what they were trying to do when the error(s) occurred. In the second and third question we asked subjects whether they had observed any behaviors that confused them or slowed them down, respectively. If they answered affirmatively, they were asked to describe how they would have expected the system to behave and how the behaviors they felt were slowing them down could be improved. All of the questions listed above were mandatory: If subjects tried to submit the questionnaire without answering them, an error message telling them that they had failed to answer all mandatory questions was displayed. The last two questions included in the questionnaire were optional. They asked participants to suggest additional functionality that would make editing rules faster and/or more convenient, and gave them a chance to leave additional comments. Choosing not to answer them did not lead to any error messages[62]. Appendix F provides a complete listing of questions and answer options that were part of the questionnaire. Figure 25 shows an excerpt of the HTML page displaying the questionnaire.

17 Results

We ran the evaluation experiment described above with a total of ten subjects. All of them were enrolled as students at Saarland University at the time of the experiments, and the majority of them (eight subjects) were majoring in Psychology. None of the participants had a background in Linguistics, Computer Science, or Computational Linguistics, but one subject was majoring in "Vergleichende Sprach- und Literaturwissenschaften sowie Translation (VSLT)"[63]. Nine subjects were native speakers of German, with one of them being a bilingual speaker of German and Greek, and one subject was a native speaker of Bulgarian. With respect to their level of computer literacy, all participants classified themselves as casual users.

[62]Note, however, that only the last question was explicitly marked as "optional".
[63]Comparative Philology/Linguistics, Literature, and Translation.


5. After completing the training phase, did you feel like you had a basic understanding of the tasks involved in creating rules?

- ◯ yes
- ◯ no

6. Was it difficult for you to finish the assignment within 45 minutes?

- ◯ Yes
  - ◯ ... because there were too many rules to create
  - ◯ ... because I was having problems with the system
  - ◯ ... because I was having a hard time coming up with variations
  - ◯ ... because:

    ...
- ◯ No

7. On average, did you spend more time working on LHS or RHS?

- ◯ LHS
- ◯ RHS
- ◯ I spent equal time on LHS and RHS
- ◯ I am not sure

Figure 25: Excerpt of HTML page showing questionnaire

Since we did not set time limits for completing training and questionnaire, we could only give a rough estimate of how long individual experiments would take when recruiting participants. Our original estimate of 75–90 minutes turned out to be fairly accurate: On average, participants completed the experiment in 82.9 minutes. One subject was able to finish in 72 minutes, and another subject took 95 minutes to complete all parts of the experiment. The remaining participants finished within the allocated time frame of 75–90 minutes. Participants completed the training phase in 27.2 minutes on average, with individual times ranging from 19 to 34 minutes. None of the subjects required assistance from the experimenter during the assignment; all of them were able to complete the assignment in 45 minutes without stopping the clock. For each participant, Table 6 provides a detailed listing of time spent on individual phases of the experiment.

70 percent of all subjects completed the training phase with errors. Nevertheless, every participant was able to achieve the minimal goal of creating five rules with non-empty LHS and RHS while working on the assignment. With respect to output string volume, however, results varied considerably across subjects: As can be seen in Table 7, participant 01 created only five output strings in total, averaging one output string per rule. On the other end of the spectrum, participants 08 and 06 created 365 and 396 unique output strings, respectively, averaging 73 and 79.2 strings per rule. For the remaining subjects, results were less extreme. They managed to create between 26 and 170 output strings in total, resulting in an average of 5.2 to 34 strings per rule. Examining the data from the

Participant ID	**Training**	**Break**	**Assignment**	**Questionnaire**	**Total time**
01	33	2	45 + 4	5	89
02	27	1	45 + 6	3	82
03	27	2	45 + 3	2	79
04	19	2	45 + 5	4	75
05	34	3	45 + 8	5	95
06	23	2	45 + 3	3	76
07	31	4	45 + 6	3	89
08	19	1	45 + 6	1	72
09	25	3	45 + 7	2	82
10	34	4	45 + 5	2	90
Total	272	24	450 + 53	30	829
Average	27.2	2.4	45 + 5.3	3	82.9

Table 6: Time to complete individual phases of the evaluation experiment by participant (in minutes). Additional time for assignment was spent on reading instructions beforehand. All subjects completed the assignment in 45 minutes without requiring assistance from the experimenter.

perspective of individual rules (cf. Table 8) we can see that they were assigned 20.4 to 34.1 output strings on average.

The fact that all participants were able to complete the assignment is consistent with the way they answered questions addressing training and assignment phases: All subjects stated that they felt they had a basic understanding of the tasks involved in creating rules after the training phase. Furthermore, only one subject reported that they found it difficult to finish the assignment within 45 minutes, attributing this to a lack of ideas for alternative verbalizations. With respect to the amount of effort spent on editing LHS and RHS of rules, seven subjects declared that they invested more time in RHS, and three subjects stated that they were not sure.

Responses to questions regarding system behavior were largely positive. Seven out of ten participants reported that they experienced no errors while working on the assignment. Judging from the error descriptions they provided, the remaining subjects misinterpreted the question and did not actually experience the kinds of system errors we were targeting with this question: One participant listed errors that they made themselves while working on the assignment, and another participant complained about accidentally deleting a rule when trying to copy it[64]. Lastly, one participant reported a "search window" popping up while they were typing. It was later confirmed that this was caused by a keyboard shortcut defined at operating system-level that had not been disabled prior to this particular run of the experiment.

Only one subject stated that they were confused by the behavior of the system, claiming that they tried to "delete an unwanted rule" but could not. If the delete operation had actually failed, this would have indicated the presence of a hitherto unknown defect in our system. However, we have reason to assume that the operation did in fact succeed: As mentioned above, rules present during training were carried over to the assignment

[64]Note that our system currently does not provide functionality for copying rules, and that the training material did not include any information suggesting the presence of such a feature.

Participant ID	**Total # of output strings**	**Average # per rule**
01	5	1
02	170	34
03	26	5.2
04	84	16.8
05	96	19.2
06	396	79.2
07	49	9.8
08	365	73
09	129	25.8
10	97	19.4
Total	1417	283.4
Average	141.7	28.34

Table 7: Total and average numbers of output strings created by individual participants

phase. One of these rules was missing from the final result produced by the participant in question. At the same time, the final result did not contain any leftover rule stubs or additional rules we did not ask for in the assignment instructions. Finally, since the number of nodes and relationships associated with a single rule can be quite large, deleting rules is not always instantaneous, and the interface for browsing rules currently does not provide visual feedback to indicate that a deletion operation is ongoing. Based on these considerations it is possible that the deletion operation was targeting the rule that was later found to be missing, and that the subject simply did not realize that the operation had been successful due to a lack of visual feedback.

Finally, two out of ten participants reported that they felt the system was slowing them down. One of them listed the repeated appearance of the "search window" as a cause for this but as mentioned above, this behavior was confirmed to be unrelated to the system being tested. The other subject stated that not being able to copy-and-paste parts or change their order by "switching" them was slowing them down. They also pointed out that having to click a button instead of simply pressing Return to finalize some of the editing operations slowed down the editing process.

We stated in Chapter 16 above that our main goal was to assess whether subjects would be able to acquire a basic understanding of both the concept of rules and system functionality with only a small amount of training. In order to do so, we examined LHS and RHS of rules created by participants for specific types of errors. The next two sections describe these errors and present our findings from analyzing rule data.

17.1 Error Types: LHS

Templates for individual rules participants had to create as part of the assignment were derived from the results of repeatedly feeding sets of input features to the rewriting engine in order to collect sets of unique output strings produced by applying existing rules. The following example shows the result obtained from this process for a single set of input features:

Participant ID	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**
01	1	1	1	1	1
02	36	24	40	10	60
03	6	4	4	6	6
04	16	16	20	20	12
05	36	12	20	20	8
06	72	32	160	90	42
07	17	10	9	7	6
08	78	192	24	15	56
09	36	12	27	18	36
10	26	6	36	17	12
Total	324	309	341	204	239
Average	32.4	30.9	34.1	20.4	23.9

Table 8: Average number of output strings created for each rule

```
### @salutation02:dvp(
<Context>(ctxt:context ^ <ChildName>marco ^ <Familiarity>yes) ^
<SpeechAct>closing)
bye!
bye, marco!
marco, bye!
marco, see you!
see you!
see you, marco!
```

As the output strings that were generated by the rewriting engine suggest, this combination of input features covers situations in which an intelligent agent is supposed to say goodbye to the current user. The corresponding rule we ask for in the assignment is called `@sayGoodbye`. For the purpose of evaluating LHS of `@sayGoodbye` rules produced by subjects, we disregard the fact that the `ChildName` and `Familiarity` features are part of a nested AVM embedded by the `Context` feature: Since we did not introduce participants to the concept of complex features during training, they were only expected to add three features – `ChildName`, `Familiarity`, and `SpeechAct` – to the LHS of the `@sayGoodbye` rule and set their values to `marco`, `yes`, and `closing`. Consequently, the rule template that we used to evaluate `@sayGoodbye` rules produced by participants looks like this:

```
// Name: @sayGoodbye
// Description: The game has ended. The robot says goodbye to Marco.
:dvp ^ <ChildName>marco ^ <Familiarity>yes ^ <SpeechAct>closing
->
bye!
bye, marco!
marco, bye!
marco, see you!
see you!
see you, marco!
```

We followed the same approach when evaluating LHS of the four remaining rules for each participant. As regards rule output, we did not expect subjects to exactly reproduce sets of strings generated from specific input structures. Section 17.2 describes the types of errors we looked for when comparing output of rules produced by subjects to these sets of strings.

Based on the considerations above we can identify a number of different types of errors that might affect LHS of rules created by participants. For instance, a given feature might be set to an *incorrect* value, i.e., its value might not match the value associated with the corresponding feature in the rule template. Secondly, LHS of individual rules might be missing *some* or *all* atomic features that are present in the corresponding rule templates. As an example, consider the LHS of the following rule created by one of the participants as part of the assignment:

```
// Name: @requestPlayQuiz
// Description: The robot asks a boy named Marco if he wants to
//              play the quiz game with him. The robot is familiar
//              with this boy, and the two of them have played
//              this game before.
:dvp ^ <ChildGender>male ^ <ChildName>Marco ^ <CurrentGame>quiz
     ^ <Encounter>notfirst ^ <Familiarity>yes ^ <GameQuiz>notfirst
     ^ <SpeechAct>greeting
->
...
```

In this rule, the `SpeechAct` feature has been set to an incorrect value: As the description indicates, the rule should apply in situations where an agent is supposed to ask a user if they want to play a quiz game. It is not supposed to cover situations in which a greeting would be appropriate. In the corresponding template rule, `SpeechAct` is set to the value `request`. Furthermore, the LHS of this rule is missing a feature called `About`, which is used to specify the topic of a `SpeechAct`.

The third class of errors that might affect rules concerns the total number of features subjects added to individual rules. Specifically, LHS of rules created by participants might reference a *smaller* or a *larger* number of features than corresponding rule templates. Finally, if the LHS of a rule to evaluate contains features that are not part of the set of features present in the rule template, these features and/or the values they are set to might be *inappropriate* for situations covered by the parent rule. The following rule incorporates a larger number of features (7) than the corresponding rule template (3). It also references the `CurrentGame` feature, which is inappropriate for the `@sayGoodbye` rule because it stores information about which type of game is currently ongoing, and games normally end before agent and user say goodbye to each other:

```
// Name: @sayGoodbye
// Description: The game has ended. The robot says goodbye to Marco.
:dvp ^ <ChildGender>male ^ <ChildName>Marco ^ <CurrentGame>quiz
     ^ <Encounter>notfirst ^ <Familiarity>yes ^ <GameQuiz>notfirst
     ^ <SpeechAct>closing
->
...
```

If any of these types of errors occur frequently across rules produced by the same subject, this points to general misconceptions about the purpose of features and values and how they are used to represent situational knowledge. On the other hand, if a given error type affects only a small number of rules, we assume that it is more likely to have been caused by misconceptions about the purpose of *specific* features and/or situations to which specific rules are supposed to apply.

Table 9 shows the average number of rules affected by a given type of error per participant. Note that different types of errors were treated equally when examining LHS of rules produced by participants: For each error type, we only recorded *if* a given rule was affected by it. We did not take into account *how many* instances of it were present in its LHS, the reason being that some types of errors affect rules as a whole while others affect individual features.

Error type	**Average # of affected rules**
Incorrect values	1.2
Some features missing	1.9
All features missing	0
Larger # of features	3.3
Smaller # of features	1.2
Inappropriate features	0.8
Inappropriate values	0.3

Table 9: Average number of rules (out of 5) affected by specific types of LHS errors

As we can see, referencing a larger number of features than corresponding rule templates is the most common error affecting LHS of rules produced by subjects. However, the number of cases in which additional features and/or values are inappropriate for the rule to which they were added is small. Participants omitted some required features in 1.9 out of 5 rules on average, but there are no cases in which all required features were omitted. In 1.2 out of 5 rules, one or more features were assigned incorrect values.

17.2 Error Types: RHS

For the purpose of evaluating RHS of rules produced by participants, we identified five different types of errors to look for. First, output strings might contain errors related to grammar, spelling, punctuation, and wording. Compared to other types of errors, these errors are less relevant with respect to answering the questions we posed in Chapter 16. Nevertheless, we decided to take them into account because depending on how severe they are they might make individual output strings unusable. In addition to *language-related* errors, there are two types of errors which are associated to participants' understanding of situations covered by individual rules. On the one hand, output strings might be *inappropriate* for the situation addressed by their parent rule: `I am sorry, could you please repeat your answer?` is appropriate for a rule covering a situation in which the agent did not understand a particular utterance made by the current user. However, it is inappropriate for situations in which the user has chosen an incorrect answer for a quiz question and the agent is supposed to tell the user to try again and choose a different answer. On the other hand, contents of output strings might simply *exceed the intended scope* of a rule. For instance, output strings belonging to a rule that matches in a situation where the agent is supposed to ask a user if they want to play a quiz game do not normally include a greeting. But if they do, we do not consider them to be inappropriate for their parent rule as long as they also contain an actual play request. The last two types of errors we considered are most likely to be caused by incorrect use of editing features provided by the OutputBuilder. For one, content that should belong to the same output string might be spread out over multiple output strings. In this case we consider individual output strings to be *incomplete*. Alternatively, output strings might be incorrect due to *invalid* combinations of parts. We already gave examples of output strings affected by this type of error in Section 13.3.3 above. The following rule produced by one of the participants of our experiments is affected by three out of the four remaining types of errors:

```
// Name: @requestFeedbackQuiz
// Description: Marco and the robot are done playing the quiz
                game. The robot asks Marco if he enjoyed playing
                the game.
...
->
Alright, Marco! That was it!
Alright, Marco! That’s it!
Alright, Marco! Well done!
Did you enjoy it?
How did you like it?
It was a pleasure to play with you!
Thank you for taking part again!
```

First of all, individual strings associated with this rule are incomplete. By themselves, the first three strings do not match the situation outlined in the description of the rule. And even when combining them with the remaining strings in order to form complete output strings, a number of problems remain. For example, output strings incorporating sentences listed in the last two lines (such as `Alright, Marco! That was it! Thank you for taking part again!` and `Alright, Marco! Well done! It was a pleasure to play with you!`) are inappropriate for the situation covered by their parent rule because they do not contain an actual request for feedback from the user. On the other hand, output strings that do include a request for feedback (such as `Alright, Marco! That’s it! Did you enjoy it?` and `Alright, Marco! That was it! How did you like it?`) still exceed the intended scope of the parent rule: According to the corresponding rule template, the rule should focus specifically on eliciting feedback about the game from the user. Its output is not supposed to provide information about the quiz game having ended.

Table 10 shows the average number of rules that are affected by each of these errors per participant. Note that the numbers listed in the table do not take into account the ratio of correct to incorrect output strings. In other words, if a single output string was found to be affected by a given type of error, the rule as a whole was counted as being affected by it.

Error type	**Average # of affected rules**
Language-related	2.3
Inappropriate content	0.5
Scope exceeded	3.4
Incomplete strings	0.5
Invalid strings	0.2

Table 10: Average number of rules (out of 5) affected by specific types of RHS errors

As we can see, contents of output strings created by subjects exceed the intended scope of their parent rules fairly commonly, and language-related errors occurred in almost half of all rules on average. In fact, only two out of ten participants created rule output that turned out to be completely free of any language-related errors. The remaining types of errors affect only a small number of rules.

17.3 Feature Suggestions and Comments

As mentioned in Section 16.3, the questionnaire participants had to fill out after completing training and assignment phases included two optional questions asking for feature suggestions and additional comments. In their answers to the first of these questions, three subjects stated that it should be possible to change the order of slots, and one of them further suggested providing functionality for adding slots *between* existing slots as an alternative. Without this functionality, any slots that should be positioned to the right of a newly added slot must be deleted and subsequently recreated to achieve the desired order of slots. One participant suggested adding functionality for copying rules, and another subject proposed adding functionality for splitting "sentences" again after they "[have] been split already".

Apart from the feature suggestions listed above we only received a small number of general comments. One participant stated that they were having doubts as to whether the amount of training provided was enough to enable participants to successfully complete the assignment, and that the training phase should perhaps be extended. This feedback came from the same participant who produced the lowest number of output strings overall and admitted to having problems thinking of alternative verbalizations for individual rules. Another subject commented that they found it "far more difficult" to decide on a set of features to add to a given LHS than to create output strings for the corresponding RHS.

In the next chapter we will discuss the results described above and relate them to the questions raised in Chapter 16.

18 Discussion

In Chapter 16 we stated that the primary question we wanted to address with our experiments was whether novice users of the system would be able to produce rules indicating a basic understanding of both the concept of rules and system functionality with only a small amount of training. In order to answer this question, we will first discuss the data presented above with respect to the first aspect, i.e., the question of whether subjects acquired a basic conceptual understanding of rules. We will then review relevant data again to address the question of whether subjects acquired a basic understanding of functionality provided by the system. Lastly, we will address the second question raised in Chapter 16 by evaluating participants' performance with respect to the amount of variation they created for individual rules.

On a general level, the fact that all subjects were able to complete the assignment within the allocated time frame and without requiring assistance from the experimenter can be seen as a broad indication that they did acquire a basic understanding of rewriting rules during training[65]. Looking at data from LHS of rules created by participants we can find further support for this claim: First of all, for most types of errors the average number

[65]More specifically, while participants were not allowed to ask questions during the assignment, they were told to notify the experimenter if they encountered any problems that made it impossible for them to continue working on the assignment without assistance (cf. Section 16.2). The fact that none of the subjects turned to the experimenter for help while completing the assignment suggests that their

of affected rules per subject is fairly low. Secondly, in a majority of cases the errors are not systematic: That is, they usually affect one or two but not a majority or all of the rules produced by a given participant. As mentioned above, this suggests that these types of errors were caused by misunderstandings about the purpose of *specific* features and/or rules rather than general misconceptions about LHS of rules.

Out of the different types of errors described in Section 17.1, one type of error was found to affect a large number of rules: Many participants added more features than necessary to LHS of individual rules[66]. However, in most of these cases additional feature-value pairs do not contradict information provided by rule descriptions. This is also true for cases in which additional feature-value pairs added by subjects *replace* features that should have been present. This is one of two reasons why we do not take the presence of these errors as evidence for general misconceptions about the purpose of rules in general, and LHS in particular. The second reason is that there is a possibility that the tendency to add large numbers of features to LHS of rules was strengthened by a specific piece of advice given to participants in the description of the assignment: Rule descriptions were designed to contain information that would be easy to translate into features. However, in order to keep rule descriptions concise, we did not include repeating information in every single description. The assignment instructions mentioned this and also advised subjects to examine rule descriptions for information that would prohibit the use of specific features if they needed help to decide whether they should add a feature to a given rule. The instructions also encouraged them to only omit a feature from a rule if the corresponding description did contain such information (cf. Figure 24 in Section 16.2)[67]. It is possible that without this piece of advice, participants would have omitted features present in rule templates more often, which is what we were trying to avoid. As mentioned above, however, it is also possible that this piece of advice contributed to participants' tendency to add more features to LHS than strictly necessary. Based on these considerations, and taking into account that subjects were basing their work entirely on information provided by training material and assignment instructions it is not surprising that LHS of many rules contain a larger number of features than necessary.

Having established that LHS data supports the claim that most participants acquired a basic understanding of rules during training, we will now look at data from RHS of rules created by participants. As Table 10 shows, the average number of rules containing output strings that are inappropriate for their parent rules per participant is small. Out of 50 rules, only five (10%) contain one or more inappropriate strings. This indicates that some subjects misunderstood the purpose of specific rules, but can not be taken as evidence for a general lack of understanding of the concept of rules.

Looking at the most common error affecting RHS of rules produced by participants, the situation is slightly different: As mentioned above, many subjects created output strings that exceed the scope of their parent rules. The presence of a large number of output strings affected by this error suggests that participants were not always able to precisely identify the intended scope of individual rules. In particular, they failed to realize that output of different rules should generally not "overlap". This is evident from the fact that many participants decided to include a greeting in the output of a rule covering situations in which the robot is supposed to ask the current user if they want to play a quiz game – despite the fact that they had already built a rule addressing the situation of greeting a familiar user during training. While this is a more general misunderstanding that is not specific to individual rules, we doubt that it indicates a universal lack of understanding

understanding was at least good enough not to let them get to a point where they were forced to ask questions.

[66] The main problem with this is that it increases the amount of knowledge an agent must have for specific rules to apply.

[67] At the same time, we did not explicitly mention to participants that rules are made more specific by adding more features to them.

of the concept and purpose of rules: The contents that cause individual output strings to exceed the intended scope of their parent rules are still appropriate for the situations covered by these rules. Otherwise, affected output strings would have been classified as inappropriate.

The remaining types of errors are not relevant for judging participants' understanding of rules: Language-related errors, which are the second most frequent errors that affect RHS of rules, can make individual output strings unusable and therefore needed to be recorded. However, since they result from participants being non-native speakers of English, these types of errors do not at all point to misconceptions about rules. Lastly, incomplete and invalid output strings result from misunderstandings about system functionality, which we will discuss next.

The general observation that none of the participants required assistance from the experimenter while completing the assignment can not only be seen as a broad indication that subjects had a basic understanding of the concept of rules. It also suggests that they knew how to use relevant functionality provided by our system. With respect to LHS of rules, none of the errors participants made point to fundamental misunderstandings about editing support provided by the InputBuilder. Data gathered from rule output provides further support for the claim that subjects had a basic understanding of functionality provided by the system: First of all, the average number of rules with incomplete and invalid strings per participant is small. Only one subject produced incomplete output strings, and out of all rules created by participants, only two incorporate one or more invalid strings. This suggests that most subjects were able to grasp the concepts of slots, parts, and output strings, and understood how parts are combined to form output strings. The fact that all participants made use of parts and slots when defining rule output provides further support for this assumption. In fact, only two out of ten participants used regular output strings at all, and only one of them created a number of rules whose output consists solely of output strings. Additionally, four out of ten subjects made correct use of multiple combination groups, which suggests a good grasp of functionality that the OutputBuilder provides for the purpose of speeding up creation of content. Finally, as evidenced by feedback they provided in the questionnaire, some participants were able to identify shortcomings of the current OutputBuilder implementation, such as the inability to reorder parts, split parts, or split output strings into more than two parts. This would not have been possible if they had not gained a basic understanding of the purpose of splitting output strings into parts during training.

The second question we wanted to address with the evaluation experiments concerns the amount of variation participants produced for individual rules. Looking at Table 7 again we can see that two participants did not come up with large numbers of alternative verbalizations. In general, however, the amount of variation produced for individual rules was satisfying: Based on the data shown in Table 8, 35 out of 50 rules (70%) contain 10 or more output strings, 29 rules (58%) contain 15 or more output strings, and almost half of all rules (23 rules, or 46%) contain more than 20 output strings. This means that based on quantity alone, 70% percent of all rules created by participants would be eligible for being fielded: With 10 or more output strings to choose from for each rule, it is fairly unlikely that an agent would repeat itself across sessions to a degree that would make interactions feel forced or "unnatural" to a user. Note that this is not to say that adding even more variation to their RHS would not benefit the rules in question. However, considering the fact that they were produced in a 45-minute session, by first-time users with no background in relevant subject areas, we think that the amount of existing variation is satisfying.

Finally, before concluding our discussion of experimental results with a summary of the points made above, there is one more aspect we would like to mention: Some of the data we obtained suggests that our system does enable users to create large amounts of variation within a limited amount of time: As mentioned in Chapter 17, participants 08 and 06 created 365 and 396 output strings in total, respectively. It is very unlikely that they would have been able to produce that many output strings without the help of the system: Even if the system had provided rule stubs for them and they had focused solely on producing output strings (while completely neglecting LHS of rules), without the help of parts, slots, and combination groups they would have had to create each output string in 7.4 and 6.82 seconds or less.

18.1 Summary

Overall, the number of errors affecting LHS and RHS of rules produced by participants is fairly low. The types of errors that were more frequent were most likely caused by the fact that subjects received only a limited amount of information about rules. With respect to LHS we could not identify any errors that would suggest misconceptions about editing support provided by the system. As mentioned above, the number of errors related to misconceptions about editing support for rule output was fairly low. This suggests that overall, participants were able to become productive with the system after only a small amount of training, and that most of them achieved a level of understanding about rules that corresponded closely to the level of detail provided by the training material.

Augmenting the training material with a small amount of additional information, and changing the portions of the material that turned out to lead participants astray (causing them to add too many features to LHS), it should be possible to further improve the results first-time users can produce after a small amount of training. Overall, the frequencies of different types of errors made by participants suggest that understanding the concept of rules is more challenging than learning how to use the system. However, our findings also suggest that the system does make it possible to explain the concept and purpose of rules to non-experts in a way that enables them to quickly start creating and working with rules.

Part VII

Conclusion

19 Summary

We have presented a new graphical system for creation, maintenance, and long-term evolution of large-scale rule bases for output generation. Results of a first set of evaluation experiments suggest that using this system, people without background knowledge in (computational) linguistics and computer science can gain a basic understanding of the concept of rules and productively design rule output with relatively little training.

In our system, challenges that arise from having to create large numbers of alternative verbalizations are addressed through specialized editing features: First of all, full output strings belonging to RHS of rules can be split into parts distributed across an arbitrary number of slots in order to avoid having to specify identical content repeatedly when trying to increase the number of output alternatives. The possibility of using multiple combination groups allows users to enter sets of verbalizations efficiently if similarity of strings is high within sets and low across sets. Secondly, the system allows content to be reused in various ways. Within rules, combination groups can be copied to reuse their content as a starting point for creating additional verbalizations. A global parts inventory facilitates reuse of content for parts and output strings across rules, and by establishing cross-references, full sets of output strings associated with individual rules can be incorporated into slots of other rules as parts. As described in Chapter 18, the amount of variation created by some participants during the evaluation experiments indicates that editing functionality provided by our system has the potential to speed up the process of defining rule output considerably. At the same time, based on the written feedback we received from people who participated in the experiments there are a number of ways in which editing support for rule output could be further improved. We will mention these again in the next chapter, where we discuss future work.

A number of system features contribute to minimizing or completely avoiding errors that are likely to occur when editing plain text representations of rewriting rules. For instance, by separating the task of defining rules from the task of defining input features and keeping features read-only for the purpose of editing LHS of rules, the possibility of introducing errors through misspellings of feature names is eliminated. In any given LHS, atomic features can only be set to values that have been defined as appropriate in the feature editor, and AVMs embedded by complex features reject features that are not listed as acceptable targets. When adding features to the LHS of a rule, default values are set according to the features' types. This ensures that users can not make the mistake of using atomic features as complex features and vice versa. When editing rule output, users can check for errors and duplicate output strings by viewing full sets of output strings for each rule. Lastly, the interface for creating and editing features prevents any operations that would compromise the integrity of the rule base: Targets can only be added to a feature if they are appropriate for the feature's type, and removing a target from a feature will fail if the feature references it in one or more rules. If a feature is in use, the system will prevent it from being deleted.

Another important goal we tried to address was making the process of creating and editing rules accessible to people without background knowledge in (computational) linguistics and computer science. This is achieved by abstracting away from native rule syntax as much as possible: Graphical representations of LHS allow users to focus on pairing features with appropriate values. When editing RHS of rules, users can concentrate solely on what an agent should say in situations corresponding to conditions specified by the LHS. In neither case is there a need to know or think about how data structures being manipulated via the graphical interface translate to native rule syntax.

Our system supports multiple workflows for defining rule output, which not only allows users to tailor their approach to the type and amount of content they intend to create, but also makes it possible for novice users to stick to using basic editing functionality until they become familiar with advanced editing features. As mentioned above, experiments conducted to evaluate our system with respect to the goal of making the process of working with rules accessible to non-experts yielded promising results. The frequencies of different types of errors made by participants suggest that becoming familiar with the concept of rules was more challenging than becoming familiar with editing functionality provided by our system. Overall, however, the number of errors affecting LHS and RHS of rules produced by participants was fairly low. Increased frequencies of specific types of errors can be traced to the presence or absence of specific pieces of information from training material and assignment instructions. With respect to misconceptions about editing support for rules we were unable to identify any errors suggesting that participants were having problems with functionality provided by the InputBuilder, and the number of errors that suggest misconceptions about editing support for rule output was fairly low. Based on these results we assume that by augmenting the training material with a small amount of additional information, it should be possible to further improve results novice users can produce.

In addition to supporting fast creation of alternative verbalizations, minimizing errors, and lowering the learning curve for new users, our system also provides features that were specifically designed to support maintenance and long-term evolution of rule bases. First of all, it allows users to associate metadata with rules and features, which is useful for documenting their purpose and communicating this information to other developers. Metadata about rules is preserved when exporting rules to native rule syntax, which makes it easier for rule developers to locate specific rules during manual post-editing. When examining a rule base via the rule browser, users can not only view detailed information about individual rules but also filter existing rules based on metadata (i.e., rule names and descriptions). Additionally, to facilitate the task of checking rule coverage for specific scenarios, the rule browser provides functionality for viewing sets of rules that are similar to a given rule. Lastly, a dedicated search interface allows users to conduct fine-grained searches for rules based on arbitrary combinations of features, feature-value pairs, and output strings.

To facilitate collaboration, the system has been developed using a Software-as-a-Service approach, allowing registered users to work on a single copy of the rule base by contacting the application from their browsers. Integration with existing tools was achieved by adding functionality for exporting rules to native rule syntax.

Summing up, we are confident that our system will help rule developers design rules with large amounts of variation quickly and efficiently, while avoiding many classes of errors that that can easily occur when working with plain text representations of rewriting rules. Furthermore, we hope that by making the process of working with rules more accessible to non-experts, our system will open the door for new forms of collaboration between seasoned rule developers and experts of other domains that could benefit from the use of conversational agents.

20 Future Work

Since the system presented in this work was implemented from scratch, there are many challenges left to address in future work. Most of these challenges concern additional functionality that has not been implemented yet, but there are also a number of modifications which could be made to existing functionality in order to improve the system. We will address these first and then propose a number of additional features that should be included in future versions of our system.

20.1 Modifications

With respect to general usability, interfaces for editing features and rules should be modified to allow users to finalize *all* operations involving text input fields by pressing Return[68]. This is a fairly obvious way of reducing the effort required to edit textual content, and was also suggested by one of the participants of our study.

Regarding functionality for preventing errors, there are multiple points that need to be addressed. First of all, functionality for displaying full sets of output strings for individual rules should be modified to incorporate cross-references. As mentioned in Section 13.3.2, cross-references should not be expanded for this purpose in order to ensure that the system stays responsive. Instead, they should be incorporated into output strings in the form of clickable rule names.

We have mentioned several times that the system does not allow users to delete entities such as rules, features, and values if they are in use, and have discussed at length why this is important. However, the information that is displayed when an entity can not be deleted could be made more helpful: Instead of displaying a generic error message, the system might give users the option to view a list of entities that would be affected by a deletion operation, or present that list automatically. With the current implementation, users have to navigate to the search interface to get a list of rules incorporating a feature if they want to know which rules embed it on their LHS. For values and cross-referenced rules, there is currently no support for obtaining lists of entities that reference them, which means that users have to compile such lists manually if they need them. Depending on demand, future versions of the system might be extended to provide specialized support for these use cases. But even then, users should benefit greatly from being able to review a list of entities that reference a given value or rule when a deletion operation fails.

With respect to rule export, a number of changes should be made to existing functionality: First of all, export of rule subsets should be made more robust against errors by checking rules to include in the result file for cross-references to rules that are not part of the current subset. As a next step, the export logic should be modified to respect *narrowing* caused by filtering rules displayed in the rule browser by name and description. In other words, the exporter should not produce a `.trf` file containing plain text representations of all rules that are currently defined if the rule browser is showing a subset of rules whose names and/or descriptions match a string entered into the input field at the top. In order to facilitate porting our system to projects using different rule formats, the client-side code that handles rule export should be extracted into a separate, substitutable JavaScript module.

[68]Note that this does not necessarily mean that the possibility to finalize certain operations using button clicks will be removed.

Finally, in Chapter 17 we mentioned that lack of visual feedback for operations which take more than a few milliseconds to complete caused one of the participants of our study to become confused: Because the interface did not indicate that a specific operation the subject was trying to perform was still ongoing, the subject assumed that it had failed. It would therefore be desirable to eliminate lack of visual feedback as a source of potential confusion in future versions of the system by implementing client-side progress indicators for any operation that involves complex processing.

20.2 Additional Features

With respect to editing functionality, future versions of the system should provide better support for incorporating variables into LHS and RHS of rules. As explained in Section 13.3.2, creating working rules whose output references variable content captured on the LHS currently requires users to be familiar with the way rules are serialized to strings during export, and involves a number of steps that are prone to errors. These issues could be addressed in the following way: Ideally, the system should allow users to mark features as allowing variable content in the feature editor and generate targets representing variables automatically. Names of targets representing variables should be derived from names of corresponding features in a systematic and transparent manner. This should enable users to quickly judge which type of content to expect for a given variable when examining or modifying existing output, and to use variables productively when creating rule output themselves. When adding parts or output strings to a given RHS, the system should check whether they contain references to variables which are not "instantiated" on the corresponding LHS (and therefore not defined in the context of the parent rule), and reject them with an appropriate error message if necessary. Lastly, in order to eliminate the need for post-processing, the system should isolate variables from surrounding content appropriately during rule export, and refrain from wrapping them in quotation marks.

Regarding the goal of enabling users to create large numbers of alternative verbalizations for individual rules quickly, a number of additional features come to mind. Based on feedback from several participants of our study, future versions of the system should provide support for changing the order of slots belonging to a combination group (e.g. via drag-and-drop). Adding this functionality is important because as mentioned in Section 17.3, repositioning slots currently involves deleting and recreating one or more slots. Similarly, users might benefit from being able to move or copy individual parts between slots.

In some cases, users might realize that they will need three or more slots to represent content efficiently right after creating a small number of initial output strings. The current version of the OutputBuilder does not provide specialized support for this scenario, which means that after splitting an existing output string to create the default slots, users have to create one or more additional slots with appropriate content manually. This workflow could be optimized by allowing users to split output strings into more than two parts. A possible way to implement this functionality would be to have the areas between individual tokens of an output string react differently to clicks modified by pressing, e.g., Ctrl or Shift: Instead of splitting the string in question right away, the system could allow users to specify multiple split points by control-clicking or shift-clicking them, and offer a button to finalize the splitting operation. Alternatively, it could allow users to press Return to carry out the actual splitting operation.

In order to make the process of copying content within rules more flexible, we might introduce functionality for copying individual slots. Minimally, users could be offered a menu for selecting slots to transfer to a new combination group when copying an existing group. Additionally, functionality for reordering slots – which, as mentioned above, is one of the most important features that are currently missing – could be extended to allow users to transfer individual slots to other groups.

To support users in locating existing content and/or features that might be relevant for a given rule, we might add server-side functionality for asynchronously analyzing co-occurrence patterns of feature-value pairs and parts. Based on the assumption that specific combinations of features and values "trigger" certain phrases in the output of a rule, this would enable us to suggest parts to use based on features that a rule incorporates and vice versa. A fairly unobtrusive way of implementing this would be to make feature and parts inventories sortable by *relevance*. Of course, the quality of suggestions would depend on the amount of data available for analysis, so it might be best not to turn on user-facing suggestion features for very small rule bases.

Lastly, the process of editing rule output – especially for rules whose RHS already define large numbers of alternative verbalizations – could be further optimized by introducing functionality for checking full sets of output strings for duplicates. Minimally, duplicates could be highlighted in any list showing all output strings associated with a given rule, so as to make users aware of them. As a long-term goal, the system could be modified to remove duplicate output strings, or parts leading to the presence of duplicate output strings, automatically. This would free users from having to pay attention to duplicates completely.

Aside from additional functionality for editing rules, there are a number of features related to maintenance and long-term evolution of rule bases that should be added to future versions of the system. First of all, in order to enable users to quickly judge which rules could benefit from additional verbalizations, entries displayed in the rule browser should show the total number of output strings currently associated with the rule they represent. This information could also be displayed, e.g., on the details page or in the table showing search results on the search interface. Secondly, functionality provided by the search interface could be extended to allow for specifying *disjunctive* constraints for search results. That is, users should be able to search for rules incorporating, e.g., a feature X *or* a feature Y. Additionally, rule developers might benefit from functionality for checking *feature space coverage*: For a given set of atomic features and associated values, this type of functionality would allow users to visualize which combinations of features and values are already present in existing rules, and which combinations have yet to be explored.

Before we conclude we would also like to mention a number of additional features that concern the system as a whole: First, information about how to use functionality provided by our system should be made available in the form of an FAQ page, or possibly a set of help pages divided by topic. This would likely benefit both novice users and experts. The training material that was created for the evaluation experiments could be used as a starting point for this purpose. It might also be helpful to extend the amount of contextual help that is shown on individual interfaces based on the amount of experience a given user has. Note that this would require extending the data model for users to store usage information.

Secondly, changes made by other developers should become visible in real-time on interfaces showing affected data. That is, users should not have to refresh a page in order to check whether the content displayed on that page was changed by another developer after they first navigated to it. This can be achieved by using *WebSockets*, which enable two-way communication between clients and remote hosts [18]. Note that it will not be necessary to introduce additional technologies for this purpose: Play, the server-side MVC framework we used to build our core system, has built-in support for handling WebSockets[69]. Client-side JavaScript code can make use of a WebSocket API that is part of a number of technologies being developed alongside the HTML specification [21].

[69]cf. `https://www.playframework.com/documentation/2.3.x/JavaWebSockets`

Lastly, future versions of our system might include extended capabilities for project management, allowing users to group sets of related rules into *collections*, and to associate collections with specific *projects*. The latter aspect would facilitate development and maintenance of separate, potentially unrelated rule bases. Furthermore, introducing projects and collections would also provide a natural basis for controlling user access to individual resources.

Part VIII

Appendix

A Client-Side Data Models

A.1 Features

The most comprehensive feature model is used by the interfaces for editing features and values (described in Chapter 12) and the interface for editing LHS of rules (described in Section 13.2). These models incorporate attributes for all properties of feature nodes stored in the database. In addition to `name`, `description`, and `type` attributes, instances of the feature model also have a `targets` attribute associated with them. This attribute stores the `name`s of all nodes that are associated with a given feature via a relationship of type `ALLOWS`: For atomic features the value of the `targets` attribute is a list of permitted values, and for complex features it is a list of features permitted in substructures they embed.

Interfaces for rule search (described in Chapter 15) and showing rule details (described Section 14.1) also make use of feature models. However, since these interfaces require less information about individual features, the feature models they define omit some of the properties that are associated with feature nodes in the database: The search interface uses model objects with a `name` and a `value` attribute, and pages showing details about specific rules define a feature model that uses a `name` and a `type` attribute.

The only interface that incorporates a dedicated model for feature values (or, more specifically, values of atomic features) is the interface for editing features and values. Model objects representing values have a single `name` attribute that corresponds to the `name` property of value nodes stored in the database.

A.2 Rules

Rule data is much more complex than feature data. Furthermore, different interfaces focus on different aspects of working with rule data. As a result, client-side models representing rules vary even more than client-side models representing features. For instance, interfaces providing functionality for browsing and searching a rule base (described in chapters 14 and 15) define a very basic model that only takes into account `name` and `description` properties of rule nodes. On the other hand, pages displaying rule details (described in Section 14.1) currently do not make use of a dedicated rule model at all. Since they provide a read-only view of rule data that users can not interact with and focus on one rule at a time it is not necessary for them to wrap rule data in a model object. These interfaces do, however, define separate collection types for LHS and RHS in order to facilitate the process of generating appropriate visual representations for these entities: LHS are modeled as collections of attribute-value *pairs*. Each pair is a model object with three attributes, namely `parent`, `attribute`, and `value`[70]. The `parent` attribute stores a reference to the AVM embedding the pair, and `attribute` references a feature object. If the feature referenced by `attribute` is atomic, `value` is a string. If the feature is complex, `value` references an embedded AVM which is again represented as a collection of pairs. RHS are modeled as collections of output strings and do not have any attributes corresponding to server-side properties of RHS nodes associated with them. Output strings are model objects with a `content` attribute that corresponds to the `content` property of output strings stored in the database.

[70]Note that the pair model has no counterpart on the server-side.

As expected, interfaces for editing rule input and output (described in Sections 13.2 and 13.3) define the most comprehensive models for rules and their components. On each of these interfaces, rule objects have attributes corresponding to `name`, `description`, and `uuid` properties of rule nodes stored in the database. Additionally, the interface for editing rule input defines an `lhs` attribute for rule objects, and the interface for editing rule output defines an `rhs` attribute for rule objects.

The `lhs` attribute references a collection of pairs which represents the LHS associated with the rule object. This collection is of type `AVM` and incorporates a number of attributes that become relevant when communicating changes to the pairs it stores to the server: `ruleName` and `ruleUUID` are set to name and UUID of the parent rule, respectively. Another attribute called `uuid` stores the UUID of the AVM itself. The name of the parent rule is relevant because it is part of the resource URL of the LHS, and the `ruleUUID` and `uuid` attributes are used when operating on nodes and relationships belonging to the LHS on the server. In addition to attributes that are relevant for communicating changes to the server, collections of type `AVM` have an `accept` attribute that defines the set of features which may be added to the corresponding AVM: Top-level AVMs accept any feature. Embedded AVMs may only contain features that are associated with the embedding feature via relationships of type `ALLOWS`. Pair objects belonging to AVMs have the same attributes as pair objects defined by the interface for displaying rule details described above.

The `rhs` attribute references a model object of type `RHS` with three attributes, namely `ruleID`, `uuid`, and `groups`. `ruleID` and `uuid` store the name of the parent rule and the UUID of the RHS itself, respectively. The name of the rule to which the RHS belongs is relevant because it is part of the resource URL of the RHS, and the `uuid` of the RHS is used to operate on contents of the RHS on the server. `groups` is a plain Backbone.js collection of combination groups associated with the RHS. Combination groups are model objects with `position`, `ruleID`, `rhsID`, `outputStrings`, and `partsTable` attributes. The `position` attribute corresponds to the `position` property of combination group nodes stored in the database. `ruleID` and `rhsID` store the name of the parent rule and the UUID of the parent RHS, respectively. `outputStrings` is a plain Backbone.js collection of output strings. In addition to a `content` attribute corresponding to the `content` property of output string nodes stored in the database, each output string object also has `tokens`, `ruleID`, and `groupID` attributes. The value of the `tokens` attribute is derived from the value of the `content` attribute: It is a list of substrings that is obtained by splitting the `content` of a given string at whitespace, and it is used when building visual representations of output strings (with clickable areas for splitting them). `ruleID` stores the name of the parent rule, and the value of the `groupID` attribute is set to the UUID of the parent group. This is necessary because resource URLs of output strings not only include UUIDs of their parent rules, but also of their parent groups. The same is true for other entities referenced by combination groups, which is why the models that represent them also define a `groupID` attribute (cf. below).

The `partsTable` attribute of a combination group object references an instance of a model that wraps the set of slots associated with a given combination group. Instances of the `partsTable` model – which has no counterpart on the server – also use `ruleID` and `groupID` attributes to store names of parent rules and UUIDs of parent groups. Additionally, they have a `slots` attribute whose value is a plain Backbone.js collection of slots. Slots are modeled as objects with `position`, `ruleID`, `groupID`, `parts`, and `refs` attributes. The `position` attribute corresponds to the `position` property of slot nodes stored in the database. `parts` is a plain Backbone.js collection of parts, and `refs` is a plain Backbone.js collection of cross-references. The interface for editing rule output defines models for both parts and cross-references. Like their parent models for slots and parts tables, these models have `ruleID` and `groupID` attributes, but they also store the UUID of their parent slots in an attribute called `slotID`. This is necessary because UUIDs of slots are included in resource URLs of parts and cross-references embedded by slots. Finally, part objects have a `content` attribute that corresponds to the `content` property of part nodes stored in the database, and objects representing cross-references have an attribute called `ruleName` which stores the name of the rule being cross-referenced.

Since the interface for editing rule output also displays an abbreviated, read-only version of the corresponding input (cf. Section 13.2), it defines a separate model for LHS. Instances of this model have a single attribute called `pairs`, whose value is a plain Backbone.js collection of pair objects. Unlike instances of the pair models discussed above, each pair object only has two attributes (`attribute` and `value`) associated with it. The values of these attributes are plain strings. This is sufficient for two reasons: First, since the visual representation that is displayed in the OutputBuilder is read-only, and therefore does not change after it is first built, there is no need to hold on to additional information about features after constructing it. Secondly, as mentioned in Section 13.3, immediate substructures of LHS are ellipsized to `[...]`, which means that values of both atomic *and* complex features can be represented as plain strings.

A.3 Users

Functionality for registering and authenticating users communicates with the server via explicit (non-AJAX) HTTP requests, and as mentioned in Section 9.3, modifications to features and rules are currently not attributed to specific users. As a result, the current version of our system does not define any client-side data models for (registered) users.

B Algorithms

B.1 Generating Lists of Output Strings

The process for generating lists of output strings for the purpose of displaying them in the OutputBuilder and on the details page has been implemented as a client-side operation: No communication with the server is necessary to obtain the full list of output strings for a given rule. For each combination group the system recursively combines parts from individual slots. This process is very similar to computing the Cartesian product of all slots: Its input consists of a list of slots, where each slot is itself a list of parts. In each recursive step, the first two slots of the input list are merged into a single slot containing all possible linear combinations of parts. That is, each part from the first slot is combined with every part from the second slot using string concatenation, and the result is added to the new slot. When merging finishes, the new slot replaces the first two slots in the input to the recursive call, shortening the original input list by one. The base case is reached when only a single slot remains in the list of slots. At this point, the list of regular output strings that the current group contains and the list of output strings from the last remaining slot are merged to obtain the full list of output strings associated with the group. As a last step, the lists of output strings for individual groups are merged to obtain the final result.

B.2 Computing Sets of Similar Rules

We gave an informal description of what it means for two rules to be similar in Section 14.2 above. Based on this description we formally define the concept of rule similarity as follows:

Definition 1. Rule Similarity.
Let A and B be rules. Furthermore, let X denote the set of features associated with A, and let Y denote the set of features associated with B.
A is *similar* to B iff $X \supseteq Y$.

We take the following steps to compute the set of rules that are similar to a given rule according to this definition: We first obtain the set of features referenced by the rule from a JSON representation of its LHS. Structural information is disregarded in this step, i.e., features are added to the result set irrespective of the nesting level at which they occur in the AVM representing the LHS. For each feature we then compute the set of rules that embed it on their LHS. This process is explained in detail in Appendix B.4 below, which discusses the algorithm that the system employs for finding rules matching a query entered via the search interface. Lastly, the rule sets obtained in the previous step are intersected to produce the final result. This is necessary because according to the notion of rule similarity we employ, the result set must not contain rules incorporating only a *subset* of the features referenced by the original rule. By computing the intersection of all rule sets we are able to eliminate exactly those rules from the result which are present in some but not all of the sets. Note that we do not check whether individual rules in the result set include one or more features which are *not* referenced by the original rule. This ensures that rules whose LHS incorporate a proper superset of the features referenced by the original rule remain part of the final result.

We have mentioned before that several interfaces provide controls for accessing the set of rules that are similar to a given rule. Since none of these interfaces possess enough information to compute sets of similar rules by themselves, and switching between different interfaces currently involves explicit (non-AJAX) HTTP requests to the server anyway, the process described above has been implemented as a server-side operation.

B.3 Export Algorithm

Mapping names, descriptions, and LHS of rules to plain text representations conforming to the syntax used for rewriting rules in ALIZ-E is fairly straightforward. The main challenges involved in exporting rules lie in finding a way to incorporate the output of cross-referenced rules into plain text representations of RHS that is efficient both with respect to the size of the resulting rule file and with respect to the processing steps that are necessary to produce these representations. The following paragraphs provide a high-level overview of the algorithm that the system uses to export rules, and explain factors which influenced the design of this algorithm. Sections B.3.1 through B.3.3 explain how plain text representations of individual rule components are constructed, and Section B.3.4 describes how the system handles incomplete rules during export.

After a user initiates the export process as described above, the system first fetches LHS and RHS data from the server for each rule. This is necessary because as mentioned in Chapter 14, the rule browser only knows about rule names and descriptions by default. Separate AJAX requests – which do not necessarily have to complete in order – are used to obtain LHS and RHS data. For each rule, a set of cross-references is extracted from the slots belonging to its RHS. This set is stored as an attribute of the client-side model object representing the RHS of the rule. Processing continues as soon as LHS and RHS data is available for all rules. As a last step before generation of plain text representations begins, the list of rules to export is sorted in ascending order by the number of cross-references associated with each rule: If a rule is cross-referenced by one or more rules, a plain text representation of its output must be incorporated into the output of each rule that is cross-referencing it. As a result, native formats for rules that are cross-referenced by one or more rules must be generated before any rules that incorporate their output can be assigned a plain text representation. By sorting the list of rules to export by the number of cross-references per rule we can reduce the number of passes that we have to make over the data to generate plain text representations for all rules without having to obtain cross-reference chains from the database[71]. In fact, if there are no forward references from rules at lower indices to rules at higher indices of the sorted list, a single pass over the data suffices to process all rules. Note, however, that sorting rules by cross-reference counts does not guarantee that rules to export will be processed in optimal order. If there are multiple rules with identical cross-reference counts, their relative ordering stays the same: As mentioned above, rules are sorted by name in the rule browser. If, according to this criterion, a rule with a cross-reference count n is listed before another rule with the same cross-reference count, the second rule will still follow the first one after sorting by cross-reference counts. This is true irrespective of the number of rules that separate the first and second rule in the original list. No checks are made to determine whether the first rule cross-references the second rule.

As an example for how sorting by cross-reference counts impacts performance of the export algorithm, consider the set of rules listed in Table 11. When processing these rules in alphabetical order (i.e., without sorting them by cross-reference counts first), four passes over the data are necessary to generate native formats for all of them, and each rule is

[71]Additionally, by caching native formats for rules that have already been processed we make sure that these formats are generated only once per rule. This is explained in more detail below.

Rule	**Cross-References**	**Count**	**Processed in Pass**
R_a	R_f	1	4
R_b		0	1
R_c	R_b, R_e, R_d	3	3
R_d	R_g	1	2
R_e		0	1
R_f	R_e, R_g, R_c	3	3
R_g		0	1

Table 11: Example for a sub-optimal ordering of rules in the context of rule export. Four passes over the data are necessary to generate plain text representations for all rules.

touched twice on average. In the first pass, all seven rules are touched but only rules with a cross-reference count of zero (R_b, R_e, and R_g) can be processed. Each of the remaining rules contains at least one cross-reference that has not been resolved when the rule is touched for the first time. Out of four rules left to be handled after the first pass, only one can be processed in the second pass (R_d). Rules R_c and R_f are processed in the third pass, and yet another pass is necessary to process rule R_a since it references R_f and is touched *before* that rule in the third pass.

Comparing these numbers to the results we obtain when operating on the same set of rules *after* sorting them by cross-reference counts (cf. Table 12) we can see that the number of passes required to process all rules has been cut in half. The same is true for the number of times individual rules are touched on average. Six out of seven rules can be processed in the first pass, and only one rule (R_a) has to be touched twice, lowering the per-rule average to 1.1.

Rule	**Cross-References**	**Count**	**Processed in Pass**
R_b		0	1
R_e		0	1
R_g		0	1
R_a	R_f	1	2
R_d	R_g	1	1
R_c	R_b, R_e, R_d	3	1
R_f	R_e, R_g, R_c	3	1

Table 12: Example for a better ordering of rules in the context of rule export. Since R_a contains a forward reference to R_f, it can not be processed in the first pass. The remaining rules have to be touched only once.

After the list of rules to export has been sorted, we process the items it contains one by one. If a rule has no cross-references, we generate a plain text representation for it right away[72], cache this representation as an attribute of the rule, and add the rule to a list of rules that have already been processed. On the other hand, if the rule that is currently

[72]More specifically, we generate a JSON object holding plain text representations of the rule's *components* (i.e., name, description, LHS, and RHS). This is necessary because rules cross-referencing the current

being processed does cross-reference one or more rules, we check whether or not each of these rules has already been processed. If this is the case, the rule itself can be processed in the manner described above. The only difference is that this time, we also store the native formats of all cross-referenced rules in order to be able to incorporate them into the final format for the rule. (See Section B.3.3 below for detailed information on how the output of cross-referenced rules is incorporated into plain text representations of rule output.) If we can find a single rule that is cross-referenced by the current rule and has not been processed yet, we add the current rule to a list of rules that still need to be processed[73]. If this list is *not* empty when we complete the current pass over the data, we repeat the process for the items it contains. If it is empty, native formats of individual rules are finalized and added to the result file, which is then offered for download. Algorithm 1 summarizes the steps involved in exporting rules.

It must be noted that the algorithm for exporting rules described above assumes that cross-reference paths are always acyclic, i.e., that there are no paths pointing from a cross-referenced rule to a rule that precedes it in a given chain of cross-references. If there are circular dependencies between rules, the algorithm will not terminate. When adding cross-references to slots of combination groups via the OutputBuilder, the system currently does not check for circular dependencies; it only prevents rules from cross-referencing themselves. This issue will have to be addressed in future versions of the system: Even if the export algorithm were able to break dependency chains in case of circularities, we would not consider rules belonging to these chains to be well-defined, as at least one rule would be missing a portion of its intended output.

We have already mentioned that due to caching, native formats of individual rules have to be computed only once, and that the algorithm performs best with respect to the number of passes it has to make over the data if no rule cross-references another rule that is located at a higher index in the input list. From a performance perspective, however, it is also important to mention input characteristics which result in the worst possible behavior from the algorithm: If there is a chain of cross-references that starts from the first rule to process and spans the whole list, the algorithm can not generate plain text representations for more than one rule in each pass it makes over the data. As an example, let

$$R_1 \rightarrow R_2 \rightarrow \ldots \rightarrow R_n$$

where $R_x \rightarrow R_y$ denotes a cross-reference from rule R_x to rule R_y. With this type of data, only the last rule in the list can be processed because *all* other rules depend on rules that follow them. Consequently, only a single rule is removed from the list of rules still to process in each pass over the data, resulting in a total run time of

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

where i denotes the number of rules touched in a given pass. Should cases like this turn out to occur frequently in practice, processing order can be optimized by incorporating knowledge about cross-reference chains into the export algorithm.

B.3.1 Native Format: Names and Descriptions

Rule names and descriptions are not part of the original syntax for rewriting rules developed in the ALIZ-E project. In order to make sure specific rules are easily located in `.trf`

rule need to be able to access the plain text representation of its RHS separately. Generation of the final format is delayed until all rules have been processed.

[73]Note that since the original list of rules is sorted and we are processing the items it contains in order, the newly generated list of rules still to process will automatically be sorted according to the same criteria.

Algorithm 1 Export algorithm

```
procedure exportRules(rules)
    for all rules do
        setLHS(rule)
        setRHS(rule)
    end for
    rules ← sortByCrossRefs(rules)
    rules ← setNativeFormats(rules, ∅)
    nativeFormats ← ∅
    for all rules do
        nativeFormat ← getNativeFormat(rule)
        add(nativeFormats, nativeFormat)
    end for
    return nativeFormats
end procedure
procedure setNativeFormats(rulesToProcess, processed)
    if len(rulesToProcess) = 0 then
        return processed
    end if
    stillToProcess ← ∅
    for all rulesToProcess do
        crossRefs ← getCrossRefs(rule)
        if len(crossRefs) = 0 then
            setNativeFormat(rule)
            add(processed, rule)
        else
            if all(crossRefs) ∈ processed then
                setNativeFormat(rule)
                add(processed, rule)
            else
                add(stillToProcess, rule)
            end if
        end if
    end for
    return setNativeFormats(stillToProcess, processed)
end procedure
```

files produced by our system, we chose to include them in the form of *comments*. The original syntax supports both single-line and multi-line comments. Single-line comments are initiated by two forward slashes (//). For each rule, the exporter generates a single-line comment for the rule's name, and another single-line comment for the associated description. Together, these comments form the *header* of a rule which precedes the rule's body in the exported file:

```
// Name: @firstEncounter
// Description: Agent meets someone for the first time.
...
```

B.3.2 Native Format: LHS

When producing native formats of LHS, the structure of the underlying AVMs must be preserved. The following example shows the JSON representation of a single AVM that is used as a basis for generating a plain text representation conforming to rewriting rule syntax:

```
{
  "pairs": [
    {
      "attribute": { "name": "Context", "type": "complex" },
      "value": {
        "pairs": [
          {
            "attribute": { "name": "Familiarity", "type": "atomic" },
            "value": "no",
          },
          {
            "attribute": { "name": "ChildName", "type": "atomic" },
            "value": "Marco",
          }
        ],
      },
    },
    {
      "attribute": { "type": "atomic", "name": "SpeechAct" },
      "value": "greeting",
    }
  ],
}
```

This AVM contains four features in total: `Context`, `Familiarity`, `ChildName`, and `SpeechAct`. Two of these features (`Context` and `SpeechAct`) belong to the top-level AVM, while the remaining features belong to a nested AVM that represents the value of the `Context` feature. The plain text representation that corresponds to this structure is:

```
<Context>(<Familiarity>no ^ <ChildName>Marco) ^ <SpeechAct>greeting
```

To generate this string, the exporter processes the attribute-value pairs of the top-level AVM one by one. For each pair, it first creates the native format for the `attribute` (i.e., feature) by surrounding its name with angular brackets. If the feature currently being processed is of type `atomic`, the `value` of the pair is simply appended to the string representing the name of the feature, and the exporter moves on to the next pair. On the other hand, when dealing with a `complex` feature the algorithm recurses and starts processing the pairs of the embedded AVM to produce the corresponding value. When the plain text representation of the value becomes available, the exporter encloses it in a pair of parentheses and then starts processing the next pair.

When the exporter encounters a complex feature that embeds an empty AVM or an atomic feature whose value is set to `underspecified`, it only adds the name of the feature (again enclosed in angular brackets) to the string representing the parent AVM.

B.3.3 Native Format: RHS

As mentioned in Chapter 8, the native syntax used for rewriting rules in the ALIZ-E project provides two constructs – `random` and `concatenate` – that we can make use of when generating plain text representations of RHS of rules. One possibility for representing output alternatives would be to generate a full set of output strings for each rule as explained in

Section 13.3.2, and pass each string as an argument to `random`. However, just as we would like to reduce the amount of redundancy rule developers have to deal with when editing rules graphically, we would also like to minimize redundancy in `.trf` files in order to keep them as small as possible. We therefore chose a different approach for building plain text representations of RHS that is tailored more closely to the way rule output is stored on the server.

Due to the fact that output alternatives for individual rules are represented using unique combinations of combination groups, slots, output strings, and parts, there are several scenarios to consider when choosing how to translate any given rule into native rule syntax. First of all, rules can differ in the number of combination groups that they incorporate. If output alternatives are spread out over multiple groups, the decision process that must be encoded using native rule syntax is more complex than if there is only a single combination group. In the latter case the rewriting engine only has to choose between the output alternatives of that particular group during on-line processing. Secondly, on combination group-level there are four alternative scenarios that need to be covered: a given group might (1) be empty, (2) contain only output strings, (3) contain only parts, or (4) have associated with it both output strings and parts. As we will explain shortly, each of these options requires a different representation. Lastly, some rules cross-reference the output of other rules, which further complicates the task of keeping `.trf` files small and names of right-hand side local variables from clashing.

If a combination group is empty, it is simply skipped, as it does not contribute anything to the RHS of its parent rule that is meaningful from the perspective of the rewriting engine. (Note, however, that special handling is necessary for rules whose RHS are *completely* empty. Section B.3.4 below provides information on how the exporter treats these rules.) For groups that are associated with one or more output strings but do not incorporate any parts, the exporter uses the "naive" approach described above. Each string is passed as an individual argument to `random`, and the return value of the `random` function is stored in an RHS-local variable for further processing:

```
###output = random("<string-1>", "<string-2>", ..., "<string-n>")
```

In this scenario a single processing step is enough to generate a plain text representation for a given group and to determine the output produced by that group during on-line processing.

The decision process is more complex if a combination group incorporates two or more non-empty slots. As described above, we would like to avoid generating full output strings from these slots. In order to achieve this goal, we break the decision process down into one decision per slot. That is, we feed individual parts belonging to a slot as arguments to the `random` function, assign the result to a uniquely named variable, and repeat the process for the remaining slots. We then instruct the rewriting engine to `concatenate` the parts that have been chosen to produce the final output string:

```
###part1 = random("<part-1>", "<part-2>", ..., "<part-n>")
###part2 = random("<part-1>", "<part-2>", ..., "<part-n>")
...
###partn = random("<part-1>", "<part-2>", ..., "<part-n>")
###output = concatenate(###part1, ###part2, ..., ###partn)
```

Note that order matters when serializing combination groups with two or more slots to strings in this way: While decisions for individual slots could theoretically be made in any order, the arrangement of arguments to the `concatenate` function must match the original order of slots as determined by their `positions` in the parent group. This is necessary to ensure that the output string chosen for a specific group is syntactically correct and part of the intended set of output alternatives for the parent rule.

The previous examples have shown plain text representations for groups using only one of two possible ways for defining rule output. If a combination group contains both a number of output strings *and* a number of non-empty slots, we combine the approaches described above as follows:

```
###x = random("<string-1>", "<string-2>", ..., "<string-n>")
###part1 = random("<part-1>", "<part-2>", ..., "<part-n>")
###part2 = random("<part-1>", "<part-2>", ..., "<part-n>")
...
###partn = random("<part-1>", "<part-2>", ..., "<part-n>")
###y = concatenate(###part1, ###part2, ..., ###partn)
###output = random(###x, ###y)
```

The exporter first produces a plain text representation that instructs the rewriting engine to choose among the output strings belonging to the current group (`###x`). It then generates the format we chose for representing output alternatives stored in slots. As a last step, it includes a line that instructs the rewriting engine to choose between the output string selected from the set of regular output strings and the output strings produced by concatenating a random part from each slot.

It should be noted that the exporter currently does not take into account how the number of regular output strings compares to the number of output strings that can be generated from the slots of a given group. If their ratio is close to 1, this does not cause specific strings to be chosen more often than others during on-line processing. However, if the difference between these numbers is large and the number of strings in one of the two sets is small, strings from the smaller group will be preferred by the rewriting engine. This is because on combination group-level, the probability for a specific string to be chosen is $\frac{1}{n} \times \frac{1}{2}$, where n is the number of strings in the same set. If n is large for one of the two sets but not for the other, the probability for an individual string from the smaller set to be chosen ($\frac{1}{n}$) approaches 0.5 (i.e., the probability for a specific set to be chosen), while the probability for a specific string from the larger set to be chosen is much smaller. As a result, the intelligent agent is more likely to repeat itself across situations requiring application of the same rules.

If the RHS of a rule incorporates multiple combination groups, the exporter produces an appropriate representation for each one of them, numbering `output` variables in the process to avoid name clashes. It then instructs the rewriting engine to choose between the outputs of individual groups to obtain the final output:

```
...
###output1 = ...
...
###output2 = ...
...
...
...
###outputn = ...
###output = random(###output1, ###output2, ..., ###outputn)
```

With respect to probabilities for individual strings to be chosen when representing the output of a rule with multiple combination groups in this manner, we must note that they are influenced by the extent to which numbers of strings vary across individual groups.

The argument is very similar to the one presented for string selection on group-level above. Across groups, the probability for a string x to be chosen is

$$p_{g_p}(x) \times \frac{1}{m}$$

where $p_{g_p}(x)$ denotes the probability for x to be chosen within its parent group g_p and m denotes the total number of groups. If $p_{g_p}(x)$ varies a lot across groups because some groups contain large numbers of strings and others only a few, output strings from groups with fewer strings will be chosen with a higher probability than output strings from groups with large numbers of strings.

Finally, before we conclude our discussion of plain text representations of rule output we must explain how output of cross-referenced rules is integrated into the formats presented above. A naive approach for treating cross-references in the context of rule export would be to pass all output strings generated by a single rule as arguments to `random` when generating plain text representations for slots cross-referencing that particular rule. However, there are two major disadvantages to this approach. First of all, depending on the overall count of cross-references and the number of output strings associated with cross-referenced rules, the size of the resulting `.trf` file could increase considerably. Secondly, additional processing would be necessary to generate full sets of output strings for cross-referenced rules. To make matters worse, the process of computing full sets of output strings would have to be repeated for rules being cross-referenced by multiple rules, or its results would have to be cached. While the latter approach would reduce the number of processing steps, it would increase the amount of memory required for producing `.trf` files.

Because of these disadvantages, we chose a different approach for exporting rule output containing cross-references to one or more rules: As explained above, after serializing the output associated with a given rule to an appropriate string, that string is cached for later use. Whenever the exporter encounters a rule that cross-references the rule from which the string was generated, the string is *prepended* to the plain text representation of the output of that rule. In fact, the exporter collects and concatenates native formats of the outputs of all cross-referenced rules before it starts to build a plain text representation for the output of the rule it is currently processing. By putting native formats of cross-referenced rules first (and prefixing names of `output` variables with rule names) we ensure that they can be referenced from output native to the current rule as many times as necessary, without having to include them more than once. Consider the following example[74]:

```
// Cross-referenced rule (@rule1):
###output1 = random("<string-1>", "<string-2>", ..., "<string-n>")
###x = random("<string-1>", "<string-2>", ..., "<string-n>")
###part1 = random("<part-1>", "<part-2>", ..., "<part-n>")
...
###partn = random("<part-1>", "<part-2>", ..., "<part-n>")
###y = concatenate(###part1, ###part2, ..., ###partn)
###output2 = random(###x, ###y)
###rule1_output = random(###output1, ###output2)

// Cross-referenced rule (@rule2):
###output1 = random("<string-1>", "<string-2>", ..., "<string-n>")
###x = random("<string-1>", "<string-2>", ..., "<string-n>")
###part1 = random("<part-1>", "<part-2>", ..., "<part-n>")
```

[74]Lines starting with // are comments and have been inserted for illustration purposes. They are not part of the format produced by the exporter. Also, native formats of rules normally do not contain any empty lines.

```
...
###partn = random("<part-1>", "<part-2>", ..., "<part-n>")
###y = concatenate(###part1, ###part2, ..., ###partn)
###output2 = random(###x, ###y)
###rule2_output = random(###output1, ###output2)

// Cross-referencing rule (@rule3):
// First group: Output strings only
###output1 = random("<string-1>", "<string-2>", ..., "<string-n>")
// Second group: Parts only
###part1 = random(###rule2-output, "<part-1>", ..., "<part-n>")
###part2 = random("<part-1>", "<part-2>", ..., "<part-n>")
...
###partn = random("<part-1>", "<part-2>", ..., "<part-n>")
###output2 = concatenate(###part1, ###part2, ..., ###partn)
// Third group: Both
###x = random("<string-1>", "<string-2>", ..., "<string-n>")
###part1 = random("<part-1>", "<part-2>", ..., "<part-n>")
###part2 = random(###rule1-output, ###rule2-output, "<part-1>", ..., "<part-n>")
...
###partn = random("<part-1>", "<part-2>", ..., "<part-n>")
###y = concatenate(###part1, ###part2, ..., ###partn)
###output3 = random(###x, ###y)
###rule3_output = random(###output1, ###output2, ###output3)
```

The RHS of the rule shown here cross-references two rules: `@rule1` is referenced from the second slot of the third combination group, and `@rule2` is referenced from the first and second slot of the second and third combination group, respectively. Note that output variables for intermediate results can be reused across groups: The rewriting engine will always pass the latest values of individual variables to `random` and `concatenate`. However, in order to be able to reference specific rules from any slot, names of variables storing the final output chosen for individual rules must be unique, which is why the rule exporter prefixes them with names of their parent rules. This is done for every rule by default. As a result, RHS of cross-referenced rules can be incorporated into RHS of any other rule without having to change the names of the final `output` variables, and without having to check for each rule whether or not it is cross-referenced by other rules before processing it.

Lastly, it must be noted that with this approach, parts that are native to a given slot will be preferred over parts that are imported from other rules via cross-references if the number of native parts is larger than the number of cross-references associated with the slot. This is because during on-line processing, the task of selecting output strings from cross-referenced rules is completed before any slots containing references to these rules are processed. Should this behavior turn out to be problematic in practice, a modified version of the naive approach for incorporating output of cross-referenced rules described above could be used instead.

B.3.4 Handling Incomplete Rules

All rules listed in the rule browser are added to the `.trf` file that is produced when users click the Export button, irrespective of whether they are complete or not. After all, users might want to fill in LHS and/or RHS of incomplete rules manually. In order to make sure, however, that rule files containing incomplete rules can be used by the rewriting engine right away – *without* any kind of manual post-editing – the exporter comments out bodies of incomplete rules when generating native formats.

We mentioned above that there is support for both single-line and multi-line comments. The exporter uses multi-line comments for rule bodies. Multi-line comments are initiated by `/*` and terminated by `*/`. No special syntax is used to mark lines between these delimiters. The following example shows the plain text representation of a rule whose LHS and RHS are completely empty:

```
// Name: @rule4
// Description: ...
/*
:dvp ^
->
# ^ :canned ^ <stringOutput>.
*/
```

B.3.5 Summary

The exporter produces `.trf` files that include rule names and descriptions, which facilitates the task of locating specific rules using search functionality provided by standard text editors. Redundancy in rule files is minimized by adjusting plain text representations of RHS depending on configurations of combination groups, slots, output strings, and parts. Output of cross-referenced rules is prepended to native output associated with RHS of rules, which ensures that it can be referenced from native output as many times as necessary without having to be included more than once. In its current version, the export algorithm completely avoids generating full sets of output strings for individual rules. With respect to the number of processing steps that are necessary to produce plain text representations of all rules, negative effects caused by the presence of cross-references are mitigated by sorting rules by cross-reference counts prior to generating native formats.

A potential drawback of minimizing plain text representations of rules as much as possible is that depending on exact configurations of components associated with a given RHS, probabilities of specific output strings to be chosen for realization are not guaranteed to be distributed uniformly. If this turns out to be problematic in practice, the behavior of the export algorithm could be modified to work with full sets of output strings.

Lastly, since users might want to specify LHS and/or RHS of some rules manually, rules are included in `.trf` files even if their definitions are incomplete. To ensure that rule files produced by the exporter are immediately usable by the rewriting engine, the bodies of incomplete rules are commented out.

B.4 Search Algorithm

In this appendix we explain the process for finding rules matching a set of search terms in detail. We first describe the overall behavior of the algorithm and how it changes depending on the type of search criteria specified by users. Since the process of matching rules against sets of features differs from the process of matching rules against sets of strings, we then discuss the steps involved in each of these processes separately. Lastly, we touch briefly on a scenario that the current implementation of the search algorithm can not handle.

At the highest level of granularity, the algorithm for searching the rule base has to support four different scenarios representing different combinations of search terms. These scenarios are listed in Table 13.

The first scenario has already been addressed above: If all input fields are empty, no search is performed, and users are immediately notified that they need to specify at least one search term for the operation to succeed. In order to reduce complexity, different

Input scenario	Features	Strings
(1)	No	No
(2)	Yes	No
(3)	No	Yes
(4)	Yes	Yes

Table 13: Possible inputs to the search algorithm

sequences of steps are performed in the remaining scenarios to arrive at the final result. For instance, if the search terms entered by a user consist solely of features and/or feature-value pairs (scenario (2)), there is no need for the system to examine every single rule: As explained in Section 9.2.1, rules are connected to features they embed via relationships of varying length. To obtain a set of rules embedding one or more features, the graph stored in the database can be traversed backwards from feature nodes matching search terms to parent rules. This makes the process of producing results for the second scenario linear ($O(n)$) in the number of features entered as search terms. A similar approach can be used for feature-value pairs. Section B.4.1 below provides more detailed information about the steps involved in finding rules matching sets of features.

It is not possible to use the same approach with the third input scenario. This is because as mentioned earlier, not all output is stored in the form of output strings, and strings to search for might cross part boundaries. For example, a user might want to search for the string `hello, how are you?`. Rules incorporating this string only "implicitly" – by referencing parts `hello,` and `how are you?` from adjacent slots – would not be included in the results for this query if the system were to examine output strings and parts separately. As a result, optimal recall can only be guaranteed for this scenario by checking the output of each rule for matches, making this scenario the worst starting point for the search algorithm. Note, however, that it is still possible to make some small optimizations that will take effect for rules whose output includes strings currently being searched for. We describe the steps involved in finding rules matching sets of strings in detail in Section B.4.2 below.

Lastly, if the input to the search algorithm includes both features and strings (scenario (4)), the two approaches described above can be combined to significantly reduce the search space that must be considered when matching rules against strings: In this case we first obtain the set of rules containing all features specified in the query. If this set is empty, we can conclude that there are no rules satisfying the original query and stop processing. This is possible because as mentioned previously, search terms are treated as conjunctive constraints. Consequently, if a rule fails to reference one or more features specified as part of a search query, it will never be included in the set of results, even if it matches all strings specified in the query. If the set of rules matching features is not empty, we continue by checking the rules it contains for the remaining search terms (i.e., strings), excluding from the final result set any rules that do not match one or more of these terms. Proceeding in this fashion we do not have to consider all rules that are currently defined when checking the rule base for rules matching strings: Unless each rule that is currently defined references all features specified in the query, the search space is narrowed down to a smaller set of rules before the system starts processing search strings. Algorithm 2 summarizes how the set of matching rules is obtained in the context of a single search operation.

Algorithm 2 Search algorithm

```
 1: procedure SEARCH(features, strings)
 2:     if LEN(features) $> 0$ then
 3:         rules $\leftarrow$ FINDMATCHINGRULES(features)
 4:         if LEN(rules) $= 0 \vee$ LEN(strings) $= 0$ then          ▷ Scenario (2)
 5:             return rules
 6:         else                                                    ▷ Scenario (4)
 7:             return FINDMATCHINGRULES(rules, strings)
 8:         end if
 9:     else if LEN(strings) $> 0$ then                              ▷ Scenario (3)
10:         rules $\leftarrow$ GETALLRULES
11:         return FINDMATCHINGRULES(rules, strings)
12:     else                                                        ▷ Scenario (1)
13:         return $\emptyset$
14:     end if
15: end procedure
```

B.4.1 Finding Rules Matching Features

Because we chose to represent and store rule data in the form of a property graph, we were able to delegate most of the work involved in finding rules matching a set of features to the database. In fact, the processing logic for obtaining rules matching a certain feature differs from the processing logic for obtaining rules matching a certain feature-value *pair* only in the Cypher query that is sent to the database.

When collecting matching rules we first check for each feature whether a value has been specified for it. If no value has been specified, we query the database for the set of rule nodes that are connected to the current feature via an LHS relationship followed by an arbitrarily long sequence of HAS relationships:

```
MATCH (s:Rule)-[:LHS]->()-[:HAS*]->(e:Feature)
WHERE e.name='<name>'
RETURN s;
```

In this query, <name> stands for the name of the current feature. Note that the query will not only fail if there are no rules satisfying these conditions, but also if there is no feature whose name matches the search term entered by the user. This means that we do not have to send a separate query checking for the existence of a given feature to the database.

If the feature currently being considered is part of a feature-value pair, this query has to be extended to further constrain the result set. In particular, the path from a rule node to the feature in question must extend to an appropriate value node via a HAS relationship for the rule to be considered a match. Furthermore, the rule property of the HAS relationship between feature and value must match the uuid property of the candidate rule: As explained in Section 9.2, features can be associated with different values in different rules. The rule property on a HAS relationship between a feature and a value records in which rule these entities are associated with each other. Incorporating these constraints into the previous query, the Cypher query for obtaining rules matching a specific feature-value pair looks like this:

```
MATCH (s:Rule)-[:LHS]->()-[:HAS*]->(e:Feature)-[r:HAS]->(v:Value)
WHERE e.name = '<fname>' AND v.name='<vname>' AND r.rule = s.uuid
RETURN s;
```

Algorithm 3 Finding rules matching features and feature-value pairs

```
 1: procedure findMatchingRules(features)
 2:     ruleSets ← $\emptyset$
 3:     for all features do
 4:         if hasValue(feature) then
 5:             value ← getValue(feature)
 6:             ruleSet ← executeQuery(feature, value)
 7:         else
 8:             ruleSet ← executeQuery(feature)
 9:         end if
10:         add(ruleSets, ruleSet)
11:     end for
12:     return intersect(ruleSets)
13: end procedure
```

We have mentioned before that we only consider rules to match a query if they incorporate *all* search terms submitted via the search interface. After obtaining sets of matching rules for all features and feature-value pairs, the system intersects these sets to produce the final result. Algorithm 3 summarizes the steps involved in finding the set of rules matching a non-empty set of features and feature-value pairs.

B.4.2 Finding Rules Matching Strings

As explained above, the procedure for finding rules matching a set of strings is more complex than the procedure for finding rules matching a set of features because we need to take into account that search strings may cross part boundaries. Still, we would like to avoid generating full sets of output strings whenever possible, or at least delay this step until other options have been exhausted: When iterating over a set of candidate rules to find matches, we first check for each search string whether there is an output string node with matching `contents` that is connected to the RHS node of the rule currently being examined via a path of length 2. This is accomplished using the following Cypher query (where `<str>` denotes the search string):

```
MATCH p=(s:RHS)-[*2]->(t:OutputString)
WHERE s.uuid='<uuid>' AND lower(t.content) =~ lower('.*<str>.*')
RETURN p;
```

If there is no output string node satisfying these conditions, the search string is added to a list of strings which have not been found, and disregarded otherwise. If this list remains empty after all search strings have been considered, we can conclude that the current rule is a match and add it to the result set immediately, without any further processing. On the other hand, if the list is *not* empty, we have to keep looking for the search strings it contains. As a first step we obtain a list of combination group nodes associated with the RHS of the current rule. Starting with the first group, we then obtain an ordered list of all slots that this group contains and generate all possible combinations of parts from it[75]. After completing these preparatory steps, each string from the list of strings still to find is matched against the list of generated output strings until a match is found or the latter list runs out. If a given search string matches none of the output strings generated from

[75]Note that since rule search has been implemented as a server-side operation, it can not reuse the code that handles generation of full output strings on the client-side. However, the algorithm that the system uses to generate output strings from slots for the purpose of matching them against search strings is identical to the algorithm described in Section 13.3.2 above.

Algorithm 4 Finding rules matching strings

```
procedure findMatchingRules(rules, strings)
    ruleSet ← ∅
    for all rules do
        stringsNotFound ← ∅
        for all strings do                              ▷ Examine output strings first
            if ¬connected(rule, string) then
                add(stringsNotFound, string)
            end if
        end for
        if len(stringsNotFound) = 0 then
            add(ruleSet, rule)
            continue                ▷ No more strings to find, so process next rule
        else
            groups ← getGroups(rule)
            for all groups do                                   ▷ Examine groups
                slots ← getSlots(group)
                ostrs ← generateOutputStrings(slots)
                for all stringsNotFound do
                    for all ostrs do
                        if contains(ostr, stringNotFound) then
                            remove(stringNotFound, stringsNotFound)
                            break
                        end if
                    end for
                end for
                if len(stringsNotFound) = 0 then
                    add(ruleSet, rule)
                    break           ▷ No more strings to find, so process next rule
                end if
            end for
        end if
    end for
    return ruleSet
end procedure
```

the current group, it remains in the list of strings still to find. If it does, we remove it from that list. This process is repeated for the next combination group belonging to the RHS of the current rule if and only if there is at least one search string left which has not been found. If the list of strings still to find is empty when the process completes for a given group, the current rule is considered a match and can be added to the result set immediately. Any remaining combination groups are left untouched in this case. If one or more search strings can not be found in any of the combination groups associated with a rule, the algorithm proceeds to the next rule without adding that rule to the result set. The steps involved in finding all rules matching a specific set of strings is summarized in Algorithm 4.

B.4.3 A Word on Cross-References

Following the steps for matching rules against search strings described above, the system is able to achieve optimal recall even if search strings cross part boundaries. It must be noted, however, that there is a scenario that the search algorithm currently does not cover: If the output of a given rule X contains a specific search term y, and another rule Z references rule X but does not itself contain search term y, only X will be included in the results for a query for rule containing y. Z will not be part of the results. This is because cross-references between rules are currently not expanded when searching for rules matching a set of strings.

B.4.4 Summary

This chapter described the algorithm that our system currently uses to compile sets of rules matching user queries. Depending on whether the input to the search algorithm consists only of features, only of values, or contains both features and values, different optimizations are made to reduce the search space. If features are part of the query, the algorithm always determines the set of rules incorporating these features first: Since the property graph stored in the database can be traversed backwards from features to rules, the system does not need to check all rules for matches. For queries which also contain one or more strings, this reduces the search space to the set of rules matching the set of features specified in the input to the search algorithm. On the other hand, if a query does not contain any features, all rules must be checked for matching output to guarantee optimal recall. In this case the system delays generation of full sets of output strings (which is necessary because search strings can cross part boundaries) by checking regular output strings associated with individual rules for matches first. If output strings do need to be generated from slots because one or more search strings have not been found in the set of regular output strings associated with a rule, this is done on a group-by-group basis: Search is aborted for a given rule as soon as a single match has been found for all search terms. Remaining groups are disregarded and the algorithm moves to the next rule. Summing up, the search algorithm performs worst if the input does not contain any features and none of the rules stored in the database match the set of strings specified as search terms. It performs best if the input does not contain any strings. Cross-references are currently not expanded when searching for rules matching a set of strings.

C Training Phase: Tasks

C.1 Creating rules

You can create new rules by clicking the “New” button in the navigation bar at the top.

Task

Create a rule called `secondEncounter` with a description that reads:

`Agent meets a boy named Marco for the second time.`

C.2 Adding features

You are now looking at the LHS of the rule you just created.
You can add features to the LHS of a rule by dragging them from the feature inventory on the left side of the screen and dropping them on the placeholder that reads “Drop feature here . . . ”.
To get more information about a feature you can hover over it with the mouse.

Task

Add the following features to the LHS of the current rule:

- `Encounter`
- `Familiarity`
- `ChildName`
- `SpeechAct`

C.3 Setting values

Each feature you just added has a drop-down menu next to it. This menu contains a list of all possible values that the corresponding feature can take.
You can set (or change) the value of a given feature by expanding the associated drop-down menu and selecting the desired value from the list.

Tasks

Set the value of the `Encounter` feature to `notfirst`.

Set the value of the `Familiarity` feature to `yes`.

Set the value of the `ChildName` feature to `marco`.

Set the value of the `SpeechAct` feature to `greeting`

C.4 Removing features

You can remove features from the LHS of a rule by clicking the `x` button that appears when hovering over their names with the mouse.

Tasks

Add another feature (any feature you like) to the LHS of the `secondEncounter` rule. If you want, you can also set it to a specific value.

Remove the feature you added in the previous step as described above.

C.5 Renaming rules

You can rename a rule by

1. double-clicking its name
2. entering the new name into the text input field that appears
3. clicking the “OK” button

Task

Make the name of the `secondEncounter` rule a bit more generic by renaming it to `encounterFamiliarPerson`.

C.6 Changing descriptions

You can change the description of a rule by

1. double-clicking it
2. entering the new description into the text input field that appears
3. clicking the “OK” button

Task

Before moving on to editing the RHS, make the description of the `encounterFamiliarPerson` rule match its name by changing it to

```
Agent meets a boy named Marco. The boy and the agent are familiar
with each other.
```

C.7 Switching between LHS and RHS

To switch from the LHS of the current rule to the RHS you can click the "OutputBuilder" button in the navigation bar. Similarly, to switch from the RHS back to the LHS you can click the "InputBuilder" button in the navigation bar.

Task

The following tasks will focus on editing rule output, so please switch to the "OutputBuilder" now.

C.8 Adding output strings

You can add a new output string to a rule by

1. clicking the placeholder that reads "Add more content"
2. entering the output string
3. clicking the "Add" button

Task

Add the following string to the `encounterFamiliarPerson` rule:

`Hello again! Good to see you.`

C.9 Modifying output strings

You can modify individual output strings by

1. double-clicking any word in the string
2. modifying the contents of the text input field that appears
3. clicking the "OK" button

Task

Modify the string you entered in the previous task to read:

`Hello Marco! Good to see you again.`

C.10 Removing output strings

You can remove individual output strings from an RHS by clicking the `x` button that appears when hovering over them with the mouse.

Tasks

Add another output string (with any content you like) to the RHS of the `encounterFamiliarPerson` rule. If you want, you can also modify the string after adding it.

Remove the output string you added in the previous step as described above.

C.11 Splitting output strings

To speed up creation of content, output strings can be split up into smaller units called *parts*. When splitting a single output string, each of the resulting parts is added to a different *slot* depending on its position in the original output string. ODE will take care of computing all possible combinations of parts from different slots to create the full list of output strings for a given rule.

To split an output string into two parts you can click the area *between* the last word that belongs to the left part and the first word that belongs to the right part.

Task

Split the string that reads

```
Hello Marco! Good to see you again.
```

such that the first part reads

```
Hello Marco!
```

and the second part reads

```
Good to see you again.
```

C.12 Adding parts

You can add parts to individual slots by

1. clicking the placeholder at the bottom of the slot
2. entering the part
3. pressing “Enter”

Tasks

Add the following part to “Slot 1”:

```
Hi Marco!
```

Add the following part to “Slot 2”:

```
How are you doing today?
```

C.13 Showing output

To show all output strings belonging to a given rule you can click the "Show output" button.

Tasks

Click the "Show output" button. You will see four output strings, each of which consists of a part from "Slot 1" followed by a part from "Slot 2".

Click the `x` button in the top right corner to dismiss the list of output strings.

C.14 Modifying parts

You can modify individual parts by

1. double-clicking them
2. modifying the contents of the text input field that appears
3. pressing "Enter"

Task

Modify the part that reads

```
Good to see you again.
```

to say

```
How have you been?
```

C.15 Removing parts

You can remove individual parts from a given slot by clicking the `x` button that appears when hovering over them with the mouse.

Tasks

Add another part (with any content you like) to any slot you like. If you want, you can also modify the part after adding it.

Remove the part you added in the previous step as described above.

C.16 Adding slots

You can create additional slots by clicking the placeholder next to the rightmost slot. When you hover over this placeholder with the mouse it says "Add slot . . . ".

Tasks

Add a single slot to the table.

Add the following parts to the slot created in the previous step:

`I've missed you.`

`It's been a while!`

Click the "Show output" button. You will see that by adding two parts you were able to double the number of output strings.

C.17 Removing slots

You can remove individual slots by clicking the `x` button that appears when hovering over their headers. The total number of slots must not drop below 2, so removing slots will only work if there are at least three slots.

Task

Remove the slot that you added for the previous task ("Slot 3").

C.18 Using the parts inventory

Every part that you create manually or by splitting an output string is added to the parts inventory automatically. The parts inventory is displayed on the left side of the screen.

You can reuse parts to create additional content for the RHS of a rule by dragging them from the inventory to one of the various placeholders:

1. To add a part as an output string you can drop it on the placeholder that reads "Add more content . . .".

   Task

   Add the part that reads
   `Hey,`
   as an output string.

2. To *extend* an existing output string you can drop individual parts from the inventory on the placeholder displayed to the right of the string.

   Task

   Use the part that reads
   `how are you?`
   to extend the output string you added in the previous step. The result should read:
   `Hey, how are you?`

3. To add a part to a specific slot you can drop it on the placeholder at the bottom of the slot.

Task

Add the part that reads
`What's up?`
to "Slot 2".

C.19 Working with multiple groups

All of the work so far has been done in the context of a single *combination group* ("Group 1"). A single group is sufficient for the content you have added so far, but it can sometimes be useful to have more than one group available for specifying the output of a given rule. As an example, consider the following set of strings:

```
I am glad to see you again.
I am glad to see you today.
I am happy to see you again.
I am happy to see you today.
Good to see you again.
Good to see you today.
```

These strings combine well with the parts listed in "Slot 1" of the group you have been working on, so it would make sense to add each one of them as a part to the second slot of this group. It is obvious, however, that these strings are very similar to each other and it would be inefficient to type them in one by one. Ideally, they should be split up like this:

`I am happy`	`to see you`	`again.`
`I am glad`		`today.`
`Good`		

Given its current contents, you can not use "Group 1" to do this. If you were to extend the slots of this group to look like this:

`Hello Marco!`	`How have you been?`	`to see you`	`again.`
`Hi Marco!`	`How are you doing today?`		`today.`
	`What's up?`		
	`I am happy`		
	`I am glad`		
	`Good`		

the output of the `encounterFamiliarPerson` rule would include output strings like

```
Hello Marco! How have you been? to see you again.
Hello Marco! How have you been? to see you today.
Hello Marco! How are you doing today? to see you again.
Hello Marco! How are you doing today? to see you today.
...
```

However, using an additional group the strings can be combined with the contents of the first slot of "Group 1" and split as shown above:

`Hello Marco!`	`I am happy`	`to see you`	`again.`
`Hi Marco!`	`I am glad`		`today.`
	`Good`		

C.19.1 Adding groups

You can add a new group to an RHS by clicking the `+` button in the header of any existing group.

Task

Add a new group to the `encounterFamiliarPerson` rule.

C.19.2 Removing groups

You can remove a group from an RHS by clicking the `x` button in the group's header.

Task

Remove the group you added in the previous step ("Group 2").

C.19.3 Copying groups

To use the contents of a given group as a starting point for creating more content in another group you can click the button between the `+` and `x` buttons in the header of the group you want to copy.

Tasks

Copy "Group 1".

Modify the new group to look like this:

`Hello Marco!`	`I am happy`	`to see you`	`again.`
`Hi Marco!`	`I am glad`		`today.`
	`Good`		

Use the "Show output" button to check for errors and duplicate output strings.

C.20 Browsing rules

To view a list of all existing rules you can click the "Browse" button in the navigation bar.

Task

Click the “Browse” button.

From this interface you can

- click an entry to view a read-only version of the corresponding rule
- use the controls that appear when hovering over a given entry to
  - view a list of similar rules (i.e., rules whose LHS contain similar features)
  - jump to the InputBuilder
  - jump to the OutputBuilder
  - delete the corresponding rule
- use the “Filter ...” field to filter entries by name and description

Task

Take a moment to familiarize yourself with the controls of the “Browse” interface. Please do not delete any rules.

D Assignment: Rules

D.1 Rule 1: Asking user to play

Name: `requestPlayQuiz`
Description: The robot asks a boy named Marco if he wants to play the quiz game with him. The robot is familiar with this boy, and the two of them have played this game before.

D.2 Rule 2: Asking user to repeat answer

Name: `answerNotUnderstood`
Description: Marco has answered a question that the robot asked him as part of the quiz game. The robot did not understand what Marco said; he apologizes to Marco and asks him to repeat his answer.

D.3 Rule 3: Trying again

Name: `requestAnswerRetry`
Description: Marco's answer to the robot's question was not correct. The robot asks him if he wants to try again.

D.4 Rule 4: Asking for feedback about the game

Name: `requestFeedbackQuiz`
Description: Marco and the robot are done playing the quiz game. The robot asks Marco if he enjoyed playing the game.

D.5 Rule 5: Saying goodbye

Name: `sayGoodbye`
Description: The game has ended. The robot says goodbye to Marco.

E Assignment: Reference Manual

ODE Reference

July 20, 2014

Contents

1 Creating rules

You can create new rules by clicking the "New" button in the navigation bar at the top.

2 Adding features

You can add features to the LHS of rules by dragging them from the feature inventory on the left side of the screen and dropping them on the placeholder that reads "Drop feature here . . . ".
To get more information about a feature you can hover over it with the mouse.

3 Setting values

After adding a feature to the LHS of a rule, the system displays a drop-down menu to the right of the feature that contains a list of all possible values that this feature can take. By default, feature values are set to `underspecified`. You can change the current value of a feature by expanding the drop-down menu and selecting the desired value from the list.

4 Removing features

You can remove features from the LHS of a rule by clicking the `x` button that appears when hovering over their names with the mouse.

5 Renaming rules

You can rename a rule by

1. double-clicking its name
2. entering the new name into the text input field that appears
3. clicking the "OK" button

6 Changing descriptions

You can change the description of a rule by

1. double-clicking it
2. entering the new description into the text input field that appears
3. clicking the "OK" button

7 Switching between LHS and RHS

To switch from the LHS of the current rule to the RHS you can click the "OutputBuilder" button in the navigation bar. Similarly, to switch from the RHS back to the LHS you can click the "InputBuilder" button in the navigation bar.

8 Adding output strings

You can add a new output string to a rule by

1. clicking the placeholder that reads "Add more content"
2. entering the output string
3. clicking the "Add" button

9 Modifying output strings

You can modify individual output strings by

1. double-clicking any word in the string
2. modifying the contents of the text input field that appears
3. clicking the "OK" button

10 Removing output strings

You can remove individual output strings from an RHS by clicking the `x` button that appears when hovering over them with the mouse.

11 Splitting output strings

To split an output string into two parts you can click the area *between* the last word that belongs to the left part and the first word that belongs to the right part.
The left part will be added to "Slot 1" and the right part will be added to "Slot 2". If "Slot 1" and "Slot 2" do not exist, they will be created first.

12 Adding parts

You can add parts to individual slots by

1. clicking the placeholder at the bottom of the slot
2. entering the part
3. pressing "Enter"

13 Showing output

To show all output strings belonging to a given rule you can click the "Show output" button.
To dismiss the list of output strings, click the `x` button in the top right corner of the list.

14 Modifying parts

You can modify individual parts by

1. double-clicking them
2. modifying the contents of the text input field that appears
3. pressing "Enter"

15 Removing parts

You can remove individual parts from a given slot by clicking the `x` button that appears when hovering over them with the mouse.

16 Adding slots

You can create additional slots by clicking the "Add slot" button.

17 Removing slots

You can remove individual slots by clicking the `x` button that appears when hovering over their headers. The total number of slots must not drop below 2, so removing slots will only work if there are at least three slots.

18 Using the parts inventory

Every part that you create manually or by splitting an output string is added to the parts inventory automatically. The parts inventory is displayed on the left side of the screen in the OutputBuilder.
You can reuse parts to create additional content for the RHS of a rule by dragging them from the inventory to one of the various placeholders:

1. To add a part as an output string you can drop it on the placeholder that reads "Add more content . . . ".
2. To *extend* an existing output string you can drop parts from the inventory on the placeholder displayed to the right of the string.
3. To add a part to a specific slot you can drop it on the placeholder at the bottom of the slot.

19 Working with multiple groups

19.1 Adding groups

You can add a new group to an RHS by clicking the `+` button in the header of any existing group.

19.2 Removing groups

You can remove a group from an RHS by clicking the `x` button in the group's header.

19.3 Copying groups

To use the contents of a given group as a starting point for creating more content in another group you can click the button between the `+` and `x` buttons in the header of the group you want to copy.

20 Browsing rules

To view a list of all existing rules you can click the "Browse" button in the navigation bar.
From this interface you can

- click an entry to view a read-only version of the LHS and RHS of the corresponding rule
- use the controls that appear when hovering over a given entry to
  - view a list of similar rules (i.e., rules whose LHS contain similar features)
  - jump to the InputBuilder
  - jump to the OutputBuilder
  - delete the corresponding rule
- use the "Filter ..." field to filter entries by name and description

F Questionnaire

1. What is your highest level of education?

   ____
2. Are you currently a student?
   - ☐ yes, my major is: ____
   - ☐ no
3. What is your native language?

   ____
4. Which of the following terms describes your knowledge about computers most accurately?
   - ☐ casual user
   - ☐ power user
   - ☐ programmer
5. After completing the training phase, did you feel like you had a basic understanding of the tasks involved in creating rules?
   - ☐ yes
   - ☐ no
6. Was it difficult for you to finish the assignment within 45 minutes?
   - ☐ Yes
     - ☐ ... because there were too many rules to create
     - ☐ ... because I was having problems with the system
     - ☐ ... because I was having a hard time coming up with variations
     - ☐ ... because ____
   - ☐ No
7. On average, did you spend more time working on LHS or RHS?
   - ☐ LHS
   - ☐ RHS
   - ☐ I spent equal time on LHS and RHS
   - ☐ I am not sure
8. Did you run into any errors during the assignment?
   - ☐ yes
   - ☐ no

   If you did, describe what you were trying to do when the error occurred:

   ____
   ____
   ____
9. Were you confused by the behavior of the system at any given point while completing the assignment?

- ☐ yes
- ☐ no

If so, please describe the behavior(s) that confused you, and how you would have expected the system to behave:

____
____
____

10. Did you notice any behavior(s) that you felt were slowing you down during the assignment?

    - ☐ yes
    - ☐ no

    If so, please describe the behavior(s) and what you think could be done to improve them:

    ____
    ____
    ____

11. What additional features should be added to the system to make editing rules faster and/or more convenient?

    ____
    ____
    ____

12. Do you have any additional comments? (optional)

    ____
    ____
    ____

Bibliography

[1] Greg Bates. Modal and modeless boxes in web design. `http://webdesign.tutsplus.com/articles/modal-and-modeless-boxes-in-web-design`, May 2012.

[2] Tilman Becker. Practical, template-based natural language generation with TAG. In *Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Frameworks*, 2002.

[3] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, 1st edition, 2009.

[4] Yevgeniy Brikman. Play Framework: async I/O without the thread pool and callback hell. `http://engineering.linkedin.com/play/play-framework-async-io-without-thread-pool-and-callback-hell`, March 2013.

[5] Stephan Busemann. Best-first surface realization. In *Proceedings of the Eighth International Natural Language Generation Workshop (INLG '96)*, 1996.

[6] Stephan Busemann. eGram - A grammar development environment and its usage for language generation. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC)*, 2004.

[7] Stephan Busemann. Ten Years After - An update on TG/2 (and friends). In *Proceedings of the 10th European Workshop on Natural Language Generation*, 2005.

[8] Stephan Busemann and Helmut Horacek. A flexible shallow approach to text generation. In Eduard Hovy, editor, *Proceedings of the 9th International Language Generation Workshop*, pages 238–247, 1998.

[9] Maria Fernanda Caropreso, Diana Inkpen, Shahzad Khan, and Fazel Keshtkar. Novice-friendly natural language generation template authoring environment. In *Proceedings of the 22nd Canadian Conference on Artificial Intelligence*, pages 195–198, 2009.

[10] Songsak Channarukul. YAG: A natural language generator for real-time systems. Master's thesis, University of Wisconsin-Milwaukee, 1999.

[11] Songsak Channarukul, Susan Weber McRoy, and Syed S. Ali. YAG: A template-based text realization system for dialog. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(6):649–659, 2001.

[12] Songsak Channarukul, Susan Weber McRoy, and Syed S. Ali. JYAG + IDEY: A template-based generator and its authoring tool. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 994–995, 2002.

[13] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report, San Jose, California*, RJ599, 1969.

[14] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[15] Chris Coyler. What are the benefits of using a CSS framework? `http://css-tricks.com/what-are-the-benefits-of-using-a-css-framework/`, October 2008.

[16] Guozhu Dong, Leonid Libkin, Jianwen Su, and Limsoon Wong. Maintaining transitive closure of graphs in SQL. *International Journal of Information Technology*, 5, 1999.

[17] Kemal Erdogan. A model to represent directed acyclic graphs (DAG) on SQL databases. `http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG`, January 2008.

[18] I. Fette and A. Melnikov. RFC 6455: The WebSocket protocol. `https://www.ietf.org/rfc/rfc6455.txt`, 2011.

[19] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, May 2002.

[20] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000.

[21] David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly Media, 6th edition, 2011.

[22] Armando Fox and David Patterson. *Engineering Software as a Service: An Agile Approach Using Cloud Computing.* Strawberry Canyon, 2014.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[24] Albert Gatt and Ehud Reiter. SimpleNLG: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation*, ENLG '09, pages 90–93, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[25] Christopher Horne. Glossary of big data terminology. `http://www.iqubemarketing.com/glossary-big-data-terminolgy/`, August 2013.

[26] Aravind K. Joshi and Yves Schabes. Handbook of formal languages, vol. 3. chapter Tree-adjoining Grammars, pages 69–123. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[27] jQuery UI Homepage. Description of functionality provided by jQuery UI. `http://jqueryui.com/`, 2014.

[28] Dan Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition.* Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, Englewood Cliffs, NJ, 2nd (pearson international edition) edition, 2009.

[29] Santosh Kumar K. *Spring And Hibernate.* McGraw-Hill Education (India), 1st edition, 2009.

[30] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 6. Auflage.* Oldenbourg, 6th edition, 2006.

[31] Bernd Kiefer. Content planner module for talking robots (version 1.2). Technical report, German Research Center for Artificial Intelligence (DFKI), March 2012.

[32] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, August 1988.

[33] Hans-Ulrich Krieger and Ulrich Schäfer. TDL - a type description language for hpsg. part 2: User guide. Technical report, DFKI, 1994.

[34] Ivana Kruijff-Korbayová, Georgios Athanasopoulos, Aryel Beck, Piero Cosi, Heriberto Cuayáhuitl, Tomas Dekens, Valentin Enescu, Antoine Hiolle, Bernd Kiefer, Hichem Sahli, Marc Schröder, Giacomo Sommavilla, Fabio Tesser, and Werner Verhelst. An event-based conversational system for the nao robot. In *Proceedings of IWSDS 2011: Workshop on Paralinguistic Information and its Integration in Spoken Dialogue Systems*, pages 125–132. Springer, 2011.

[35] Ivana Kruijff-Korbayová, Heriberto Cuayáhuitl, Bernd Kiefer, Marc Schröder, Piero Cosi, Giulio Paci, Giacomo Sommavilla, Fabio Tesser, Hichem Sahli, Georgios Athanasopoulos, Weiyi Wang, Valentin Enescu, and Werner Verhelst. Spoken language processing in a conversational system for child-robot interaction. In *Proceedings of WOCCI 2012 - Workshop on Child, Computer and Interaction Satellite Event of IN-*

TERSPEECH, September 2012.

[36] Ivana Kruijff-Korbayová, Elettra Oleari, Clara Pozzi, Stefania Racioppa, and Bernd Kiefer. Analysis of the responses to system-initiated off-activity talk in human-robot interaction with diabetic children. In *Proceedings of the 18th Workshop on the Semantics and Pragmatics of Dialogue*, pages 90–97. Heriot Watt University, 2014.

[37] Bruce Lawson and Remy Sharp. *Introducing HTML5.* Pearson Education, 2011.

[38] P. Leach, M. Mealling, and R. Salz. RFC 4122: A Universally Unique IDentifier (UUID) URN namespace. `http://www.ietf.org/rfc/rfc4122.txt`, 2005.

[39] Susan W. McRoy, Songsak Channarukul, and Syed S. Ali. Yag: A template-based generator for real-time systems. In *Proceedings of the First International Conference on Natural Language Generation - Volume 14*, INLG '00, pages 264–267, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[40] Karthik Sankaran Narayan, Charles Lee Isbell Jr., and David L. Roberts. DEXTOR: Reduced effort authoring for template-based natural language generation. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2011.

[41] Olga Nikitina. Italian utterance planning rules for Simon scenario. Technical report, German Research Center for Artificial Intelligence (DFKI), 2011.

[42] A. Osmani. *Developing Backbone.js Applications.* Building Better JavaScript Applications. O'Reilly Media, 2013.

[43] Jonas Partner, Aleksa Vukotic, and Nicki Watt. *Neo4j in Action (MEAP).* Manning Publications Co., 2014.

[44] DFKI project page for ALIZ-E. ALIZ-E mission statement. `www.dfki.de/lt/project.php?id=Project_576`, 2014.

[45] Leonard Richardson and Sam Ruby. *RESTful Web Services.* O'Reilly Media, 2008.

[46] Matthias Rinck. Ein Metaregelformalismus für TG/2. Master's thesis, Department of Computational Linguistics, Saarland University, 2003.

[47] Peter Rob and Carlos Coronel. *Database Systems: Design, Implementation, and Management.* Cengage Learning, 8th edition, 2007.

[48] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases.* O'Reilly Media, 2013.

[49] Clinton W. Smullen III and Stephanie A. Smullen. An experimental study of AJAX application performance. *JSW*, 3(3):30–37, 2008.

[50] Holger Stenzhorn. XtraGen – A natural language generation system using XML- and Java-technologies, 2002.

[51] The Neo4j Team. The Neo4j manual v2.0.0. Technical report, Neo Technology, 2013.

[52] Mariët Theune. Natural language generation for dialogue: system survey, May 2003.

[53] Annarita Tranfici. Responsive web design: Custom grid layouts. `http://www.sitepoint.com/responsive-web-design-grid-layouts/`, February 2013.

[54] Annarita Tranfici. Understanding responsive web design: Cross-browser compatibility. `http://www.sitepoint.com/understanding-responsive-web-design-cross-browser-compatibility/`, March 2013.

[55] Wolfgang Wahlster, Norbert Reithinger, and Anselm Blocher. SmartKom: Multimodal communication with a life-like character. In *Proceedings of Eurospeech 2001*, pages 1547–1550, 2001.

[56] Susan Weber McRoy, Songsak Channarukul, and Syed S. Ali. An augmented template-based approach to text realization. *Natural Language Engineering*, 9(4):381–420, 2003.