VOnDA

A Framework for Implementing Reactive Dialogue Agents

Version 3

Bernd Kiefer, Anna Welker

October 23, 2025

Contents

Chapter 1

Purpose and Goal

VOnDA is a framework to implement the dialogue management functionality in dialogue systems. Although domain-independent, VOnDA is tailored towards dialogue systems with a focus on social communication, which implies the need of a long-term memory and high user adaptivity. VOnDA's specification and memory layer relies upon (extended) RDF/OWL, which provides a universal and uniform representation, and facilitates interoperability with external data sources. The starting point for designing VOnDA was the Information State-Update approach to dialogue systems, which has a strong resemblance to the Belief-Desire-Intention approach to Artificial Agents. Thus, it is not surprising that VOnDA can also serve as a base formalism for agent functionality. A comparison of VOnDA to other dialogue management systems and other related information can also be found in Kiefer et al. [2019].

VOnDA consists of three parts: A programming language tailored towards the specification of reactive rules and transparent RDF data store usage, a compiler that turns source code in this language into Java code, and a run-time core which supports implementing dialogue management modules using the compiled rules.

The framework is domain-independent. It was originally designed for multi-modal human-robot interaction, but there is currently no specific functionality in the core to either support the multi-modality nor the human-robot interaction. The architecture (see figure 1.1) of the framework is open and powerful enough to add these things easily.

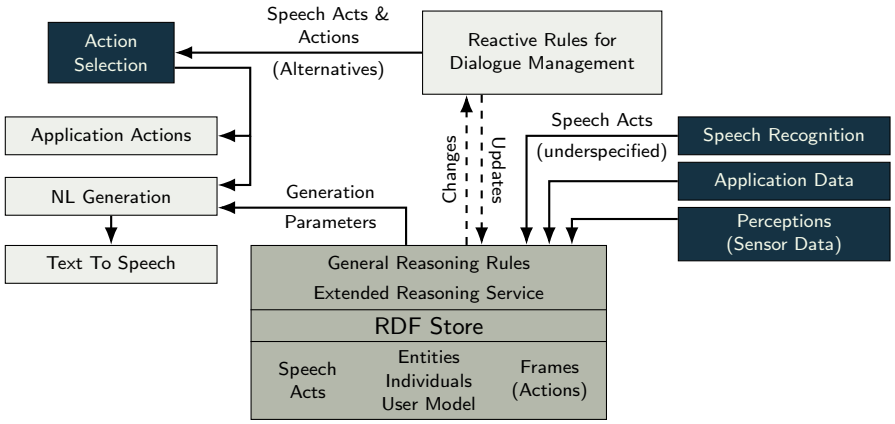


Figure 1.1: Schematic VOnDA agent

At the base is an RDF store which implements the belief state and takes incoming sensor and interaction data and stores it as RDF data. The data format is backed by a data specification in the form of an ontology developed as part of the dialogue manager, making the data (via the

specification) available to all other components.

The RDF store and reasoner of choice used in VOnDA is HFC [Krieger, 2013]. For further details about the general functionalities of HFC see chapter 3.6. Section 2.1 contains an example how HFC is used as database in a VOnDA project.

The dialogue manager gets several inputs from various sources, the ones already used are: input from automatic speech recognition (ASR) or typed natural language input, user parameters, like name, age, hobbies, etc. but also more dynamic ones like mood or health data, and also triggers from high-level planning.

The second major component is the rule processor for the dialogue management rules. When new data is added, a set of declaratively specified reactive rules will propose dialogue moves or other actions and send these proposals to the action selection mechanism. This mechanism selects the "best" of the proposed actions and sends it back. If the proposed action results in dialogue acts, these are turned into verbal output and gestures with the help of a multimodal generation component, which retrieves parameters from the RDF database to adapt the generation to the user's likings, and can also take into account sensor data such as her or his estimated mood. The rules themselves can use all available data, the incoming new data, but also the interaction history and other data stored in the RDF database to make decisions.

The last major component contains the language interpretation module (not explicitly shown in the picture), which turns spoken or written utterances into dialogue acts, possibly with an intermediate step that involves a more elaborate semantic format, and a multimodal generation component, which converts outgoing dialogue acts into natural language utterances and gestures.

Chapter 2

A Hands-On Example

This chapter will walk you through the creation of a simple interaction manager, which you can either create yourself, or just follow by looking at the toy system named `ChatCat`, which has its own github repository at `https://github.com/bkiefer/vonda-chatcat`. More complex examples are planned to be added soon, to demonstrate how to connect VOnDA to external NLP components and external (robotic) systems.

The simplest version of an interaction manager analyses natural language coming from the user, and generates natural language and gestures for the robot resp. its virtual replacement, an avatar. Generation is based on incoming stimuli, like speech or text input, or high-level action requests coming from some strategic planning component, or any other sensor input, if available.

In this tutorial, we will create a very simple example system that has a database representation of itself and the user it will interact with. It can greet the user, ask for his/her name and say goodbye.

2.1 Setting up the Basic Data Structures

A dialogue system aiming for social interaction does need some kind of memory representation. Therefore, the first step of building your dialogue manager with VOnDA will be to set up your basic data structures in the form of an OWL ontology. The RDF database serves two purposes: it contains 1) the data structure definitions of the data that is used in the dialogue system, like a `User` class and the properties that are associated with it, and 2) the (dynamic) data objects, which are created or extended on-the-fly when the dialogue system is running. The advantage of using and RDF store for the data structure specifications lies in its flexibility. Extending or changing them is easy, which is important since your system will be evolving and becoming more and more elaborate.

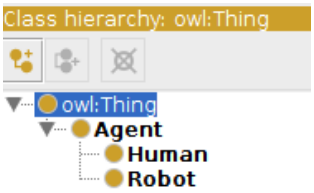
For the specification of dialogue acts, we recommend that you use the dialogue act hierarchy provided in `examples/chatcat/src/main/resources/ontology/dialogue.nt`, which is based on the ISO standard DIT++ hierarchy, as well as two `default` files which are necessary for basic OWL functionality in HFC.

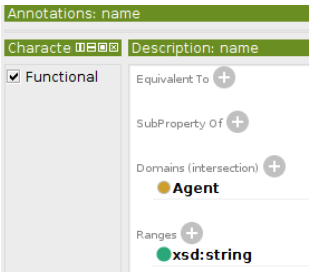
2.1.1 Creating N-Triples Files

HFC currently loads data only from files in the `N-Triples` format. The majority of RDF software packages works with the more common RDF/XML format, which can be automatically translated with a simple shell script that is provided with the example (`examples/chatcat/ntcreate.sh`). This script uses Raptor [Beckett, 2017], which is provided for example in the `raptor2-utils .deb` package. This tutorial uses screenshots from

Protégé [Stanford Research, 2017], which we used for the creation of the basic ontology, but you can just use your favourite RDF/OWL IDE.

First, create a new file that includes an RDF class `Agent`, and two subclasses of this class, `Robot` and `Human`. After that, create a (functional) data type predicate `name` for the class `Agent` with the range `xsd:string`.





As we do not know the user a priori, we will have the system create an instance for him/her at run-time. However, we know our robot in advance, so we create an instance of the class Robot and name it *Robert*, and then convert the new ontology using Raptor to `N-Triples` (i.e., with the `ntcreate.sh` script). Now, a HFC configuration file (in YAML syntax) must be created, as described in the next section.

Important: If you are using Protégé, you should save the file in RDF/XML Syntax; the script will not work properly otherwise.

2.1.2 Creating a HFC Configuration File

Which ontology files are loaded into a HFC instance, and which reasoning rules are applied is specified in a YAML configuration file. It contains also various other settings for the available HFC parameters. Figure 2.1 shows the config file used for the `chatcat` example. In the following, we will explain some aspects of the config file in detail.

Settings Some of the settings are currently required for VOnDA projects to work correctly, namely `minArgs`, `maxArgs`, `eqReduction`, `addTimestamps`, the tuple file `default.eqred.nt` and the rule file `default.eqred.quads.rdl`. If you create your own HFC config file for a VOnDA project, please copy those as shown here.

`noOfCores` determines how many threads HFC will use maximally if processing can be parallelized. A reasonable number depends on your computer's load that is independent from VOnDA, but should not exceed half of the physical cores available.

For most applications concerning dialogue management, it is important to specify a `persistenceFile` to save the data between two runs of the system. This file can be put in any location, it will be created automatically in the specified place. If your application relies on inter-session memory, you probably don't want it to reside in some temporary directory. All new information that your dialogue system enters into the database will be collected here. The persistency file can also be used to find out which tuples have been created, i.e, for on-line and post-mortem debugging. If you want to wipe the memory of your system after each session, simply delete this file, or do not specify a `persistenceFile`.

namespaces This section contains abbreviations for ontology namespaces. The abbreviation `dial` in figure 2.1, allows to refer to `<http://www.dfki.de/lt/onto/common/dialogue.owl#Accept>` using `<dial:Accept>` instead, for example in queries to the database. As you can see, we included a shortcut for our chatcat ontology here.

```
minArgs: 3
maxArgs: 4
noOfCores: 4
#noAtoms=100000
#noTuples=500000
persistenceFile: ../../../../persistent.nt
characterEncoding: UTF-8
eqReduction: true
addTimestamps: [0]
shortIsDefault: true

namespaces:
  # namespaces for XSD, RDF, RDFS, and OWL are already defined
  dial: 'http://www.dfki.de/lt/onto/common/dialogue.owl#'
  cat: 'http://www.semanticweb.org/anna/ontologies/2018/3/chatcat#'

tupleFiles:
  # the axiomatic triples for OWL-Horst w/ EQ reduction, needed if eqReduction
  # is true
  - default.eqred.nt
  # the project sub-ontologies
  - dialogue.nt
  - chatcat.nt

ruleFiles:
  # we need special rules for transaction time (mixture of triples/quadruples)
  - default.eqred.quads.rdl
```

Figure 2.1: An exemplary HFC `config.yml` file

tupleFiles Here all ontology files have to be listed that should be loaded into the knowledge base on start-up. The persistency file, if you have specified one, will be loaded automatically. You should also include the file `dialogue.nt` which, as previously mentioned, contains the specifications of the dialogue acts usually used by the VOnDA framework.

ruleFiles This specifies the set of rules that HFC uses for OWL reasoning. Currently, the file `default.eqred.quads.rdl` is required for proper operation of VOnDA, since it relies on the so-called *transaction time* representation, which allows to keep a (possibly) infinite memory, while still preserving a monotonic RDF store, i.e., only adding and never deleting tuples. This representation uses quadruples, where the forth element is the timestamp when the tuple was added to the store. For tuples with infinite resp. universal validity, the timestamp should be set to zero. For further information, please refer to the documentation of HFC.

2.2 Setting up the Basic Java Classes

First, we implement the project's abstract (Java) "agent" class as a subclass of `Agent` in package `de.dfki.mlt.rudimant.agent`, which is part of the run-time library of VOnDA. In this simple case, this is not really necessary, and only provided to allow the functionality of the runtime library's `Agent` class to be modified. If you do this, you will need to specify the name of your specialised `Agent` class as `agentBase` in the config file, otherwise, the topmost generated file will extend `de.dfki.mlt.rudimant.agent.Agent`.

Furthermore, we implement a subclass of `de.dfki.mlt.rudimant.agent.CommunicationHub`. To see an example of what that should contain, take a look in the source folder of `ChatCat`.

The two most important things here are that there is an active connection to a database (as an instance of `RdfProxy`) and that you have an instance of the beforementioned VOnDA `Agent` (or

your extended implementation of it) in your client. Of course this code can not compile until you build your first rule file, i.e., your VOnDA Agent. Then, a `main` has to create an instance of your client and is started using the `startListening()` method.

We recommend to have a look at the classes of the ChatCat system as a base for your own system and extend it. It comes you with a very simple GUI to enter text or dialogue acts which you can use to test your first dialogue steps.

2.3 First Interaction Rules

Now that the basics have been arranged, we are set up for writing our first dialogue management rules. First we want to react to the user greeting the system, what we expect to be happening on startup. In the SRGS file (`src/main/resources/grammars/srgs/chatcat.abnf`), we defined that an utterance of the user like "Hello" will be parsed as the dialogue act `InitialGreeting`, with the proposition `Greet`. We now can define a rule reacting to this utterance:

```
greet_back:
  if (lastDA() <= #InitialGreeting(Greet)) {
    user = new Human;
    if (! saidInSession(#Greeting()) {
      propose("greet_back") {
        emitDA(#ReturnGreeting(Greet));
      }
    }
    lastDAprocessed();
  }
```

This will create a new instance of the RDF class `Human` we defined when setting up the ontology, storing it in a global variable `user` that in our case has been defined in the ChatAgent and will be present during the whole conversation. The check `! saidInSession(#Greeting)` currently doesn't seem to make sense, why this is necessary will be obvious when we have completed the example. This test already shows an important property of the system: `Greeting` is the superclass of `InitialGreeting` and `ReturnGreeting` in the DIT++ ontology, and the function will return `true`, no matter what type of greeting we gave, since it tests for `subsumption`, like the comparison operators `<=` and `<` that work on dialoge act arguments that we use in the next rule example. More details about how to use this functionality will be given in section 3.3.7

After greeting, we want to find out the user's name. We thus define a rule as follows:

```
ask_for_name:
  if (!user.name && !(myLastDA() <= #WHQuestion(Name))) {
    propose("ask_name") {
      emitDA(#WHQuestion(Name));
    }
    lastDAprocessed();
  }
```

And once we got the answer from the user, we can store this knowledge in the database:

```
remember_name:
  if (lastDA() <= #Inform(Name)) {
    user.name = lastDA().what;
    lastDAprocessed();
  }
```

We currently don't have a person detector, so we assume that someone's here when the system is started. To make sure the conversation starts even if the user doesn't start with a greeting, we use a *timeout* ot implement a system greeting after some time.

```
timeout("robot_starts", 4000) {
 start_conversation:
```

```
  if (! (receivedInSession(#Greeting(top)) || saidInSession(#Greeting(top)))) {
    propose("robot_greets") {
      tod = Date.timeOfDay();
      emitDA(#InitialGreeting(Greet, when={tod}));
    }
  }
}
```

The use of `Date.timeOfDay()` is an example how to use code that is better implemented in plain Java. You can use any Java class adding an `import` statement at the beginning of your `.rudi` file, and you maybe also want to add a type declaration for the return and argument types of the methods that you are using, since currently the compiler can not figure out this information by itself (see section 3.3.9).

You are not limited to static methods as in the example, you can also create a Java helper object in the topmost `rudi` file and use the object's public methods.

These are just enough rules to start a (very short) conversation, so let's compile and try out the new dialogue system.

2.4 How to Compile and Run your Project

Now that we have implemented our first rules, we need to compile them. In the `bin` directory of you VOnDA installation is a script `vondac` that will use a configuration file to compile your project. The most convenient way to use this script is either to establish a softlink in a directory that is already in your `PATH` or to add VOnDA's `bin` directory to it.

While you can pass most necessary parameters on the command line, it is recommended to create a configuration file in YAML syntax (e.g., `config.yml`), which contains these parameters, also because the graphical debugger these to work, too.

All paths in this YAML configuration are resolved relative to the location of the configuration file if they are not absolute directory references (which is discouraged for portability). You can safely add your own options of whatever type (map, list, etc.) into the configuration file, provided you put them under a top-level key, since VOnDA ignores all keys it does not make use of itself, so you can use these to configure the rest of your system, for example host addresses or ports of remote modules, custom resources, etc.

For the VOnDA compiler, your `config.yml` should contain the following parameters:

`inputFile`	Relative to the current location, where is the top-level rule file?
`outputDirectory`	Relative to the current location, where should the compiled classes go?
`agentBase`	The name of your abstract Java Agent class, including package prefix (optional)
`typeDef`	The name of a file containing type definitions for Java fields and methods that the compiler could not find out by itself (optional)
`ontologyFile`	The path to your ontology `.yml`, relative to the current location
`rootPackage`	The topmost package to put the compiled Java classes in
`failOnError`	If `true`, exit compilation on any type error, otherwise continue

Concerning `failOnError`, keep in mind that although VOnDA's type checking is becoming more and more elaborate and reliable, it is by no means complete. In some cases, setting this switch to true might make your project uncompilable although when compiling it ignoring the type errors results in a perfectly sound Java project.

2.4.1 Resolving Name Ambiguities

As you may have noticed looking at chatcat's `config.yml`, there are further parameters used in the compile configuration of our example project:

```
nameToURI:
   Agent: "<cat:Agent>"
nameToClass:
   Date: de.dfki.chatcat.util.Date
```

When trying to compile without the first two lines, you will find that VOnDA produces the warning `"base name Agent can be one of <http://.../chatcat#Agent>, <dial:Agent>, please resolve manually."`.

This is the compiler telling us that when defining the RDF class `Agent` in the database step, we actually redefined an existing class. VOnDA warns us about this and urges us to resolve this ambiguity. Thus, we could either rename our class, or explicitly state which namespace should be accessed whenever the class `Agent` is used, which can be achieved by defining this mapping under `nameToURI`. You can also use this functionality to remap RDF class names: VOnDA will always map the name on the left to the class URI provided on the right.

The second specification serves to resolve type checks in favour of Java instead of RDF classes. The fully specified name is currently not used, but might be used in later versions to generate Java `import` statements.

2.5 A Sample Project Configuration

This is the configuration file of the `vonda-chatcat`[1] example project, which contains runtime and debugger parameters in addition to the previously discussed compiler configuration parameters

```
# Points to the HFC config file
ontologyFile:       src/main/resources/ontology/chatcat.yml

# ########## COMPILE CONFIGURATION SETTINGS ##########
inputFile:          src/main/rudi/ChatCat.rudi
outputDirectory:    src/main/gen-java
agentBase:          de.dfki.chatcat.ChatAgent
typeDef:            ChatAgent.rudi
failOnError:        false
rootPackage:        de.dfki.chatcat
nameToURI:
  Agent: <cat:Agent>
nameToClass:
  Date: de.dfki.chatcat.util.Date
#printErrors: true      # TODO: DESCRIBE
#visualise: true        # produces a graphical representation of the parsed rule files
                        # with type resolution information
#persistentVars: true   # puts compiler into persistent variables mode: local variables
                        # in included rule files also keep their values during runtime

# ########## RUN CONFIGURATION SETTINGS ##########
NLG:
  eng:
    class: de.dfki.mlt.rudimant.agent.nlp.LanguageGenerator
    mapperProject: src/main/resources/grammars/cplanner/allrules-mapper
    generationProject: src/main/resources/grammars/cplanner/allrules
    #translateNumbers: true  # translates numbers to text internally, if, e.g., the TTS
                             # can not do it by itself
NLU:
  eng:
    class: de.dfki.mlt.rudimant.agent.nlp.SrgsParser
    grammar: src/main/resources/grammars/srgs/chatcat.abnf
    converter: src/main/resources/grammars/cplanner/srgsconv
    tokenizer:
      class: de.dfki.mlt.rudimant.agent.nlp.TrivialTokenizer
      toLower: false            # turn all output to lower case
      removePunctuation: true   # remove all punctuation tokens

# ########## DEBUGGER/GUI SETTINGS ##########
debugPort: 9777                 # the port where the Agent talks to the debugger

customCompileCommands:
  mvncompile: mvn install
```

[1] https://github.com/bkiefer/vonda-chatcat

```
  mvncleancmp: mvn clean install
  vcompile: vondac -v -c config.yml
defaultCompileCommand: Compile
```

The configuration can be used both for compilation and to start your compiled system. Although the compile and the runtime phase of VOnDA need different parameters (except `ontologyFile`), e.g., the runtime phase needs NLU and NLG components, you can put all configuration into one `yaml` file for simplicity, since irrelevant keys will be ignored. For runtime initialization, the configuration is passed to the `init` method of your `Agent`, which simplifies configuration also in multi-language settings.

The example uses our SRGS implementation[2] to build a simple NLU and cplan[3] to create natural language from dialogue acts. You can also create your own NLU and NLG by implementing the `Interpreter` and `Generator` Interfaces and adding the appropriate configuration settings.

[2] `https://github.com/bkiefer/srgs2xml`
[3] `https://github.com/bkiefer/cplan`

Chapter 3

Structured Overview

3.1 The VOnDA Compiler

The compiler turns the VOnDA source code into Java source code using the information in the ontology. Every source file becomes a Java class. The generated code will not serve as an example of good programming practice, but a lot of care has been taken in making it still readable and debuggable. The compile process is separated into three stages: parsing and abstract syntax tree building, type checking and inference, and code generation.

The VOnDA compiler's internal knowledge about the program structure and the RDF hierarchy takes care of transforming the RDF field accesses to reads from and writes to the database. Beyond that, the type system, resolving the exact Java, RDF or RDF collection type of arbitrary long field accesses, automatically performs the necessary casts for the ontology accesses.

3.2 VOnDA's Architecture

Figure 3.1 shows the architecture of a runnable VOnDA project.

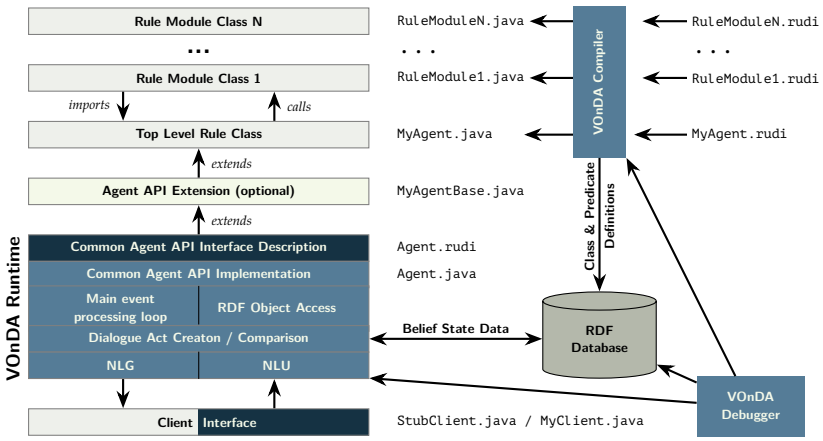


Figure 3.1: Schematic of a VOnDA interaction manager implementation

A basic VOnDA project consists of an ontology, a client interface to connect the communication channels of the application to the agent, and a set of rule files that are arranged in a tree, using `include` statements. The blue core in Figure 3.1 is the run-time system that is part of VOnDA, while all light grey elements are the application specific parts of the agent. A YAML project file contains all necessary information for compilation: the ontology, the top-level rule file and other parameters, like custom compile commands for VOnDA's debugger.

As you can see in the figure, instead of the top-level rule file directly extending from the framework's abstract `Agent` class, you can optionally insert a custom extension of this class and let the rule agent derive from that. This is meant for special cases when the Agent class functionality does not suffice or is not exactly the way you need it (e.g., when more complex synchronisation is needed). Then, you can use the configuration key `agentBase` with a fully specified class name to extend the top-level generated file from your custom agent base class.

The VOnDA compiler translates rule files with the extension `.rudi` to Java files, which means that each file is turned into a public class. During this process, the ontology storing the RDF classes and properties is used to automatically infer types, resolve whether field accesses are actually accesses to the database, etc (see section 3.3.7). Every rule file can define variables and functions in VOnDA syntax which are then available to all included files, since the generated Java classes create objects for each included file, and the references are appropriately handled by the compiler.

For the creation of the objects representing the included rule files, there are two possibilities: in the default case, the object representing the topmost rule file will be created once at the system start and will remain the same until the end of the process, which means that all variables defined there will also be persistent and keep their values, but all other rule file objects will be recreated during each rule cycle, thus resetting their local variables each time. By setting the `persistentVars` configuration flag to `true` (or using the `-p` command line flag), all rule file objects will be only created once, and all local variables will keep their values during runtime. This means that the user/programmer is responsible for resetting them when a rule cycle starts, if that is required.

The current structure assumes that most of the Java functionality that is used inside the rule files will be provided by the `Agent` superclass. There are, however, alternative ways to use other Java classes directly (see section 3.3.9 for further info). You can either use Java helper classes and/or objects, which is the preferred way, or you create your own methods and fields in a custom `agentBase` and thus make them available to all rule files. If the compiler does not pick up the type definitions by itself, you can support it by declaring fields and methods in the type definition file. In the example of figure 3.1, this would be `MyAgent.rudi`).

3.3 The VOnDA Rule Language

VOnDA's rule language looks very similar to Java/C++. There are a number of specific features which make it much more convenient for the specification of dialogue strategies. One of the most important features is the way objects in the RDF store can be used throughout the code: RDF objects and classes can be treated similarly to those of object oriented programming languages, including the type inference and inheritance that comes with type hierarchies.

3.3.1 The Structure of a VOnDA File

A VOnDA file usually consists of a list of (possibly nested) rule statements, often complemented by variable and function definitions. In this section, we will describe the elements of the syntax in more detail.

VOnDA does not require to group statements in some kind of high-level structure like e.g. a class. It is, in fact, not possible to define classes in `.rudi` files at all, rules and method declarations have to be put directly into the rule file. The same holds for every kind of valid (Java-) statement, like assignments, `for` loops etc. From this, the compiler will create a Java class where the methods and rules that are transformed are represented as methods of this specific (generated) class. All other statements as well as auto-generated calls to the methods representing the rules will be put into the `process()` method that VOnDA creates to build a rule evaluation cycle. In doing so, the execution order of all statements, including the rules, is preserved.

This functionality offers possibilities to e.g. define and process high-level variables that you might want to have access to in subsequent rules or to insert termination conditions that prevent some rule executions.

Warning: Variables declared globally in a file will be transformed to fields of the Java class, as was described above. We found that in very rare occasions when running the default mode (only the variables of the top-level rule file keep their values), this can lead to unexpected behaviour when using them in a propose or timeout block as well as changing them in a global statement. As proposes and timeouts will not be exceuted immediately, they need every variable used inside them to be effectively final. VOnDA leaves the evaluation of validness of variables for such blocks to Java. We found that Java might mistakenly accept variables that

are not effectively final, which might lead to completely unexpected behaviour when proposes and timeouts with changed variable values are executed.

The globally defined variables and methods defined in the top-level rule file are always persistent throughout the whole runtime. This is on purpose, and can be used to define persistent variables also usable in lower-level rule files, or to be accessed from other java code. If you set the compiler to the `persistentVars` mode, this will be true for all variables defined in the rule files.

3.3.2 Rules and rule labels

The core of VOnDA dialogue management are the dialogue rules, which will be evaluated at run-time system on every trigger generated from the environment or the internal processing. A rule (optionally) starts with a name that is given as a Java-like label: an identifier followed by a colon. Following this label is an `if`-statement, with optional `else` case. The clause of the `if`-statement expresses the condition under which the rule, or rather the `if` block, is to be executed; in the `else` block you can define what should happen if the condition is `false`, like stopping the evaluation of (a sub-tree of) the rules if necessary information is missing.

```
intro:
  if (introduction) {
    is_user_known:
      if (user.unknown) {
        ask_for_name: if (talkative) askForName();
      } else {
        greetUser();
      }
  }
```

Figure 3.2: A simple rule

Rules can be nested to arbitrary depth, so `if`-statements inside a rule body can also be labelled. The labels are a valuable tool for debugging the system at run-time, as they can be logged live with the debugger GUI (cf. chapter 3.5). The debugger can show you which rules were executed when and what the individual results of each base clause of the conditions were.

3.3.3 The `propose` and `timeout` constructs

There are two statements with a special syntax and semantics: `propose` and `timeout`. `propose` is VOnDA's current way of implementing probabilistic selection. All (unique) propose blocks that are in active rule actions are collected, frozen in the execution state in which they were encountered, like closures known from functional programming languages. When all possible proposals have been selected, a statistical component decides on the "best", whose closure is then executed.

`timeouts` generate the same kind of closures, but with a different purpose. They can for example be used to trigger proactive behaviour, or to check the state of the system after some amount of time, or in regular intervals. A timeout will only be created if there is no active timeout with the same name, otherwise, if the time delay is different than that of the last `timeout` call, the delay will be set to the new value. For special needs, the functions in figure 3.4 are useful to achieve specific behaviours based on `timeouts`.

There are two variants of `timeout`: *labeled timeouts*, like the one in the previous example which run out after the specified time (unless they are cancelled before running out) and then execute their body, and *behaviour timeouts*, where the first argument is a dialogue act (see next section) instead of a label. These are executed either when the specified time is up or the behaviour that was triggered by the dialogue act is finished (e.g. the audio generated by a text-to-speech engine ended, or a specified motion came to an end), whatever comes first.

The following code patterns may help to use the different possibilities that timeouts offer:

```
// timeout triggered exactly once per session
if (! hasActiveTimeout("robot_starts") && ! isTimedOut("robot_starts"))
```

```
if (!saidInSession(#Salutation(Meeting)) {
  // Wait 7 secs before taking initiative
  timeout("wait_for_greeting", 7000) {
    if (! receivedInSession(#Greeting(Meeting))
      propose("greet") {
        da = #InitialGreeting(Meeting);
        if (user.name) da.name = user.name;
        emitDA(da);
      }
  }

  if (receivedInSession(#Salutation(Meeting))
    propose("greet_back") { // We assume we know the name by now
      emitDA(#ReturnGreeting(Meeting, name={user.name});
    }
  }
}
```

Figure 3.3: `propose` and `timeout` code example

`isTimedOut(`*`label`*`)`	returns `true` if a timeout with that label fired. This can be reset only by calling `removeTimeout(`*`label`*`)`, and is especially convenient to implement timeouts that should only be triggered once in a session.
`removeTimeout(`*`label`*`)`	see `isTimedOut(`*`label`*`)`
`cancelTimeout(`*`label`*`)`	cancels an *active* timeout if there is one, has no effect otherwise
`hasActiveTimeout(`*`label`*`)`	returns true if there is an active timeout with that label

Figure 3.4: Functions for fine tuning `timeout` behaviour

```
  timeout("robot_starts", 4000) { ... }

// timeout reoccurring every 1000 milliseconds
if (! hasActiveTimeout("reptimeout"))
  timeout("reptimeout", 1000) { ... }

// ensure that something happens even if the expected condition does not
// become true after 10 seconds
if (! condition && ! hasActiveTimeout("ensure_cond")) {
  timeout("ensure_cond", 10000) {
    if (! condition) {
      // clean up
    }
  }
}
```

3.3.4 Stopping Rule Evaluation

There are multiple ways to stop rule evaluation locally (i.e. skipping the evaluation of the current subtree) or globally (i.e. stopping the whole evaluation cycle). You can skip the evaluation of a specific rule you are currently in with the statement `break label_name;`. This will only stop the rule with the respective label (no matter how deep the break statement is nested in it), such that the next following rule is evaluated next.

If the evaluation is cancelled with the keyword `cancel`, all of the following rules in the current file will be skipped (including any included rule files). If the keyword `cancel_all` is used, none of the following

rules, neither local nor higher in the rule tree, will be evaluated. This is the VOnDA way of deciding to not further evaluate whatever triggered the current evaluation cycle and will mostly be used as an 'emergency exit', as the dialogue rules should be rejecting any non-matching trigger by themselves.

To leave `propose` and `timeout` blocks, you need to use a `return` statement without return value, as they are only reduced representations of normal function bodies.

A detailed description of how the rules of a VOnDA project are evaluated will follow in section 3.4.1.

3.3.5 RDF access and functional vs. relational properties

```
1 user = new Animate;
2 user.name = "Joe";
3 set_age:
4 if (user.age <= 0) {
5   user.age = 15;
6 }
```

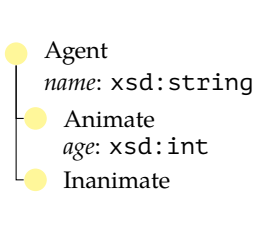


Figure 3.5: Ontology and VOnDA code

Figure 3.5 shows an example of VOnDA code, and how it relates to RDF type and property specifications, schematically presented on the right. The domain and range definitions of properties are picked up by the compiler and then used in various places, e.g., to infer types, do automatic code or data conversions, or create "intelligent" boolean tests, like in line 4, which will expand into two tests, one testing for the existence of the property for the object, and in case that succeeds, a test if the value is smaller or equal than zero.

The connection of VOnDA to the ontology loaded into HFC during compile time enables the compiler to recognise the correct RDF class to create a new instance when creating a new RDF object with `new`, similar to a Java object, and to resolve field/property accesses to all RDF instances. Field accesses as shown in line 2 and 3 of figure 3.6 will be analysed and transformed into database accesses. Object creation or assignments, i.e. changes to existing objects, will be immediately reflected in the database.

```
1 c = new Child;
2 nm = c.name;
3 c.name = "new name";
4 Set middle = c.middleNames;
5 c.middleNames += "John";
6 c.middleNames -= "James";
7 c.middleNames = null;
```

⇒

```
1 c = palAgent._proxy.getClass("<dom:Child>").getNewInstance(palAgent.DEFNS);
2 nm = c.getString("<upper:name>");
3 c.setValue("<upper:name>", "new name");
4 middle = c.getValue("<upper:middleNames");
5 c.getValue("<upper:middleNames>").add("John");
6 c.getValue("<upper:middleNames>").remove("James");
7 c.clearValue("<upper:middleNames>");
```

Figure 3.6: Examples for an RDF property access

VOnDA will also draw type information from the database. If the name property of the RDF class `Child` is of type `String`, exchanging line 2 by the line `int name = c.name` will result in a warning of the compiler. During this process, the compiler will automatically also use the correspondence of XSD and Java types shown in figure 3.7.

If there is a chain of more than one field/property access, every part is tested for existence in the target code, keeping the source code as concise as possible (see also figure 3.8 in section 3.3.7). Also for reasons of brevity, the type of a new variable needs not be given if it can be inferred from the value assigned to it.

<xsd:int>	Integer	<xsd:integer>	Long
<xsd:string>	String	<xsd:byte>	Byte
<xsd:boolean>	Boolean	<xsd:short>	Short
<xsd:double>	Double	<xsd:dateTime>	Date
<xsd:float>	Float	<xsd:date>	XsdDate
<xsd:long>	Long	<xsd:dateTimeStamp>	Long

Figure 3.7: Standard RDF types and the Java types as which they will be recognized

Moreover, VOnDA determines whether an access is made using functional or relational predicates and will handle it accordingly, assuming a collection type if necessary. In the rule language, the operators `+=` and `-=` are overloaded. They can be used with sets and lists as shortcuts for adding and deleting objects. `a += b` will be compiled to `a.add(b)` and `a -= b` results in `a.remove(b)`, as shown in figure 3.6.

Creating RDF instances with `new` In figure 3.6, `c = new Child;` is used to create a new RDF instance of class `Child`. Since every RDF instance must be part of a namespace, a default namespace string field `DEFNS` is defined in the `Agent` class. Its default value is `"def"`, which corresponds to the long URI `http://www.dfki.de/lt/default.owl#`. You have to supply this name in the initialisation method (`Agent.init`) of your agent (it has to be a valid short namespace name, make sure you have the appropriate namespace mapping in place)[1], and all RDF objects created in the generated VOnDA code will then be created in your custom namespace.

Parameterising field access To be able to *parameterise* the field access to RDF objects, VOnDA has a special mechanism. Instead of the above `c.middleNames`, you could have done the following:

```
mid = "<upper:middleNames>";
Set middle = c.{mid};
```

if you use `{<exp>}` in a field access, the compiler assumes `exp` to evaluate to a `String`, and the string resulting from evaluating `exp` at runtime will be used as if you would have specified an identifier with the same name. Be aware that this only works for access to RDF objects, and that you have to take care of all type checking and casting by yourself, since the compiler can not figure out in advance which properties you will access.

3.3.6 Casting types in VOnDA

The syntax for casting expressions explicitely to a specific type is slightly different than the Java syntax, for reasons of better readability and easier treatment in parsing the code. VOnDA uses the `isa` keyword as infix operator, similar to the `cast()` in C++, so a Java-style cast `((Child)c)` will be `isa(Child, c)` in VOnDA syntax

3.3.7 Type inference and overloaded operators

VOnDA allows static type assignments and casting, but in many cases these can be avoided. If, for example, the type of the expression on the right-hand side of a declaration assignment is known or inferrable, it is not necessary to explicitely state it.

You can also declare variables final.

A time-saving feature of VOnDA which also improves readability is the automatic completion of boolean expressions in the clauses of `if`, `while` and `for` statements. As it is obvious in these cases that the result of the expression must be of type boolean. VOnDA automatically fills in a test for existence if it is not. When encountering field accesses, it makes sure that every partial access is tested for existence (i.e., not `null`) to avoid a `NullPointerException` in the runtime execution of the generated code.

[1] see also the `init` method of `ChatAgent` from the example project

```
if (! c.user.personality.nonchalance){ ... }          ⇒

if (!((((c != null) && (c.user != null)) && (c.user.personality != null))
      && (c.user.personality.nonchalance != null))) {
  ...
}
```

Figure 3.8: Transformation of complex boolean expressions

Be aware that the expansion in the figure only occurs if the multiple field access is used as boolean test. In the following example, the first clause in the boolean expression should not be omitted, since a `NullPointerException` could still occur because the second clause does not trigger an automatic test for existence of the `status` of `activity`:

```
if (activity.status && activity.status == "foo"){ ... }
```

Many operators are overloaded, especially boolean operators such as **<=**, which compares numeric values, but can also be used to test if an object is of a specific class, for subclass tests between two classes, and for subsumption of dialogue acts.

```
if (sa <= #Question){
  ...
}
```

```
if (sa.isSubsumedBy(new DialogueAct("Question")) {
  ...
}
```

Figure 3.9: Overloaded comparison operators

3.3.8 Dialogue Acts

A central functionality of a dialogue system is receiving and emitting dialogue acts that result from a user utterance resp. can be transformed to natural language by a generation component to communicate with the user. In VOnDA, the function for sending dialogue acts is called `emitDA`.

The dialogue act representation is an internal feature of VOnDA. We are currently using the DIT++ dialogue act hierarchy [Bunt et al., 2012] and shallow frame semantics along the lines of FrameNet [Ruppenhofer et al., 2016] to represent dialogue acts. The natural language understanding and generation units connected to VOnDA should therefore be able to generate or, respectively, process this representation.

```
emitDA(#Inform(Answer, what={solution}));
```

Figure 3.10: Dialogue Act Example

Figure 3.10 shows the dialogue act representation in VOnDA, as passed to, e.g., the `emitDA` function. `Inform(...)` will be recognized by VOnDA as dialogue act because it has been marked with `#`. It will then create a new instance of the class DialogueAct that contains the respective modifications. As a default, arguments of a DialogueAct creation (i.e., character strings on the left and right of the equal sign) are seen as and transformed to constant (string) literals, because most of the time that is what is needed. Surrounding a character sequence with curly brackets (`{}`) marks it as an expression that should be evaluated. In fact, arbitrary expressions are allowed inside the curly brackets, and converted automatically to a string, if necessary and possible.

While this kind of shallow semantics is enough for many applications, we already experience its shortcomings when trying to handle, for example, social talk. One of the next improvements will be the extension of Dialogue Acts to allow for embedded structures.

3.3.9 Declaring External Methods And Fields

As mentioned before, you can use every method or field you declare in some Java class or your custom `Agent` subclass implementation in your VOnDA code. Their declaration in the Java/rudi interface looks like

a normal Java field or method definition (cfg. figure 3.11). It is possible to use generics in these definitions, although their names are, for complexity reasons, restricted to one single uppercase letter.

```
MyType someVariable;          // field of class MyType
MyType someFun(ClA a, ClB b); // method someFun with signature
                              // (ClA, ClB) --> MyType
                              // return type can be void for a procedural method
```

Figure 3.11: Defintions of existing Java fields and methods for VOnDA

There is a variety of standard Java methods called on Java classes that VOnDA automatically recognises, like e.g. the `substring` method for Strings. If you find that you need VOnDA to know the signature of a new method or a field that is defined in some other class that is not your `Agent` subclass, you can provide VOnDA with knowledge about them by adding their definition to the interface as follows:

```
#SomeClass myType Function(typeA a); // declaration of SomeClass method
#SomeClass myType someVar;           // declaration of SomeClass field
#List<T> T get(int a);               // use of Generics is possible and
                                     // used in type inference
```

Figure 3.12: Definition of a non-static method of Java objects

It is important to realise that all declarations in the interface are only compile time information for VOnDA and will not be transferred to the compiled code, whereas declarations in the rule code itself will also appear in the compiled code.

Functional constructs

VOnDA allows to specify `Function` arguments, where lambda constructions can then be used in the code. Currently, the functions listed in figure 3.13 are pre-defined in the `Agent` class. If you for example want to filter a set of RDF objects by a sub-type relation, you can write:

```
des = filter(agent.desires, lambda(d) (d <= UrgentDesire));
```

or

```
des = filter(agent.desires, lambda(d) { return (d <= UrgentDesire); });
```

```
boolean some(Collection<T> coll, Function<Boolean, T> pred);
boolean all(Collection<T> coll, Function<Boolean, T> pred);
List<T> filter(Collection<T> coll, Function<Boolean, T> pred);
List<T> sort(Collection<T> coll, Function<Integer, T, T> comp);
Collection<T> map(Collection<S> coll, Function<T, S> f);
int count(Collection<T> coll, Function<Boolean, T> pred);
T first(Collection<T> coll, Function<Boolean, T> pred);
```

Figure 3.13: Functions that take lambda expressions as an argument

Using rules of other rule files with `include`

With the `include` statement, e.g. `include RuleFile;`, which needs to appear at the root level of rule files, the (compiled) rules and definitions in `RuleFile.rudi` and its included files are added at the position of the `include`. This is not a macro-like inclusion, the compiler generates Java classes for every `include`.

This inclusion has two important effects. On the one hand, it triggers the compilation of the included file at exactly this point, such that any fields and methods known at this time will be available in the included file. On the other hand, all the rules contained in the included file will be inserted in the run-time rule cycle

at the specific position of the `include`, that is, in the resulting code the `process()` method of the generated code for the included file will be executed.

`include` makes it possible to organize the rules and local declarations of a project into meaningful sub-units. This supports modularity, as different subtrees of the `include` hierarchy can easily be added, moved, taken away or re-used in different projects.

Java-Code verbatim in rule files

To maintain simplicity, VOnDA intentionally only provides limited Java functionalities. Whatever is not feasible in VOnDA source code should be done in methods in the wrapper class or other helper classes.

In cases where this is not possible and you urgently need a functionality of Java that VOnDA cannot parse or represent correctly, you can use the verbatim inclusion feature. Everything between `/*@` and `@*/` will be treated like a multi-line Java comment, meaning the content is not parsed or evaluated further. It will be transferred to the compiled code as is into the intended position.

It is strongly discouraged to use this feature extensively, but to use helper objects/classes instead whereever possible. Not only will code written this way likely become unreadable if there is too much of it, it might also not be portable to new VOnDA versions, if the way of generating the Java code changes.

Java `import` statements

You can use `import` statements as in Java syntax in rule files, but only at the very beginning. You should however be aware that VOnDA will not know that these classes have been imported, nor their methods and fields. It will however accept creations of instances of unknown classes, as well as your casting of results of unknown methods. If you want VOnDA to have type information about methods called on instances on one of these classes, you can put this information into your `typeDef` file (see the beginning of this section).

3.4 The Run-Time System

The run-time library contains the basic functionality for handling the rule processing, including the proposals and timeouts, and for the on-line inspection of the rule evaluation. There is, however, no blueprint for the main event loop, since that depends heavily on the host application. The run-time library also contains methods for the creation and modification of shallow semantic structures (`DialogueActs`), and especially for searching the interaction history for specific utterances. Most of this functionality is available through the abstract `Agent` class, which needs to be extended to a concrete subclass for your application.

There is also functionality to talk directly to the HFC database using queries (see section 3.6.1), in case the object view that was described in before is not sufficient or too awkward.

3.4.1 Rule Evaluation Cycle

Your VOnDA rule files form a tree, starting at the top-level file that you specify in the configuration file, and the `included` rule files. The evaluation of the rule starts in the top-level files and proceeds in pre-order through this tree. If you use a `cancel` or `cancel_all` statement (cf. section 3.3.4), the rule evaluation will be either locally or globally stopped.

The set of your reactive VOnDA rules is executed whenever there is a change in the information state, which is stored in the database. These changes can be caused by incoming sensor or application data, intents from the speech recognition, or expired `timeouts`. A rule can have direct effects, like changes in the information state, or system calls. Furthermore, the `proposals`, which are (labeled) blocks of code in a frozen state that will not be immediately executed, but collected, similar to closures.

All rules are repeatedly applied until a fix point is reached: No new proposals are generated and there is no change of the information state in the last iteration. Then, the set of collected proposals is evaluated by a statistical component, which will select the best alternative. This component can be exchanged to make it as simple or elaborate as necessary, which also allows to take into account arbitrary features from the data storage.

At the start of your program, an object of the generated top-level class will be created which will exist as long as the program is executed. When the compiler was used in the default mode, temporary objects will be created for the execution of embedded rules, which cease to exist as soon as all relevant rules of

the subtree have been evaluated. As a consequence, no values created in these objects will survive the rule evaluation if they are not stored in a persistent location (e.g., a top-level variable or the database). When the `persistentVars` config flag was set to `true`, the compiler generates also the objects for embedded rules only on startup and keeps them, making all local variables permanent, which includes the coder's responsibility to reset them for a rule evaluation cycle if necessary.

The embedded rules have access to all the variables and methods declared in higher-level rule files, and all the values produced up to their call (see also 5.1.4)

3.4.2 Functionality Provided by the Run-Time System

The following methods are declared in `src/main/resources/Agent.rudi`; their implementation is provided by Java itself or the VOnDA framework.

Pre-added Java methods

```
#Object boolean equals(Object e);
#String boolean startsWith(String s);
#String boolean endsWith(String s);
#String String substring(int i);
#String String substring(int begin, int end);
#String boolean isEmpty();
#String int length();

#List<T> T get(int a);
#Collection<T> void add(Object a);
#Collection<T> boolean contains(Object a);
#Collection<T> int size();
#Collection<T> boolean isEmpty();
#Map<S, T> boolean containsKey(S a);
#Map<S, T> T get(S a);
#Array<T> int length;
```

Short-hand conversion methods from Agent

```
int toInt(String s);
float toFloat(String s);
double toDouble(String s);
boolean toBool(String s);
String toStr(T i);  // T in (int, short, byte, float, double, boolean)
```

Other Agent methods

```
// Telling the Agent that something changed
void newData();

String getLanguage();

// Random methods
int random(int limit); // return and int between zero and limit (excluded)
float random();        // return a random float between zero and one (excluded)
T random(Collection<T> coll); // select a random element from the collection

long now();     // return the current time since the epoch in milliseconds

Logger logger;  // Global logger instance

// discarding actions and shutdown
void clearBehavioursAndProposals();
void shutdown();
```

Timeouts

```
void newTimeout(String name, int millis);
boolean isTimedOut(String name);
void removeTimeout(String name);
boolean hasActiveTimeout(String name);
// cancel and remove an active timeout, will not be executed
void cancelTimeout(String name);
```

Methods dealing with dialogue acts

```
// sending of dialogue acts
DialogueAct createEmitDA(DialogueAct da);
DialogueAct emitDA(int delay, DialogueAct da);
DialogueAct emitDA(DialogueAct da);
#DialogueAct String getDialogueActType();
#DialogueAct void setDialogueActType(String dat);
#DialogueAct String getProposition();
#DialogueAct void setProposition(String prop);
#DialogueAct boolean hasSlot(String key);
#DialogueAct String getValue(String key);
#DialogueAct void setValue(String key, String val);
#DialogueAct long getTimeStamp();

// Access to dialogue acts of the current session
// my last outgoing resp. the last incoming dialogue act
DialogueAct myLastDA();
DialogueAct lastDA();

// Did I say something like ta in this session (subsumption)? If so, how many
// utterances back was it? (otherwise, -1 is returned)
int saidInSession(DialogueAct da);
// like saidInSession, only for incoming dialogue acts
int receivedInSession(DialogueAct da);

// Check if we asked a question that is still pending
boolean waitingForResponse();
// Mark last incoming DA as treated and not pending anymore (stop rules firing)
void lastDAprocessed();

DialogueAct addLastDA(DialogueAct newDA);
#DialogueAct void setProposition(String prop);
```

Functions allowing lambda expressions (functional arguments)

```
boolean some(Collection<T> coll, Function<Boolean, T> pred);
boolean all(Collection<T> coll, Function<Boolean, T> pred);
List<T> filter(Collection<T> coll, Function<Boolean, T> pred);
List<T> sort(Collection<T> coll, Function<Integer, T, T> c);
Collection<T> map(Collection<S> coll, Function<T, S> f);
int count(Collection<T> coll, Function<Boolean, T> pred);
T first(Collection<T> coll, Function<Boolean, T> pred);
```

Methods on `Rdf` and `RdfClass` objects

```
Rdf toRdf(String uri);
#Rdf String getURI();
#Rdf boolean has(String predicate);
#Rdf long getLastChange(boolean asSubject, boolean asObject);
```

```
RdfClass getRdfClass(String s);
boolean exists(Object o);

// return only the name part of an URI (no namespace or angle brackets)
String getUriName(String uri);
```

3.5 Debugger/GUI

VOnDA comes with a GUI [Biwer, 2017] that helps navigating, compiling and editing the source files belonging to a project. It can also be attached to your VOnDA project at runtime to support debugging by logging the evaluation of rule conditions.

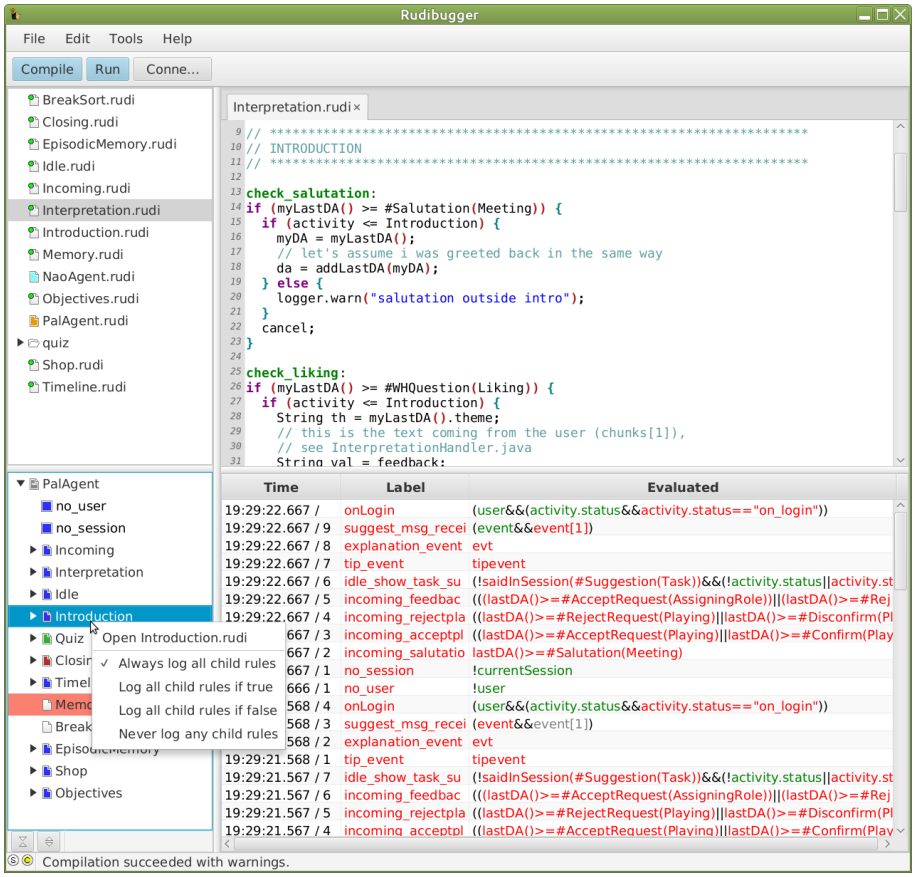


Figure 3.14: The VOnDA GUI window

For further details, please take a look into rudibuggers own documentation. The project can be found on `https://github.com/yoshegg/rudibugger`.

3.6 The RDF Database HFC

VOnDA follows the information state/update paradigm. The information state is realized by an RDF store and reasoner with special capabilities (HFC Krieger [2013]), namely the possibility to directly use n-tuples instead of triples. This allows to attach temporal information to every data chunk Krieger [2012, 2014]. In this way, the RDF store can represent *dynamic objects*, using either *transaction time* or *valid time* attachments, and as a side effect obtain a complete history of all changes. HFC is very efficient in terms of processing speed and memory footprint, and has also provides some stream reasoning facilities. VOnDA can use HFC either directly as a library or as a remote server.

The following is the syntax of HFC queries (EBNF):

```
<query>     ::= <select> <where> [<filter>] [<aggregate>] | ASK <groundtuple>
<select>    ::= {"SELECT" | "SELECTALL"} ["DISTINCT"] {"*" | <var>^+}
<var>       ::= "?"{a-zA-Z0-9}^+ | "?_"
<nwchar>    ::= any NON-whitespace character
<where>     ::= "WHERE" <tuple> {"&" <tuple>}^*
<tuple>     ::= <literal>^+
<gtuple>    ::= <constant>^+
<literal>   ::= <var> | <constant>
<constant>  ::= <uri> | <atom>
<uri>       ::= "<" <nwchar>^+ ">"
<atom>      ::= "\""  <char>^* "\"" [ "@" <langtag> | "^^" <xsdtype> ]
<char>      ::= any character, incl. whitespaces, numbers, even '\"'
<langtag>   ::= "de" | "en" | ...
<xsdtype>   ::= "<xsd:int>" | "<xsd:long>" | "<xsd:float>" | "<xsd:double>" |
                "<xsd:dateTime>" | "<xsd:string>" | "<xsd:boolean>" | "<xsd:date>" |
                "<xsd:gYear>" | "<xsd:gMonthDay>" | "<xsd:gDay>" | "<xsd:gMonth>" |
                "<xsd:gYearMonth>" | "<xsd:duration>" | "<xsd:anyURI>" | ...
<filter>    ::= "FILTER" <constr> {"&" <constr>}^*
<constr>    ::= <ineq> | <predcall>
<ineq>      ::= <var> "!=" <literal>
<predcall>  ::= <predicate> <literal>^*
<predicate> ::= <nwchar>^+
<aggregate> ::= "AGGREGATE" <funcall> {"&" <funcall>}^*
<funcall>   ::= <var>^+ "=" <function> <literal>^*
<function>  ::= <nwchar>^+
```

Table 3.1: BNF of the database query language

The reserved symbols `ASK`, `SELECT`, `SELECTALL`, `DISTINCT`, `WHERE`, `FILTER` and `AGGREGATE` do *not* need to be written in uppercase, but neither `filter` predicates nor `aggregate` functions should be named like reserved symbols.

don't-care variables should be marked *explicitely* by using `?_`, particularly if `SELECT` is used with `*` as in:

```
SELECT DISTINCT * WHERE ?s <rdf:type> ?_
SELECT * WHERE ?s <rdf:type> ?o ?_
```

To put a restriction on the object position you can also use *don't-care* variables and filters:

```
SELECT ?s WHERE ?s <rdf:type> ?o ?_ FILTER ?o != <foo-class>
```

Aggregates in HFC take whole tables or parts of them and calculate a result based on their entities. As the type of aggregates and filter functions cannot be overloaded, there are multiple similar functions for different types, e.g. F for `float`, L for `long`, D for `double`, I for `int`, and S for `String`.

Apart from `==` and `!=`, functional operators can be used in `filter` expressions as well. As for aggregates, there are multiple versions of the same function for different data types.

CountDistinct	FSum	LMax
Count	FMean	LMean
DMean	LGetFirst2	LMin
DSum	LGetLatest2	LSum
DTMax	LGetLatest	LGetLatestValues
DTMin	LGetTimestamped2	Identity

Table 3.2: Available aggregates

CardinalityNotEqual	FNotEqual	IntStringToBoolean	LMin
Concatenate	FProduct	IProduct	LNotEqual
DTIntersectionNotEmpty	FQuotient	IQuotient	LProduct
DTLessEqual	FSum	IsAtom	LQuotient
DTLess	GetDateTime	IsBlankNode	LSum
DTMax2	GetLongTime	IsNotSubtypeOf	LValidInBetween
DTMin2	HasLanguageTag	ISum	MakeBlankNode
EquivalentClassAction	IDecrement	IsUri	MakeUri
EquivalentClassTest	IDifference	LDecrement	NoSubClassOf
EquivalentPropertyAction	IEqual	LDifference	NoValue
EquivalentPropertyTest	IGreaterEqual	LEqual	PrintContent
FDecrement	IGreater	LGreaterEqual	PrintFalse
FDifference	IIncrement	LGreater	PrintSize
FEqual	IIntersectionNotEmpty	LIncrement	PrintTrue
FGreaterEqual	ILessEqual	LIntersectionNotEmpty	SameAsAction
FGreater	ILess	LIsValid	SameAsTest
FIncrement	IMax2	LLessEqual	SContains.java
FLessEqual	IMax	LLess	UDTLess
FLess	IMin2	LMax2	
FMax	IMin	LMax	
FMin	INotEqual	LMin2	

Table 3.3: Available filter functions

3.6.1 Usage of HFC in VOnDA

The RDF store contains the dynamic and the terminological knowledge: specifications for the data objects and their properties, as well as a hierarchy of dialogue acts, semantic frames and their arguments. These specifications are also used by the compiler to infer the types for property values (see sections 3.3.7 and 3.3.5), and form a declarative API to connect new components, e.g., for sensor or application data.

The ontology contains the definitions of dialogue acts, semantic frames, class and property specifications for the data objects of the application, and other assertional knowledge, such as specifications for "forgetting", which could be modeled in an orthogonal class hierarchy, and supported by custom deletion rules in the reasoner.

For queries which are too complex to be handled the VOnDA way, or if you want to do reasoning which for efficiency reasons should be handled by HFC rather than Java (e.g., if you are filtering for specific property values in a pool of many instances of the same class), there also is a direct communication port to HFC.

```
List<String> uris = new ArrayList<>();
// the ancestor is that hyponym which has the shortest path to syn
String ancestors = "select ?s  where ?s <wn20schema:hyponym> {} ?_ ";
QueryResult qr = proxy.selectQuery(ancestors, syn);
uris = RdfProxy.getValues(qr);
```

The above code, for example, retrieves all hyponyms of a given synset `syn`.

Currently, it is recommended to place such code in Java methods that you can then use in your VOnDA code to indirectly perform the queries. In the future, functionality will be added to support facilitated query construction directly in VOnDA code.

3.7 Extensions to the SRGS/VoiceXML Formalism

VOnDA comes with a simple NLU module that is based on the SRGS[2] grammar format. SRGS grammars are so-called augmented context free grammars, where the rules contain terminal and non-terminal symbols that define the set of strings that are valid matches of the *language* that the grammar defines, but also semantic actions that collect the output of the syntactic analysis in form of possibly complex objects. Our implementaton of the SRGS formalism[3] provides some extensions to the default grammar formalism, currently only in the semantics (action part) of the rules, providing return values not only based on the names of non-terminal symbols, but also using relative positions of matched strings or tokens

- `$%n` refers to the value returned by the rule n positions before this tag
- `$$n` refers to the the matched string n positions before this tag

Note that if `$%n` or `$$n` refers to an alternative, the result of the matched alternative is returned.

`n` always has to be greater than zero, since zero would be at the position where the semantic element is that makes the tag reference, which does not make sense. For determining `n`, you have to count *every* grammar token, including semantic tokens.

As an example, taken from the test cases of the `srgsparser` module, we parse the sentence "I want a medium pizza with Mushrooms please" with the grammar shown below, which will return a JSON object of the form:

```
{ "order": { "size": "medium", "topping": "mushrooms" } }
```

since the `$%2` in the `size` rule returned the output of the `$small | $medium | $large` alternative, and the `$$1` returned the string matched in the `medium` rule immediately before the tag. The rest is done with the ordinary JSON semantics present in the standard formalism.

```
#ABNF 1.0 UTF-8;

language en-EN;
root $order;
mode voice;
tag-format "semantics/1.0";

$politeness1 = [I want];
$politeness2 = [please];

$small = small {out = "$$1";};
$medium = medium {out = "$$1";};
$large = large {out = "big";};

$size = (($small | $medium | $large) pizza {out = $%2;})
        | (hot chili) { out="chili"; } ;

$topping = Salami {out = "salami";}
         | Ham {out = "ham";}
         | Mushrooms {out = "mushrooms";} ;

public $order =
  {out = new Object(); out.order = new Object;}
  $politeness1
  (
  [a] $size pizza {out.order.size = rules.size;}
  | [a] [pizza with] $topping {out.order.topping = rules.topping;}
  | [a] $size {out.order.size = rules.size;}
    with $topping {out.order.topping = rules.topping;}
```

[2] `https://www.w3.org/TR/semantic-interpretation/`

[3] `https://github.com/bkiefer/srgs2xml`, for more details see the ReadMe and documentation there

```
)
$politeness2 ;
```

Chapter 4

Using NLP Modules

Here¸ we describe the integration of the built-in NLU and NLG modules, but also the sketch how th integration of other ASR, NLU or NLG components can be done.

4.1 Configuration

The global configuration file can also contain the configurations for sub-modules of your system, which need not be part of the core framework. In the following, we describe how the SRGS parser and cplannner are integrated as an example of the plug-in infrastructure for NLP modules.

In the configuration file, there are tho sections, `NLU` and `NLG`, whose values are passed to the `LanguageServices` class, a factory to generate language interpreter and generator classes. The `LanguageServices` class and supporting interfaces and abstract classes, together with the configuration, form a flexible plug-in framework that also allows to provide your own processing modules.

In these NLU and NLG sections, there are sections for all supported languages, which is obligatory for `LanguageServices`; even if your system only supports one language, you still have to provide the language key.

The `LanguageServices` factory uses the language and the `class` attribute (mandatory) in the language section to create a `NLProcessor` object for the given task. The rest of the options can be used in the `init` method of the concrete processor to set it up accordingly. As was already said in section 2.4, non-VOnDA options are allowed in the config file (no error will result from adding new options at any level) and can be used by the application, as needed.

The framework has two basic built-in implementations, based on our extended SRGS implementation and the cplanner graph rewriting framework, respectively. In addition, as support for the SRGS NLU, there is a basic tokenizer implementation, which is configured using th `tokenizer` section inside the NLU section, in the same way as the NLU and NLG services.

4.2 NLU

The current built-in NLU is based on our extension of the SRGS formalism, and uses the following configuration keys: `grammar`, `converter` and `tokenizer`, which contains a whole subsection to get a `Tokenizer` object from the `NLProcessor` factory, which will be described later.

`grammar` points to the root SRGS grammar file in the file system, relative to the config file's position, and `converter` to a cplanner config file that defines a translation the SRGS NLU output in a declarative way into the internal form (of dialogue acts). The second part is optional, it is also possible to hard-code that, but this makes it less flexible, and the current mechanism requires almost no coding at all (except in cplanner rules).

All NLU classes have to extend the abstract `Interpreter` class. You have to implement the `DialogueAct analyse(String text);` method, and you may want to supersede the `init` method. The class already provides a several convenience functions, e.g., for converting JSON into internal datastructures, which can then be massaged into the right form using cplanner.

4.2.1 Configuration keys for `TrivialTokenizer`

The `TrivialTokenizer` splits the input string using white space as separators. This is currently the only tokenizer that is available by default and has the following configuration parameters:

- `toLower` (`boolean`): Convert the string to lower case before handing it on
- `removePunctuation` (`boolean`): Remove punctuation strings. What is considered to be punctuation is specified in `punctuationRegex`
- `punctuationRegex` (`String`): Must be a valid Java regular expression. All matches are removed from the input string.

4.2.2 Conversion of results using CPlan

Most NLUs will have fixed formats for the output they generate, which might not have the desired form or names for keys, etc. This kind of conversions can be hard-coded, which then results in bigger changes every time the NLU structures change, or done in a more declarative form using cplanner[1].

While converting list-valued structures is more complex, the usual moving around and renaming is quite easy, as you can see in the ChatCat example, which was described in the beginning.

4.3 NLG

Text generation currently is implemented using the cplanner graph rewriting framework (see above), which is a bit of an overkill for small projects. At time of creation, it was meant to shape the dialogue acts in a way to create valid input structures for an OpenCCG generator, which is a technology seldomly used today.

In addition, cplanner provides a quite powerful canned-text generation, with variables and, e.g., the possibility to compactly describe morphological variants. Whie it is a quite powerful and maybe not mastered easily, for larger projects there might be a benefit to simple dialogue act to string mapping with a hash map at some point.

To implement your own generator, you have to extend the abstract `Generator` class, which contains the abstract method

```
Pair<String, String> generate(DialogueAct da)
```

The first element of the pair is the text to print or send to TTS, while the second is meant to be a string representation of a robot or avatar movement to be executed in parallel to the (spoken) text. You will probably also have to override the `init` method to properly set up your component.

[1] For a complete documentation, check out the gui/doc directory of `https://github.com/bkiefer/cplan`

Chapter 5

Building VOnDA Agents

5.1 Implementation Patterns and Caveats

5.1.1 Proper Usage of `lastDAprocessed` and `emitDA`

`lastDAprocessed()` is a built-in method that helps you clean up after a dialogue act has been dealt with. You usually want to call it in your `propose` block, because when the block is executed that means that the dialogue act has been processed. The method's effect is to set an internal timestamp at the moment it has been called, which affects the return value of `lastDA()`: `lastDA()` will only return a dialogue act if it has been sent after the the point in time specified by the `lastDAprocessed` timestamp.

Be aware this also means that if the statement you execute in your `propose` block is `lastDAprocessed();`, all following calls to `lastDA()` will evaluate to an empty dialogue act. Thus, using expressions like `theme=lastDA().theme` in an `emitDA` are strongly discouraged, because they will fail if the `emitDA` is used after calling the cleanup method. There is, however, good reason to not move the `lastDAprocessed()` to the very end of your proposal, as proposals are executed in a separate thread and your VOnDA rules are (re-)evaluated in parallel. This might, in rare cases where your proposal code takes more time to process (for one possible reason, see 5.1.2), lead to your system generating and executing new proposals based on the "old" dialogue act, thus responding more than once to one input.

5.1.2 A Few Words About `emitDA` and `createBehaviour`

There is a feature to the `emitDA` method which has not been mentioned in section 3.3.8, but might become important for synchronisation. Usually, the client communication runs in a permanent loop in it's own thread, which then start the rule evaluation.

`emitDA` actually only is a wrapper method which uses the given dialogue act to create a behaviour, which is the actual thing being sent to the communication hub. `createBehaviour` wants to be passed a delay parameter, which specifies the amount of time the client communication thread should be paused after emitting the given behaviour. This might be important to your application if you use for example TTS and want to delay the next utterance after playing the current one has been finished. Normal `emitDA` sets the delay to `Behaviour.DEFAULT_DELAY`, which by default is zero, but you can also call `emitDA(delay, dialogueAct)` to directly specify a delay, or even override `createBehaviour` to perform a more complex computation of the delay time, e.g. to adopt to text lenght * speed of your TTS voice.

Attention! Once you are doing this, make sure that you use `lastDAprocessed()` early in your `propose` block as suggested in 5.1.1. If you don't and the thread the proposal is executed in is delayed long enough, new proposals will be generated based on the old dialogue act and your agent might end up saying things multiple times.

5.1.3 Waiting for a User's Answer in a Conversation

It's not very polite to be talking all the time without letting the interlocutor say something themselves. Particularly, you'll want to make sure that once the system asked a question, it will at least wait for some time

before going on, to give the user a chance to answer. To this end, you can use the pre-built `waitingForResponse` method, which returns true if the system was the last one to speak and the dialogue act it uttered was a question or a request.

5.1.4 Volatile variables in rule files and how to keep information between evaluation cycles

If the compiler was used in the default mode, during run-time only one object of the top-level generated class permanently exists. For all `included` rule files, only temporary objects are created, which live long enough to do all relevant rule evaluations. That means that they have access to all values of the top-level rule file and other rule files above them that have been created up to the point of the rule file inclusion.

This means that whenever the VOnDA rules are executed in a new cycle, they are executed in a "clean state" where all variables you previously set in the rule file have been reset. The only exception from this are variables which are either located in your custom (Java) Agent class or in your top-level rule file.

Thus: Always keep in mind that only variables defined in the Java Agent instance or in the top-level rule file are persistent, everything else is volatile! If you want to keep values between executions, store them in a top-level variable or in the database.

Alternatively, you can instruct the compiler to generate permanent embedded objects for rule execution and thus permanent variables using the `persistentVars` configuration flag. Be aware that if you need a defined state at every rule execution, you have to reset the corresponding variables yourself.

5.2 Troubleshooting: Typical Problems

- **The execution of my `propose` or `timeout` block does not have the effect I expected**
  Are you using any variables inside that block whose contents are changed by other parts of your code after the block has been issued?
- **The fields of the dialogue act sent by `emitDA` in my Proposal do not contain the values they should according to my conditions**
  Check whether you have "buffered" those values in final variables before issuing the Proposal and are referring to those. Using `lastDA` directly in the `propose` block is dangerous because it might already contain the next dialogue act (or none at all).
- **I get a NullPointerException in my Proposal**
  Please check whether all the variables you're using in the `propose` block are final and can't be changed by someone else between the time the Proposal is registered and the time it is executed. Also make sure you're not trying to read the fields of `lastDA()` after you called `lastDAprocessed`.
- **My system seems to execute the same Proposal multiple times**
  Make sure that in your `propose` block, you are calling `lastDAprocessed` as described in 5.1.1 and also "resetting" everything else that triggers the rule to be executed.
- **The variable that I use in my rules for storing information does not have the contents it is supposed to**
  Be aware that variables not defined in the top-level rule file are not persistent between rule evaluation cycles, thus you should not store information there which you want to keep (see section 5.1.4).
- **When compiling, I get the warning `"base name x can be one of y, z, please resolve manually"`**
  It seems your ontology is redefining an existing class. Refer to section 2.4.1 to see how to resolve the problem.

Chapter 6

VOnDA Syntax Overview

```
//
import java.nio.file.*;
import static java.nio.file.Files.*;
/*@
// A "Java" comment, everything in this comment will be transferred literally
// to the compiled file, without the comment brackets.
public static int MAX_INT = 100;
@*/

// ************************************************************************
// Tests and Comparisons
// ************************************************************************

// declare a Java method
#Object boolean equals(Object);

boolean funcWithJava(String h) {
  boolean ex = false;
  // inject plain java
  /* TODO: CURRENTLY DOES NOT WORK CORRECTLY
    ex = Files.exists(new File(h).toPath());
    */
  return ex;
}

// Define a variable of an RDF type
Child child;

// test for value under path, intermediate tests are generated
 if (child.hasFather.surname) System.out.println("foo");

// subsumption of dialogue acts, (equality is also possible)
if (lastDA() <= #AcceptOffer(Bringing)) System.out.println("foo");

Child c1;
Agent c2;
// Test if two RDF objects are the same object.
// Mark the difference to the followin example! Here, the objects are compared,
// (same for !=), but all other comparison operators compare the classes!
if (c1 == c2) System.out.println("foo");
```

```
// Is the class of c1 a subclass, resp. a strict subclass of the class of c2 ?
if (c1 <= c2) System.out.println("foo");
if (c1 < c2) System.out.println("foo");

// Is the class of c1 a subclass of the class 'Child'?
if (c1 <= Child) System.out.println("foo");

// Set the child's name, then clear it again
c1.name = "new name";
c1.name = null;

// Is s not null and not empty?
String s = "bar";
if (s) System.out.println("foo");

// Is s.compareTo(t) < 0
s = "bar"; t = "baz";
if (s < t) System.out.println("foo");

// Is i not null and not zero? (Same for all other Number containers)
Integer i = -1;
if (i) System.out.println("foo");

// Is o not null?
Object o = new Object();
if (o) System.out.println("foo");

// ************************************************************************
// Forward declaration of methods
// ************************************************************************

// This is the forward declaration
String foo(String a, String b);

// A rule using foo
a_rule:
if (foo("a", "b")) {
  // this java code should appear in the body of the rule
  /*@ String[] lbls = { "a", "b" }; @*/
  System.out.println("yes!");
}

// Function foo defined here
String foo (String a, String b) { return a + b; }

// ************************************************************************
// Breaking out of rules
// ************************************************************************

// break can be used to exit the rule specified, rules following this
// rule will still be executed
test_rule:
  if (true) {
    second_rule:
```

```
      if (false) {
        break test_rule;
      }
  }

// return from a file (not looking at subsequent rules) with cancel
test_rule2:
  if (true) {
    second_rule:
      if (false) {
        cancel;
      }
  }

// Stop all rule execution (even at higher levels) with cancel_all
test_rule3:
  if (true) {
    second_rule:
      if (false) {
        cancel_all;
      }
  } else {
    System.out.println("what?");
  }

// ***************************************************************************
// Near-to-Java features
// ***************************************************************************

// you can declare a variable final if you need to use it in a propose or
// timeout block
test_rule4:
  if(true) {
    final c = "hi";
    final boolean b = false;
    propose("test") {
      if(c.equals("bye"))
        System.out.println("bye");
    }
  }

// ***************************************************************************
// Coverage - To be revised and sorted properly
// ***************************************************************************

// there can always be an 'else' case
if (true && !false) { /* do something */ } else { /* do something else */ }

// TODO: sum1 and sum2 should, but do not become a proper declaration here
// An unknown function is used. The system tries to infer the type of
// operator arguments as best as possible
somevar = getSomething();
sum1 = 3 + somevar;
sum2 = somevar + 3;
anothervar = getSomething();
```

```
someint = 1 < 3 ? anothervar : 3;

// You can use casts and conditional expressions just like in Java
Integer anotherint = (Integer) i;
anotherint = anotherint < 3 ? anotherint : 3;

// There are also while and for loops and switch-case blocks available
while (anotherint > 7000) { continue; }
for (abc = 1; abc < 10; abc++) { ++abc; }

ArrayList<String> iterlist;
// The following two lines are equivalent, the type of a is inferred
for (String a : iterlist) {}
for (a : iterlist) {}

String str;
switch (str) {
  case "a": break;
  case "b":
  default:
}

// It is possible to perform implicit casts on iterable elements in a for
// loop; there is, however, no type safety promise from our side!
ArrayList<Object> iterlist2;
for (Integer a : iterlist2) {}

// These are some examples for lambda expressions
List<String> newlist = {"a", "cd", "ab", "b"};
allas = filter(newlist, (element) -> element.contains("a"));
dosort = sort(newlist, (a, b) -> a.length() - b.length());

// Creation of dialogue acts. Elements that are Variables need to be
// enclosed in {}, all others will be converted to Strings (symbols are default)
robotname = "Robert";
dia = #Inform(Liking, agent=username, patient={robotname});

void callToSomeMethod() {
  // a stub for an external function
}

// You can set labelled timeouts that will execute the code given to them
// once the time (given in milliseconds) is over
timeout("do_something", 15000) {
  callToSomeMethod();
}

// You can also condition a timeout on the finishing of a behaviour. The code
// block will be executed when either the specified time runs out or the
// behaviour finishes
timeout_behaviour(15000, #Inform(Call, when=soon)) {
  callToSomeMethod();
}

// shortcut notation for adding to and removing from collections
```

```
Set set = { 1, 2, 3 };
set += 4;
set -= 1;

// work for numbers the usual way
i = 9;
i += 1;
i -= 1;

// Automatic conversion to string of POD types
s = "number" + 10;
s = 10.0 + "number";
// and of RDF (returns URI)
s = "nameOf_" + child;
// Also in assignments
String number = 10;
// And dialogue acts
DialogueAct da1 = #Confirm(Correct, number={10});

// you can call any function on the top level
t = "10";
i = toInt(t);

// ************************************************************************
// Access to Dialogue Acts
// ************************************************************************
DialogueAct da = #Inform(Posessing);
if (! da.theme) {
  // set the value of any argument
  da.theme = "Ball";
}
// does the DialogueAct has any arguments?
if (da.getValues()) {
  s = da.getValues().get(0);
}

dont_like_foot:
if (da.theme == "Football") {
  da.theme = "Basketball";
  final dialogueAct = "Inform";
  propose("new_liking"){
    emitDA(#{dialogueAct}(Liking, theme={da.theme}));
  }
}

System.out.println(da.theme);
da.setDialogueActType("Confirm");
da.setProposition("Acquiring");
//System.out.println(da);

// creation of Java objects works as usual, java.util.* is imported by default
List<String> li = new ArrayList(2);

// include a set of rules, can be put in a subdirectory
include sub.Sub;
```

```
Quiz p;
h = filter(p.hasHistory, (c) -> c.turnId == 1);
if (h) {
  x = h.get(0);
}

Map<String, String> m;
if (m.containsKey("foo")) {
  s = m.get("foo");
}

if (! m.containsKey("bar")) {
  s = m.put("bar", "foo");
}

// Check redefinition of local variables in lambda and for
z = filter(p.hasHistory, (h) -> h <= QuizHistory);
z1 = filter(p.hasHistory, (h) -> ((QuizHistory)h) <= QuizHistory);
z2 = filter(p.hasHistory, (QuizHistory h) -> h <= QuizHistory);
z3 = filter(p.hasHistory, (h) -> h <= QuizHistory);

for (int h = 0; h < 3; ++h) { System.out.println(Integer.toString(h)); }

int func(String h) {
  h = "foo";
  return h.length();
}

{
  int h = 1;
  ++h;
}

// pre- and post-increment and decrement work for all functional properties
// with numerical range;
++child.weight;
d = child.weight++;
d = --child.weight;
child.weight--;

// and this is it!

//
```

Chapter 7

Changes to VOnDA Version 2

Syntax changes (breaking changes)

- the `import` keyword was replaced by `include`. `import` is now used exclusively like in Java, as class import definitions which are only allowed at the beginning of a rule file
- the syntax of the type declaration for external classes has changed from
  `[SomeClass]. myType myMethod(argTypeA, argTypeB);`
  to
  `#SomeClass myType myMethod(argTypeA, argTypeB);`
- `{<exp>}` expansion syntax in field access
  In version 2, if one of the identifiers in a field access chain (like `child.name`) matched an existing variable of type string, the identifier was replaced with the value of the variable. This sometimes produced confusing results and unforeseen errors, and was too hard to grasp for novel users. Therefore, this has been replaced by the more visible approach also used for value expansion in the dialogue act specifications (see section 3.3.5).
- developing an unambiguous grammar for the java cast syntax `((Type)val)` is very hard. The current parser generator (bison) does not provide the functionality to get this right in all cases. Therefore, it has been replaced by an infix operato `isa`, so `((Type)val)` becomes `isa(Type, val)`.
- the same holds for Java lambda expressions (`c -> <expression>`). They were also replaced by infix syntax: `lambda(c) <expression>` resp. `lambda(c) { <statements>; }`

Configuration changes (breaking changes)

- the formerly required extension of the run-time library `Agent` class is now optional. You can still do this, and specify the fully qualified name in the config file under `agentBase`, or the compiler will make the top-level rule class a subclass of `de.dfki.mlt.rudimant.agent.Agent`.
- If you want to provide a file for type definitions to help the compiler, you have to specify it explicitely in the config under the key `typeDef`. In version 2, it was named after the now optional wrapper class, now it can take any name and is also optional
- NLG and NLU are now treated alike in the plugin factory, which means that NLG needs a `class:` key also for the default generator

Additional built-in functionality

- The NLP components have been restructured, now also support for tokenizer functionality to support NLU it built in

Bibliography

Dave Beckett. Raptor RDF Syntax Library. http://librdf.org/raptor/, März 2017. URL `http://librdf.org/raptor/`. Zuletzt überprüft: 11.10.2017.

Christophe Biwer. rudibugger - Graphisches Debugging der Dialogmanagementtechnolgie VOnDA. Bachelor's Thesis, Saarland University, 2017.

Harry Bunt, Jan Alexandersson, Jae-Woong Choe, Alex Chengyu Fang, Koiti Hasida, Volha Petukhova, Andrei Popescu-Belis, and David R Traum. ISO 24617-2: A semantically-based standard for dialogue annotation. In *LREC*, pages 430–437. Citeseer, 2012.

Bernd Kiefer, Anna Welker, and Christophe Biwer. VOnDA: A Framework for Ontology-Based Dialogue Management. In *International Workshop on Spoken Dialogue Systems Technology (IWSDS)*, page 12. Springer, 2019. URL `https://arxiv.org/abs/1910.00340`.

Hans-Ulrich Krieger. A temporal extension of the Hayes/ter Horst entailment rules and an alternative to W3C's n-ary relations. In *Proceedings of the 7th International Conference on Formal Ontology in Information Systems (FOIS)*, pages 323–336, 2012.

Hans-Ulrich Krieger. An efficient implementation of equivalence relations in OWL via rule and query rewriting. In *Semantic Computing (ICSC), 2013 IEEE Seventh International Conference on*, pages 260–263. IEEE, 2013.

Hans-Ulrich Krieger. A detailed comparison of seven approaches for the annotation of time-dependent factual knowledge in rdf and owl. In *Proceedings 10th Joint ISO-ACL SIGSEM Workshop on Interoperable Semantic Annotation*, 2014.

Josef Ruppenhofer, Michael Ellsworth, Miriam RL Petruck, Christopher R Johnson, and Jan Scheffczyk. *FrameNet II: Extended theory and practice*. Institut für Deutsche Sprache, Bibliothek, 2016.

Stanford Research. Protege.stanford.eu. http://protege.stanford.edu, 2017. Zuletzt überprüft: 11.10.2017.