VOnDA, A Framework for Dialogue Management

Bernd Kiefer, Anna Welker

July 5, 2019

Contents

Chapter 1

Purpose

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 RDF store and reasoner of choice used in VOnDA is HFC (Krieger, 2013). For further details about the general funcitonalities of HFC see chapter 3.4; for an example of HFC as a database in a VOnDA project please refer to section 2.1.

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, that supports implementing dialogue management modules using the compiled rules.

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

1.1 Internal Structure

Figure 1.1 shows the main components of a VOnDA agent.

At the base there is an RDF store, which takes incoming sensor and interaction data, and stores it as RDF data, based on 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 dialogue manager will get several input types from the nexus, the ones currently foreseen are: input from automatic speech recognition (ASR) or typed natural 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. The rules can not only use new data, but also the interaction history stored in the RDF database to take their decisions. 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. The selection 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 sensory data such as her or his estimated mood.

The last two parts are a 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 turns outgoing dialogue acts into natural language utterances and gestures.

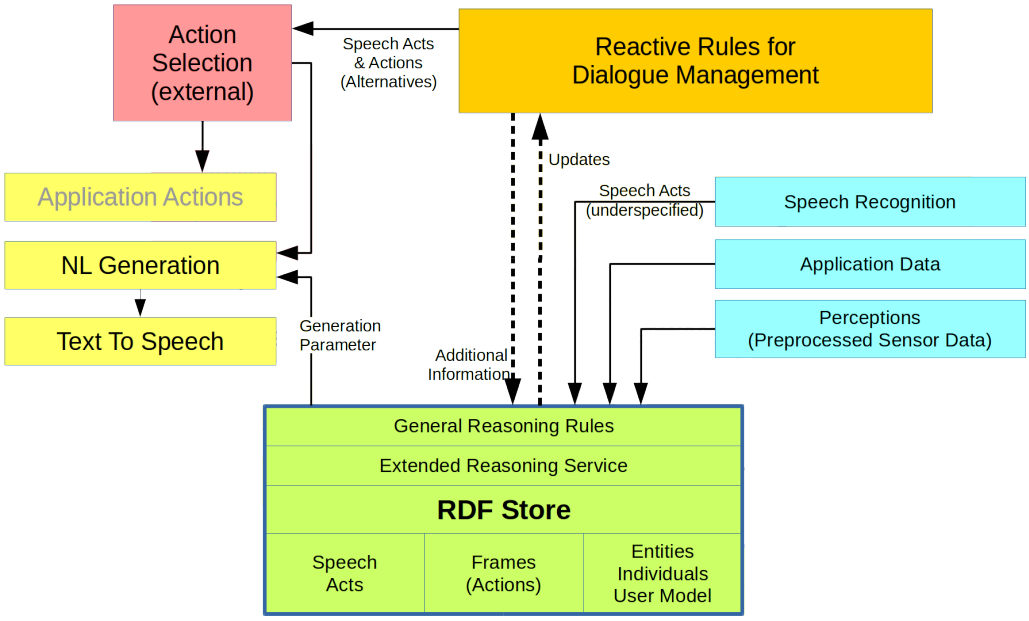


Figure 1.1: Schematic VOnDA agent

Chapter 2

Sketching a Simple Interaction Manager

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, the 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 their name and say goodbye.

To see the example system that is constructed in this chapter, switch to the folder named examples. It contains the play-ground dialogue system ChatCat. More complex examples are planned to be published soon.

2.1 Setting up the RDF Store of your Project

A dialogue system aiming for social interaction does need some kind of memory representation. Therefore, the first step of building your dialogue manager should be to set up the ontology. It can of course be extended on-the-fly, while your system is becoming more and more elaborate, but you should still create a base from which to start from.

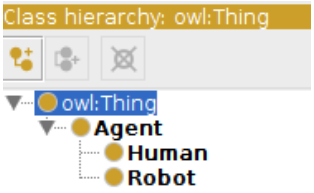
At the very least, you are strongly recommended to use the dialogue hierarchy (provided in `examples/chatcat/src/main/resources/ontology/dialogue.nt`) as well as two default information files pertaining to the dialogue acts used in the VOnDA framework.

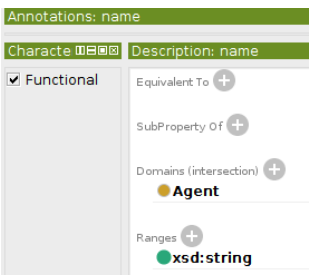
2.1.1 Creating n-Tuple Files

HFC loads data from n-Tuple files. You can however create files in the more common OWL format and then automatically transfer it with a simple shell script, using the Raptor (Beckett, 2017) (e.g. `examples/chatcat/ntcreate.sh`). This tutorial uses Protégé (Stanford Research, 2017).

Open your favourite IDE and create a new file that includes an RDF class Agent, and two subclasses of this class, Robot and Human.

After that, create a predicate `name` for the class Agent that has its range in `xsd:string`.





As we do not know the user a priori, we will make the system create an instance for them at run-time. However, we know our robot, so we will create an instance of the class Robot and name it *Robert*.

Now save your new ontology. Then you can use Raptor (i.e., the `ntcreate.sh` script) to transform it into n-tuple format. Afterward, create your `.ini`-file as described in the next section and add your tuple file to it.

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

2.1.2 Combining multiple ontology files - How to Use the .ini

An HFC ontology is held together - and addressed to - via the ini file. This is a plain text file with the ending `.ini` that contains various settings and information. To create your own `.ini`, best copy the example below and add your ontology information to it as described in the following paragraphs.

```
[Settings]
#minNoArgs=3
#maxNoArgs=4
#noAtoms=100000
#noTuples=500000
PersistencyFile=tuples.nt
Encoding=UTF-8

[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#

# instead, you can also load one or more namespace files
#default.ns

[Tuples]
# the axiomatic triples for OWL-Horst w/ EQ reduction
default.eqred.nt

[Tuples]
dialogue.nt
chatcat.nt

[Rules]
# we need special rules for transaction time (mixture of triples/quadruples)
default.eqred.quads.rdl
```

Figure 2.1: An examplary `.ini` file

[**Settings**] For most applications concerning dialogue management, it will be important to specify a `PersistencyFile`. You can put it anywhere you like (i.e., it will be created automatically in the place you specify), but 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. So, if you want to find out which specific tuples have just been entered, you can refer to this file. At the same time, if you want to wipe the memory of your system, delete this file.

[**Namespaces**] In this section, you can specify abbreviations for your ontology namespaces. With the definition of the variable `dial` in figure 2.1, it is made possible to refer to classes, predicates and instances of this namespace with e.g. `<dial:Accept>` instead of `<http://www.dfki.de/lt/onto/common/dialogue.owl#Accept>` in your queries to the database.

As you can see, we included a shortcut for our chatcat ontology here.

[**Tuples**] Here you can put all the ontology files that you created. You should also put the file `dialogue.nt` which, as previously mentioned, contains the specifications of the dialogue acts used by the VOnDA framework.

[**Rules**] This specifies the set of rules that HFC uses to build the ontology. It is recommended to simply use the file `default.eqred.quads.rdl` here. For further information, please refer to the documentation of HFC.

2.2 The Framework: Setting up the Base of your Dialogue System

You need to implement your own abstract (Java) agent as a subclass of `de.dfki.mlt.rudimant.agent.Agent`. Furthermore, you will need an implementation of `de.dfki.mlt.rudimant.agent.CommunicationHub`. To see an example of how this should be done, take a look in the source folder of ChatCat.

The two most important things here are that you make sure that the database is initialized (as an instance of RdfProxy) and that you do use an instance of your VOnDA Agent implementation in your client. Of course this code will not compile until you build your first rule file, i.e., your VOnDA Agent.

After this, the next thing you need is to create some main method where an instance of your client is built and set running using the `startListening()` method.

As a start, we recommend you to use the ChatCat system as a base for your own system and extend it. This will also provide you with a GUI where you can test your first dialogue steps.

2.3 Connecting NLU and Generation Components

Basically, you can connect any NLU and NLG components to your project that are able to create or, respectively, process dialogue acts of the format that VOnDA provides (cfg. 3.1.2).

For the sake of simplicity, this example uses SRGS to build a primitive NLU and cplan to create natural language out of the dialogue acts the agent outputs.

2.4 Writing some First Rules

Now that everything else has been arranged, we are set up for writing our first dialogue management rules.

So, let's 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.xml), we defined that an utterance of the user like "Hello" will be parsed as the dialogue act Salutation, with the proposition Greet. We now can define a rule reacting to this utterance:

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

This will create a new instance of Human, which is stored in a global variable `user` that in our case has been defined in the ChatAgent and will be present during the whole conversation.

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();
  }
```

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

2.5 Specifying how to Compile and Run your Project

You can compile your VOnDA rule files by executing the command
`<yourVondaDirectory>/compile -c <yourCompileYml> <YourToplevelRudiFile>`.

The compile.yml should contain the following parameters:

`outputDirectory:`	Relative to the current location, where should the compiled classes go?
`wrapperClass:`	The name of your abstract Java Agent, fully qualified with package specifcation
`ontologyFile:`	The path to your .ini, relative to the current location
`rootPackage:`	The package where you want the compile results to be in
`failOnError:`	Set to true to exit compilation on any encountered type errors, otherwise set to false (*)

(*) Note that although VOnDA type checking is becoming more and more reliable, it is not complete. In some cases, setting this switch to true might make your project uncompilable although when compiling it when ignoring type errors would result in a perfectly sound Java project.

To provide the possibility to easily change the Ontology, NLU and NLG components of your project, it is recommended to create a second yml file containing the following specifications (This is an example from ChatCat):

```
ontologyFile:       src/main/resources/ontology/chatcat.ini
NLG:
eng:
mapperProject: src/main/resources/grammars/cplanner/allrules-mapper
generationProject: src/main/resources/grammars/cplanner/allrules
NLU:
eng:
class: de.dfki.chatcat.SrgsParser
grammar: src/main/resources/grammars/srgs/chatcat.xml
```

This configuration can then be easily read in when starting your project, and be passed to the init method of Agent, which requires such information.

2.5.1 Resolving Namespace Ambiguities

As you might have noticed whilst looking at chatcat.yml, there is another parameter used in the compile configuration of our example project:

```
nameToURI:
Agent : "<cat:Agent>"
```

When trying to compile without these two lines, you will find that VOnDA produces the warning "`base name Agent can be one of <http://www.semanticweb.org/anna/ontologies/2018/3/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. `nameToURI` can be used to do the latter.

Of course, you can also use this functionality to "rename" Rdf classes as you please: VOnDA will always map the name on the left to the class URI provided on the right.

2.6 Advanced Features

2.6.1 Connecting to a second HFC Server

In the standard setup, your VOnDA project uses one HFC server that on starting your system loads all the information from your ontology and that receives your new database entries and modifications.

However, there might be cases where this approach is not what you want. If your project uses a big database with static information, that you use but do not need to write to, you might prefer to not start the server anew each time you start your system, as this might consume time (e.g., loading ???? WordNet[1] triples takes ???minutes on a reasonably fast machine).

numbers on PAL computer

In this case, there is another solution: you can start your HFC server remotely and then connect to it in your VOnDA agent.

hfc server start script

This additional server does of course not have the same status as the innate HFC proxy, as you only connect to it at run-, not at compile-time. You can query it for information as described in 3.4.1, but classes from this database will not be recognized in the rudi code and you cannot write to it .

code example adding new client and proxy for use in java

or can you? Try?!

[1] https://wordnet.princeton.edu/

Chapter 3

Installing and Getting Started

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.

3.1.1 VOnDA's Architecture

Figure 3.1 shows the architecture of a runnable VOnDA project.

A VOnDA project consists of an ontology, a custom extension of the abstract `Agent` class (the so-called *wrapper class*), 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 `import` statements. The blue core in Figure 3.1 is the run-time system that is part of VOnDA, while all green elements are the application specific parts of the agent. A `Yaml` project file contains all necessary information for compilation: the ontology, the wrapper class, the top-level rule file and other parameters, like custom compile commands.

The VOnDA compiler translates rule files with the extension `.rudi` to Java files. 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 3.1.2).

Every rule file can define variables and functions in VOnDA syntax which are then available to all imported files.

The current structure assumes that most of the Java functionality that is used inside the rule files will be provided by `Agent` superclass. There are, however, alternative ways to use other Java classes directly.

The methods and fields from the custom wrapper class can be made available to all rule files by declaring them in the interface connecting the `.rudi` code to the Java framework. This interface must have the same name as the wrapper class but end with `.rudi` (in the example of figure 3.1, this would be `MyAgent.rudi`).

3.1.2 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 strageties. One of the most important features is the way objects in the RDF store can be used throughout the code: RDF

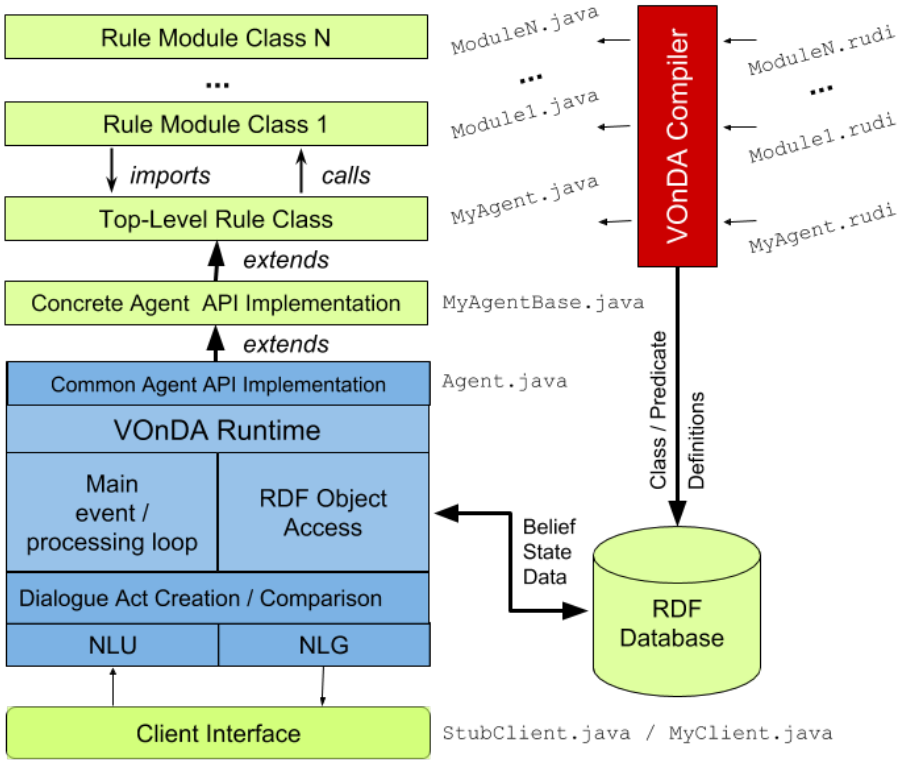


Figure 3.1: Gesamtarchitektur von Rudimant

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.

The structure of a VOnDA file

VOnDA does not demand to group statements in some kind of high-level structure like e.g. a class construct. In fact, it is currently not possible to define classes in `.rudi` files at all. Rules and method declarations can just be put into the plain file. The same holds true for every kind of valid (Java-) statement, like assignments, for loops etc. From this basis, the compiler will create a Java class where the methods and rules that are transformed to methods are represented as the methods of this specific class. All other statements will be put into the `process()` method that VOnDA creates to build a rule evaluation cycle. In doing so, the 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 all rules or to insert termination conditions that are not contained in rules.

Warning: It is important to know here that variables declared globally in a file will be transformed to fields of the Java class. We found that in very rare occasions, 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 immediately be executed, 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, what might lead to completely unexpected behaviour when proposes and timeouts with changed variable values are executed.

```
user = new Animate;
user.name = "Joe";
set_age:
if (user.age <= 0) {
  user.age = 15;
}
```

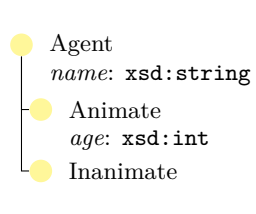


Figure 3.2: Ontology and VOnDA code

RDF accesses and functional vs. relational properties

Figure 3.2 shows an example of VOnDA code, and how it relates to RDF type and property specifications, schematically shown on the right. The domain and range definitions of properties are picked up by the compiler and 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 succeeeds, a test if the value is greater than zero. 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. 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.

New RDF objects can be created with `new`, similar to Java objects; they are immediately reflected in the database, as are all changes to already existing objects.

```
Child c;
String name = c.name;
c.name = "new name";
Set middle = c.middleNames;
c.middleNames += "John";
c.middleNames -= "James";
```

⇓

```
String name = (String)c.getValue("<upper:name>");
c.setValue("<upper:name>", "new name");
Set middle = (Set<Object>)c.getValue("<upper:middleNames");
c.add("<upper:middleNames", "John");
c.remove("<upper:middleNames", "James");
```

Table 3.1: Examples for an RDF property access

Due to the connection of VOnDA to HFC during compile time, it has full access to the database. Thereby, it cannot only recognize the correct RDF class to create a new instance from when encountering `new`, but it can also resolve property accessses to such instances. Field accesses as shown in line 2 and 3 of table 3.1 will be analyzed and transformed into database accessses. 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.

In this process, the compiler will automatically recognize the equivalences of the following RDF and Java types:

Moreover, VOnDA can see whether an access is done by functional or relational predicates and will handle it accordingly, deriving 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)`.

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


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

Figure 3.3: A simple rule

Rules and labels

The core of VOnDA dialogue management are the dialogue rules, which will be evaluated in the run-time system on every environment trigger.

A rule starts with its name, from now on called label, followed by a colon. Pertaining to this label is an if-statement, possibly with 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 rule cannot be executed, like stopping the evaluation of (a sub-tree of) the rules if necessary information is missing.

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 life system, as they can be logged live with the debugger gui (cfg. chapter 3.3). The debugger can show you which rules were executed when and what the individual results of each part of the conditions were.

Interrupting the rule evaluation cycle

There are multiple ways to stop the 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 imported rules). 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 not to evaluate whatever triggered the current evaluation cycle and should only be used as an 'emergency exit', as the dialogue rules should be rejecting any sound, non-matching trigger by themselves.

To leave `propose` and `timeout` blocks, you need to use an empty `return`, as they are only reduced representations of normal function bodies.

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

Figure 3.4: VOnDA code example

`propose` and `timeout`

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 which one will be taken and the closure is executed.

`timeout`s also generate closures, but with a different purpose. They can be used to trigger proactive behaviour, or to check the state of the system after some time period, or in regular intervals. A timeout will only be created if there is no active timeout with that name.

There are two variants of `timeout`: *labeled* timeouts, like the one in the previous example, run out after the specified time (unless they are cancelled before that using the `cancelTimeout(`*`label`*`)` function) 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, whatever comes first.

Dialogue acts

Without doubt, an important task of a dialogue system is emitting dialogue acts that can be transformed to natural language by the generation component to communicate with the user. In VOnDA, the function responsible for this is `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 3.5: VOnDA code example

The dialogue act passed to `emitDA` should look as shown in figure 3.5. `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 =) 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.

Type inference and overloaded operators

VOnDA allows static type assignments and casting, but neither of them is always necessary.

If, for example, the type of the expression on the right-hand side of declaration assignment is known or inferrable, it is not necessary to explicitely state it.

You can also declare variables final.

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

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

Table 3.2: Transformation of complex boolean expressions

A time-saving (and code-readability-improving) feature of VOnDA 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 boolean. 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.

Be aware that the expansion in the figure only occurs if the multiple field access is used as boolean test. In the example of 3.3, if the first clause in the boolean expression is omitted, a `NullPointerException` could still occur.

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

Table 3.3: Transformation of complex boolean expressions

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")) {
  ...
}
```

Table 3.4: Overloaded comparison operators

External methods and fields

As mentioned before, you can use every method or field that you declare in your custom Agent implementation in your VOnDA code. Their declaration in the Java-rudi interface should look like a normal Java field or method definition (cfg. figure 3.5). It is possible to use generics in these definitions, although they are, for complexity reasons, restricted to be one single uppercase letter.

There is a variety of standard Java methods called on Java classes that VOnDA automatically recognizes, like e.g. the substring method for Strings. If you find that you need VOnDA to know the return type of a new method that can or should only be called upon instances of a specific class, you can provide VOnDA with knowledge about them by adding their definition to the interface as follows:

`myType someVariable;`	there is a variable of type `myType`
`myType someFunction(typeA a, typeB b);`	the method someFunction takes arguments of types `typeA` and `typeB` and returns an object of type `myType` (void = void)

Table 3.5: Defintions of existing Java fields and methods for VOnDA

`[type]. myType Function(typeA a);`	declaration of a function that has to be called on an instance of class `type`.

Table 3.6: Definition of a non-static method of Java objects

Fields of Java classes that you want to access over the instances of such a class can be defined analogously: `[type]. myType someVar`

Of course you can also use generics in these definitions. For example, the get method on lists is defined as follows in the VOnDA framework:

```
[List<T>]. T get(int a);
```

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

Functional constructs

VOnDA allows for using lambda constructions. At the moment, their usage is limited to the implementation of `Predicate` or `Comparator` in the following functions that are pre-defined in the Agent framework.

For example, if you want to filter a set of RDF objects by a subtype relation, you can write:

```
des = filter(agent.desires, (d) -> ((Desire)d) <= UrgentDesire);
```

`import`

`import` is a keyword in the VOnDA language. A global line like "`import File;`", to be placed at an arbitrary positions between the rules, results in the inclusion of the file `File.rudi` at exactly this position.

This inclusion has two important effects. On one side, it triggers the compilation of the included file at exact this point, such that any fields and methods known at this time will be available in the important file and it does not need to be compiled seperately. On the other side, it has the effect that all the rules contained in the important file will be inserted in the rule cycle at the specific point of the `import`, where in the resulting code the `process()` method of the imported file will be executed.

So the `import` functionality make sit possible to distribute the rules of a project into multiple files, respectively modules, and pin them together in such a way that they are handled as one unit when compiling or executing the compilied code. This is not only useful for clarity and structuring of project, but also supports modularity, as different subtrees of the `import` hierarchy can easily be added, moved, taken away or re-used in different projects.

Java-Code verbatim in rule files

To maintain simplicitely, VOnDA intentionally only provides limited Java functionalities. Whatever is not feasible in `.rudi` code should be done in methods in the wrapper class (cfg. **??**).

In cases where this is not enough and you urgently need a functionality in a `.rudi` file that VOnDA cannot parse or represent correctly, you can use its verbatim funciton. Everything between `/*@` and `@*/` will be treated like a multi-line Java comment, meaning the content is not parsed

```
boolean contains(Collection coll, Predicate pred);
boolean all(Collection coll, Predicate pred);
List<Object> filter(Collection coll, Predicate pred);
List<Object> sort(Collection coll, Comparator c);
```

Table 3.7: Functions that take lambda expressions as an argument

or evaluated further. It will be transferred to the compiled code at exactly this position between statements, but without the comment symbols.

In particular, this functionality can be used to import Java classes at the beginning of a rule file. You should however be aware that VOnDA will not know these classes 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, be it for commodity or for other reasons, you want VOnDA to have type information about methods called on instances on one of these classes, you can put this information into the type interface of the wrapper class (cfg. chapter 3.1.2).

3.2 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. It also contains methods for the creation and modification of shallow semantic structures, and especially for searching the interaction history for specific utterances. Most of this functionality is available through the abstract `Agent` class, which has to be extended to a concrete class for each application.

There is functionality to talk directly to the HFC database using queries, in case the object view is not sufficient or to awkward. The natural language understanding and generation components can be exchanged by implementing existing interfaces, and the statistical component is connected by a message exchange protocol. A simple generation engine based on a graph rewriting module is already integrated, and is used in our current system as a template based generator. The example application also contains a VoiceXML based interpretation module.

A set of reactive rules is executed whenever there is a change in the information state (IS). These changes are caused by incoming sensor or application data, intents from the speech recognition, or expired timers. Rules are labeled if-then-else statements, with complex conditions and shortcut logic, as in Java or C. The compiler analyses the base terms and stores their values during processing for dynamic logging. A rule can have direct effects, like changes in the IS, or system calls. Furthermore, it can generate so-called *proposals*, which are (labeled) blocks of code in a frozen state that will not be immediately executed, similar to closures.

All rules are repeatedly applied until a fix point is reached: No new proposals are generated and there is no IS change in the last iteration. Then, the set of 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, taking into account arbitrary features from the data storage.

3.2.1 Functionalities (methods) Provided by the Run-Time System

The following methods are declared in `src/main/resources/Agent.rudi` and implemented in 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);

[List<T>]. T get(int a);
```

```
[Collection<T>]. void add(Object a);
[Collection<T>]. boolean contains(Object a);
[Collection<T>]. int size();
```

- Short-hand conversion methods from Agent

```
int toInt(String s);
float toFloat(String s);
double toDouble(String s);
boolean toBool(String s);
```

- Other Agent methods

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

String getLanguage();

// Math methods
int random(int limit);
float random();

// logging methods
Logger logger;

// 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 handling with dialogue acts

```
// sending of dialogue acts
DialogueAct createEmitDA(DialogueAct da);
DialogueAct emitDA(int delay, DialogueAct da);
DialogueAct emitDA(DialogueAct da);

// 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);

boolean waitingForResponse();
void lastDAprocessed();
```

```
DialogueAct addLastDA(DialogueAct newDA);
[DialogueAct]. void setProposition(String prop);
```

- Methods using lambda expressions

```
// lambda: first class argument of Function is return type of function object
boolean contains(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);
```

3.3 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.

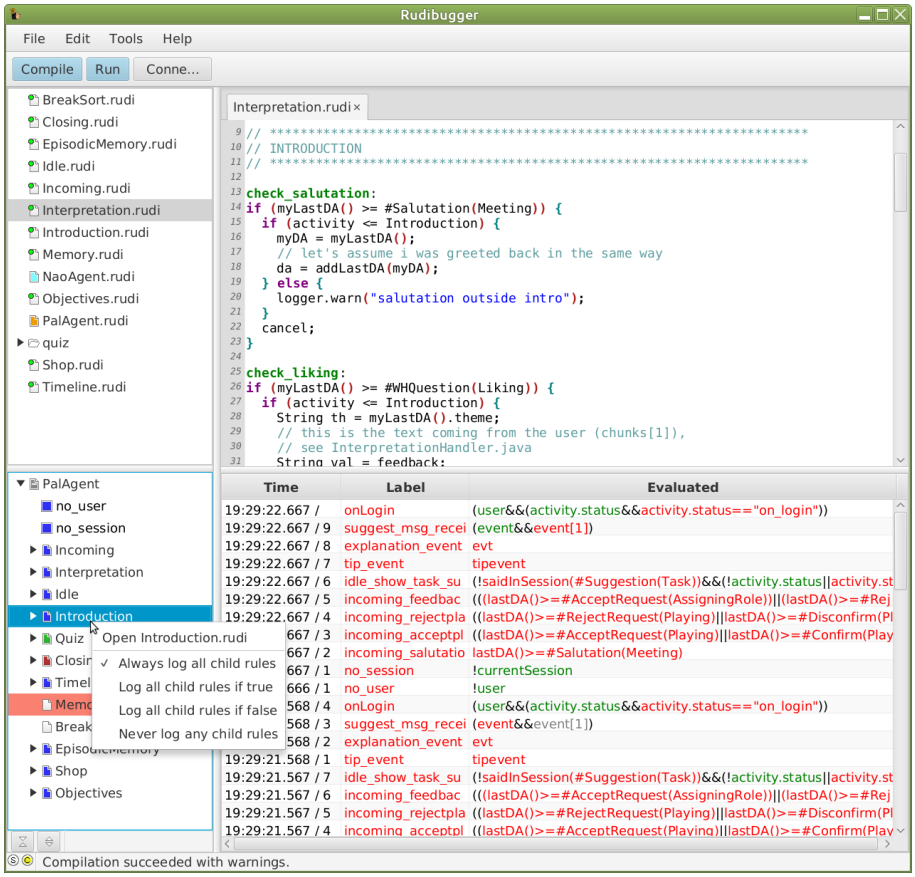


Figure 3.6: The VOnDA GUI window

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

3.4 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 recently be extended with stream reasoning facilities. VOnDA can use HFC either directly as a library, or as a remote server, also allowing for more than one instance, if needed.

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 3.8: BNF of the database query language

Notes The reserved symbols `ASK`, `SELECT`, `SELECTALL`, `DISTINCT`, `WHERE`, `FILTER` and `AGGREGATE` do *not* need to be written in uppercase.

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 change the object position without projecting it you can use *don't-care* variables:

```
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 `!=`, funcitonal operators can be used in `filter` expressions as well. As for aggregates, there are multiple versions of the same function for different datatypes.

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

Table 3.9: 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 3.10: Available filter functions

3.4.1 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 3.1.2 and 3.1.2), 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.

Add example of how to query proxy in Java code, and mention that we want to add functionality for easy querying in rudi code

Chapter 4

Building VOnDA Agents

4.1 Implementation Patterns

4.1.1 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 of 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 time of `lastDAprocessed` timestamp.

Actually, shouldn't we already explain lastDAprocessed somewhere else?

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 executed in parallel. This might, in rare cases where your proposal code takes more time to process (for one possible reason, see 4.1.2), lead to your system generating and executing new proposals based on the "old" dialogue act, thus responding more than once to one input.

4.1.2 What to Keep in Mind when Using `emitDA`

4.2 Caveats

Bibliography

Beckett, D. (2017). Raptor RDF Syntax Library. http://librdf.org/raptor/. Zuletzt überprüft: 11.10.2017.

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

Bunt, H., Alexandersson, J., Choe, J.-W., Fang, A. C., Hasida, K., Petukhova, V., Popescu-Belis, A., and Traum, D. R. (2012). ISO 24617-2: A semantically-based standard for dialogue annotation. In *LREC*, pages 430–437. Citeseer.

Krieger, H.-U. (2012). 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.

Krieger, H.-U. (2013). 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.

Krieger, H.-U. (2014). 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*.

Ruppenhofer, J., Ellsworth, M., Petruck, M. R., Johnson, C. R., and Scheffczyk, J. (2016). *FrameNet II: Extended theory and practice*. Institut für Deutsche Sprache, Bibliothek.

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