VOnDA, the DFKI MLT dialogue engine

Bernd Kiefer, Anna Welker

April 1, 2018

Contents

Chapter 1

Purpose

Natural language dialogue systems are becoming more and more popular, be it as virtual assistants such as Siri or Cortana, as Chat Bots on websites providing customer support, or as interface in human-robot interactions in areas ranging from Industry 4.0 (Schwartz et al., 2016) over social human-robot-interaction (ALIZ-E, 2010) to disaster response (Kruijff-Korbayová et al., 2015).

A central component of most systems is the *dialogue manager*, which controls the (possibly multi-modal) reactions based on sensory input and the current system state. The existing frameworks to implement dialogue management components roughly fall into two big groups, those that use symbolic information or automata to specify the dialogue flow (IrisTK (**?**), RavenClaw (Bohus and Rudnicky, 2009), Visual SceneMaker (Gebhard et al., 2012)), and those that mostly use statistical methods (PyDial Ultes et al. (2017), Alex (Jurčíček et al., 2014)). Somewhat in between these is OpenDial (Lison and Kennington, 2015), which builds on probabilistic rules and a Bayesian Network.

When building dialogue components for robotic systems or in-car assistants, the system needs to take into account *various* system inputs, first and foremost the user utterances, but also other sensoric input that may influence the dialogue, such as information from computer vision, gaze detection, or even body and environment sensors for cognitive load estimation.

The integration and handling of the different sources such that all data is easily accessible to the dialogue management is by no means trivial. Most frameworks use plug-ins that directly interface to the dialogue core. The multi-modal dialogue platform SiAM-dp (Neßelrath and Feld, 2014) addresses this in a more fundamental way using a modeling approach that allows to share variables or objects between different modules.

In the application domain of social robotic assistants, it is vital to be able to maintain a relationship with the user over a longer time period. This requires a long-term memory which can be used in the dialogue system to exhibit familiarity with the user in various aspects, like personal preferences, but also common knowledge about past conversations or events, ranging over multiple sessions.

In the following, we will describe VOnDA, an open-source framework to implement dialogue strategies. It follows the information state/update tradition (Traum and Larsson, 2003) combining a rule-based approach with statistical selection, although in a different way than OpenDial. VOnDA specifically targets the following design goals to support the system requirements described before:

- Flexible and uniform specification of dialogue semantics, knowledge and data structures
- Scalable, efficient, and easily accessible storage of interaction history and other data, resulting in a large information state
- Readable and compact rule specifications, facilitating access to the underlying RDF database, with the full power of a programming language
- Transparent access to Java classes for simple integration with the host system

Chapter 2

Interaction with the overall system

The interaction 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.

All these inputs are stored as RDF data, based on an ontology developed as part of the interaction manager, and available to all other components as a data format specification.

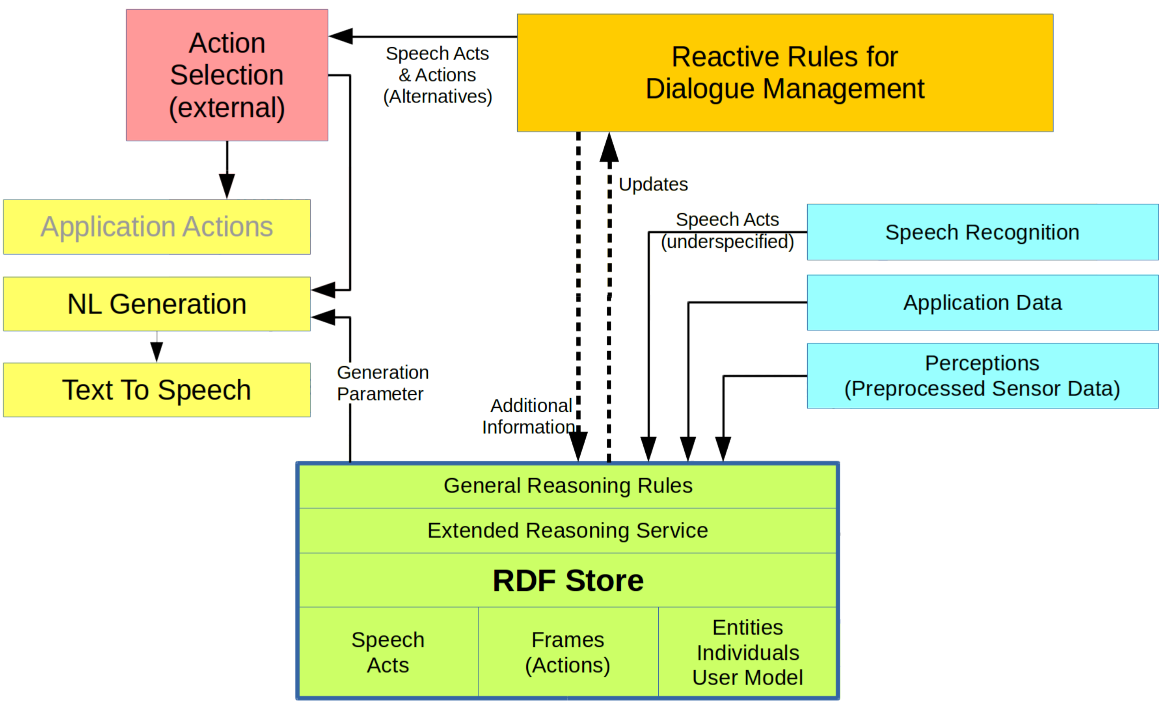
When new data is added, a set of declaratively specified reactive rules will propose dialogue moves or other actions and send these proposals to the action selection mechanism. The selection mechanism eventually selects one of the proposed actions and sends it back. If the proposed action results in 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.

2.1 Internal structure

As shown in the picture below, the interaction manager consists of the RDF store, which also contains the functionality to store incoming data in the format specified by the ontology, thereby making it readily accessible for other components.

The second major component is the rule processor for the dialogue management rules, which generates proposals for actions when new incoming data arrives. The rules not only use the new data, but also the interaction history stored in the RDF database to take its decisions.

The last two parts are a robust natural 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.



2.2 Options for translating rules and .rudi files

Naming convention: a label followed by a colon followed by an *if* is called *rule* from now on.

There are two main options to translate:

A) translate the whole project into one large Java class / method Problems with this approach

  - The relation to the source code is hard to track: no modularity, one large blob
  - The execution regime can not be changed except by changing the translation (no dynamic adaptation of execution)

B) Create a class for each .rudi file and each top-level rule

  - Clear structure that is isomorphic to the .rudi files
  - Dynamic execution strategy is easier to imagine, albeit not really feasible (do we want/need it?)
  - Variables on the top level of a file can be implemented by class fields and fully specified access, such as Introduction.special_variable

We're going for version B), assuming that most of the variables which have non-local scope are state variables in the agent, which also might be specified in special top-level files. These should also contain specifications for general framework functions, i.e., the whole signatures including types for the arguments and return types, to enable advanced type checking.

Similar files could be used for custom user state variables and functions, i.e., for Quiz logic, custom knowledge bases, etc.

Embedded rules will be treated differently to avoid the overhead of handling local scope and lifetime of variables.

`return` statements can have optional labels that indicate the exit point / level, which may be local rule, top-level rule or file. A return without label exits from the innermost scope.

Example for a top-level rule:

```
a:
if ( XXXX ) {
  x = child_27;
  b:
  if (YYYY) {
     x = child_3;
     c:
     if ( ZZZZ ) {
        ....
        if (....) return b:;
        ....
     } // c: ends
     ....
  } // b: ends
} // a: ends
```

The labeled exits are implemented by using Java's label and labeled break functionality.

Chapter 3

Installing and getting started

3.1 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 3.1: 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`.

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

Table 3.2: Available aggregates

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.

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

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 3.2 and 3.2), 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.

3.2 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.

VOnDA's architecture

Figure 3.1 shows the architecture of a runnable VOnDA project.

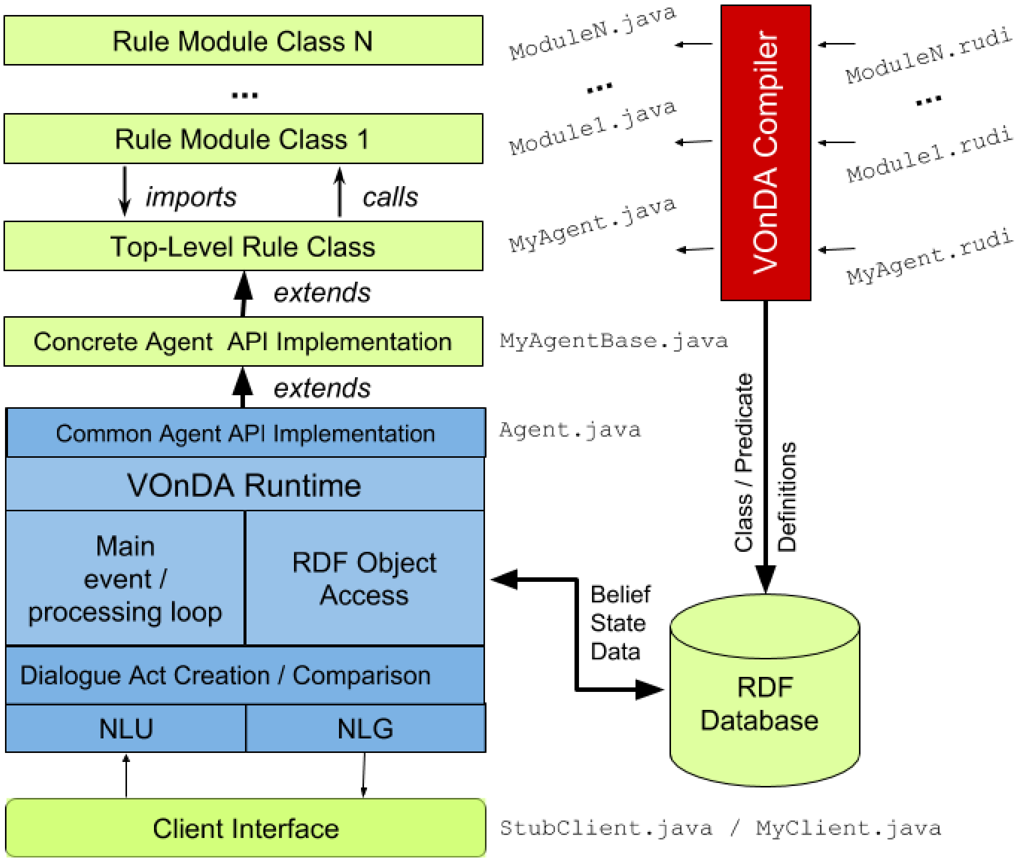


Figure 3.1: Gesamtarchitektur von Rudimant

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 3.2).

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`).

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 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.

visibility

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.

RDF accesses and functional vs. relational properties

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

Agent
name: `xsd:string`
- Animate
  age: `xsd:int`
- Inanimate

Figure 3.2: Ontology and VOnDA code

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.

where to put?: 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.

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

Table 3.4: Examples for an RDF property access

missing creation of child in right row, plus "Set middle" is no proper code

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 3.4 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.

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)`.

Rules and labels

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

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 3.4). 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.

`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.

`timeouts` 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.

Dialogakte und Evaluation mittels ^

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 interface with the natural language understanding and generation units.

Figure 3.4 contains an example of the short-hand notation for shallow semantic structures (starting with #). Since they predominantly contain constant (string) literals, this is the default when specifying such structures. The special "hat" syntax in `user=^user.name` allows to insert the value of expressions into the literal, similar to an *eval*.

Eine wichtige Aufgabe eines Dialogsystems ist zweifellos die Ausgabe von Dialogakten, die von der Generierungskomponente zu Kommunikation mit dem Nutzer verarbeitet werden können. In rudimant ist die Funktion `emitDA` hierfür zuständig.

Argument von `emitDA` ist ein Dialogakt in der oben gezeigten Form. Dass es sich bei `Inform(...)` nicht um einen Funktionsaufruf, sondern um einen Dialogakt handelt, wird hier und an allen anderen Stellen, an denen man einen

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

Figure 3.3: VOnDA code example

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

Figure 3.4: VOnDA code example

Dialogakt benutzen möchte, durch das # gekennzeichnet. Rudimant wird so erkennen, dass im Java-Code eine neue Instanz der Klasse DialogueAct mit den entsprechenden Modifikationen erstellt werden muss. Hierbei kommt dem Marker ^ eine besondere Bedeutung zu: Per default werden alle in den Klammern angegebenen Argumente, egal, ob sie rechts oder links eines = stehen, als Strings an den neuen DialogueAct übergeben. Möchte man explizit den Inhalt einer Variablen oder eines Ausdrucks übergeben, so ist die Benutzung von ^ vor der entsprechenden rechten Seite einer Zuweisung notwendig, die dafür sorgt, dass der Ausdruck evaluiert wird, und nicht als atomares Symbol interpretiert.

Type inference and overloaded operators

Rudimant erlaubt statische Typzuweisungen sowie Casting, beides ist jedoch nicht zwingend notwendig.

Ist beispielsweise der Typ der rechten Seite einer Variablendeklaration mit Zuweisung bekannt oder inferierbar, so ist es nicht notwendig, den Typ der Variablen explizit anzugeben.

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

Table 3.5: Übersetzung komplexer Zugriffe

Als zeitsparendes Feature bietet rudimant insbesondere das automatische Vervollständingen von bool'schen Ausdrücken in den Clauses von if, while und for an. Da in diesem Fall bekannt ist, dass das Ergebnis boolean sein muss, ergänzt rudimant automatisch den Test auf die Existenz eines Objektes in der Clause, sollte dieses nicht Typ boolean sein. Bei Feldzugriffen testet es für jeden Teilzugriff, dass das erhaltene Objekt nicht null ist, um einer NullPointerException zur Laufzeit des generierten Codes vorzubeugen.

Die Vergleichsoperatoren in rudimant sind überladen. Neben den aus Java bekannten Anwendungen als Vergleichsoperatoren können sie auch auf Dialogakte angewendet werden. In diesen Fällen werden sie zu Subsumptionsoperatoren.

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

Table 3.6: Überladene Vergleichsoperatoren

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 ??). It is possible to use generics in these definitions, although they are, for complexity reasons, restricted to be one single uppercase letter.

`myType someVariable;`	es gibt die Variable myVariable vom Typ `myType`
`myType someFunction(typeA a, typeB b);`	Funktion someFunction nimmt Argumente vom Typ `typeA` und `typeB` und gibt ein Objekt vom Typ `myType` zurück (void = void)

Table 3.7: Defintions of existing Java fields and methods for VOnDA

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:

`[type]. myType Function(typeA a);`	Funktionsdeklaration wie oben; die Funktion muss auf einem Objekt der Klasse `type` aufgerufen werden. Generics sind eingeschränkt erlaubt, mit T als festgelegtem Namen der Parameter-Klasse, Beispiel: `[List<T>]. T get(int i);`

Table 3.8: Definition of a non-static method of Java objects

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.

is it still true that this also accounts for declarations in rudi?

Functional constructs

is this still true?

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` ist ein Schlüsselwort in rudimant. Eine Zeile "`import File;`" auf globaler Ebene, an einer beliebigen Stelle zwischenn den Regeln, bedeutet die Inklusion der Datei File.rudi an exakt dieser Stelle. Hiermit wird zum einen erreicht, dass rudimant auch diese Datei zu ausführbarem Java-Code kompiliert, sodass nicht alle Dateien eines Projektes separat kompiliert werden müssen. Zum anderen wird das vollständige Regelinventar der Datei an der entsprechenden Stelle im Java-Code mithilfe ihrer process-Methode aufgerufen und durchgearbeitet.

`import` ermöglicht es also, das Regelinventar für ein Projekt auf mehrere Dateien bzw. mehrere Module aufzuteilen und diese zusammenzustecken, sodass ein einziger Aufruf ausreicht, um sie zu kompilieren und später auch, um die Auswertung der Regeln anzustoßen. Dies ist nicht nur nützlich zur Übersichtlichkeit und Strukturierung des Projektes, sondern fördert auch die Modularität, da verschiedene 'Unterbäume' der `import`-Hierarchie leicht ausgeklinkt oder aus anderen Projekten wiederverwendet werden können.

```
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 3.9: Funktionen, die mit Lambda-Ausdrücken benutzt werden können.

Java-Code verbatim im Regelfile

Rudimant verarbeitet eine Regelsprache, die mit voller Absicht nur eingeschränkte Java-Funktionalitäten zur Verfügung stellt. Was in rudi-Code nicht umzusetzen ist, sollte in Funktionen in die übergeordnete Java-Klasse wandern (vgl. **??**).

Für Fälle, in denen dies nicht genügt und vor Ort eine Funktionalität benötigt wird, die rudimant nicht parsen oder nicht richtig darstelllen kann, ist eine verbatim-Umgebung vorgesehen. Alles, was innerhalb der Zeichenfolgen `/*@` und `@*/` steht, wird behandelt wie ein mehrzeiliges Java-Kommentar, wird also in genau diesem Aussehen in den resultierenden Code übertragen. Dabei werden die klammernden Zeichenfolgen ausgelassen.

Diese Funktionalität kann insbesondere dazu benutzt werden, am Anfang einer `rudi` Datei Java-Klassen zu importieren, die im kompilierten Code gebraucht werden.

3.3 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.

Default-Funktionalität im Laufzeitsystem

Alles was in `Agent` bereitgestellt wird. Die aktuelle Liste der bereitgestellten Funktionen finden sich in `rudimant` unter `src/main/resources/Agent.rudi`.

- timeouts

  ```
  void newTimeout(String name, int millis);
  boolean isTimedOut(String name);
  void removeTimeout(String name);
  boolean hasActiveTimeout(String name);
  ```

- Senden von Dialogakten an die Generierung

  ```
  DialogueAct emitDA(int delay, DialogueAct da);
  DialogueAct emitDA(DialogueAct da);
  ```

- Zugriff auf DialogAkte aus der 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();
```

3.4 Debugger/GUI

VOnDA comes with a GUI (Biwer, 2017) that helps navigating, compiling and editing the source files belonging to a project. It uses the project file to collect all the necessary information.

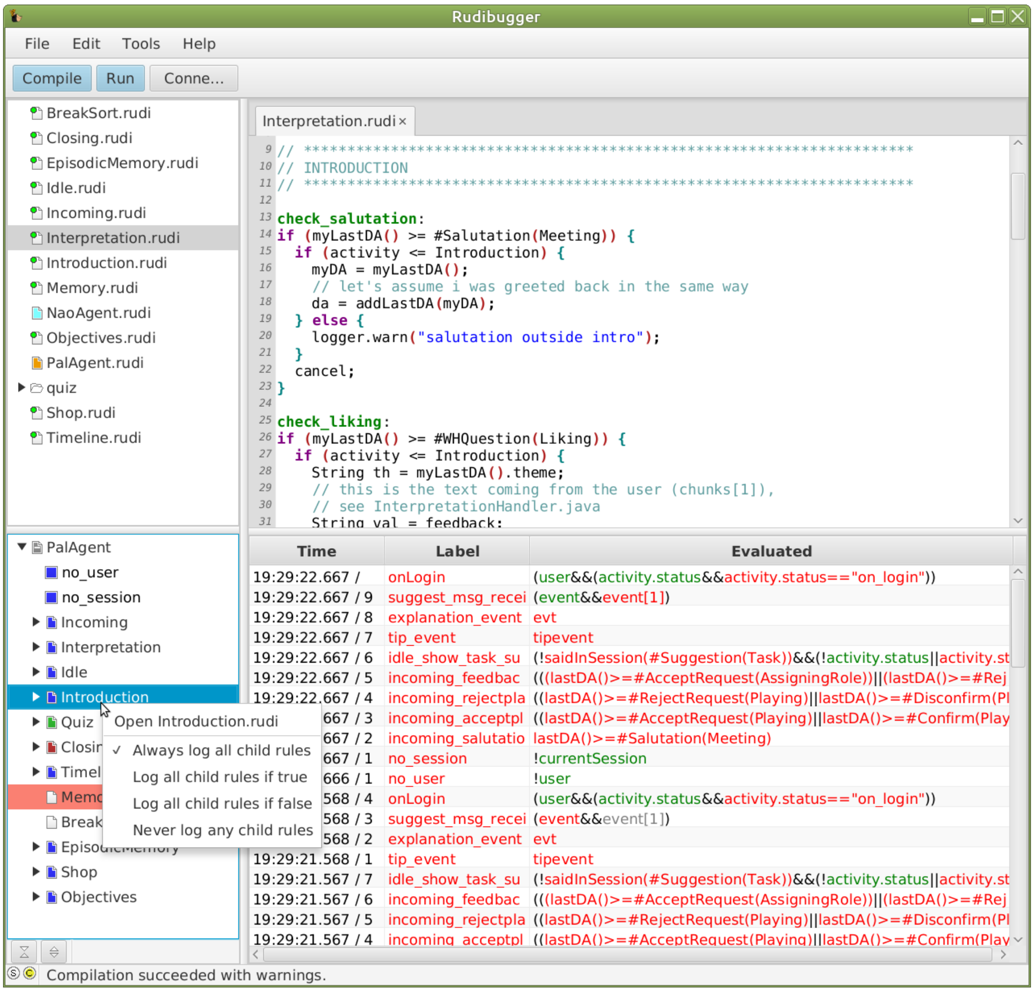


Figure 3.5: The VOnDA GUI window

Upon opening a project, the GUI displays the project directory (in a *file view*). The user can edit rule files from within the GUI or with an external editor like Emacs, Vim, etc. and can start the compilation process. After successful compilation, the project view shows what files are currently used, and marks the top-level and the wrapper class files. A second tree view (*rule view*) shows the rule structure in addition to the module structure. Modules where errors or warnings were reported during compilation are highlighted, and the user can quickly navigate to them using context menus.

Additionally, the GUI can be used to track what is happening in a running system. The connection is established using a socket to allow remote debugging. In the rule view, multi-state check boxes are used to define which rules should be observed under which conditions. A rule can be set to be logged under any circumstances, not at all or if its condition evaluated to true or to false. Since the rules are represented in a tree-like structure, the logging condition can also be set for an entire subgroup of rules, or for a whole module. The current rule loggging configuration can be saved for later use.

The *logging view* displays incoming logging information as a sortable table. A table entry contains a time stamp, the rule's label and its condition. The rule's label is colored according to the final result of the whole boolean expression, and each base term is colored accordingly, or greyed out if short-cut logic led to premature failure or success of the expression.

Bibliography

ALIZ-E (2010). ALIZ-E project. http://www.aliz-e.org/.

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

Bohus, D. and Rudnicky, A. I. (2009). The RavenClaw dialog management framework: Architecture and systems. *Computer Speech & Language*, 23(3):332–361.

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.

Gebhard, P., Mehlmann, G., and Kipp, M. (2012). Visual SceneMaker—a tool for authoring interactive virtual characters. *Journal on Multimodal User Interfaces*, 6(1-2):3–11.

Jurčíček, F., Dušek, O., Plátek, O., and Žilka, L. (2014). Alex: A statistical dialogue systems framework. In *International Conference on Text, Speech, and Dialogue*, pages 587–594. Springer.

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*.

Kruijff-Korbayová, I., Colas, F., Gianni, M., Pirri, F., de Greeff, J., Hindriks, K., Neerincx, M., Ögren, P., Svoboda, T., and Worst, R. (2015). TRADR project: Long-term human-robot teaming for robot assisted disaster response. *KI-Künstliche Intelligenz*, 29(2):193–201.

Lison, P. and Kennington, C. (2015). Developing spoken dialogue systems with the OpenDial toolkit. *SEMDIAL 2015 goDIAL*, page 194.

Neßelrath, R. and Feld, M. (2014). SiAM-dp: A platform for the model-based development of context-aware multimodal dialogue applications. In *Intelligent Environments (IE), 2014 International Conference on*, pages 162–169. IEEE.

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.

Schwartz, T., Zinnikus, I., Krieger, H.-U., Bürckert, C., Folz, J., Kiefer, B., Hevesi, P., Lüth, C., Pirkl, G., Spieldenner, T., et al. (2016). Hybrid teams: flexible collaboration between humans, robots and virtual agents. In *German Conference on Multiagent System Technologies*, pages 131–146. Springer.

Traum, D. R. and Larsson, S. (2003). The information state approach to dialogue management. In *Current and new directions in discourse and dialogue*, pages 325–353. Springer.

Ultes, S., Barahona, L. M. R., Su, P.-H., Vandyke, D., Kim, D., Casanueva, I., Budzianowski, P., Mrkšić, N., Wen, T.-H., Gasic, M., et al. (2017). Pydial: A multi-domain statistical dialogue system toolkit. *Proceedings of ACL 2017, System Demonstrations*, pages 73–78.