Jazzyk

–

Programming language for cognitive agents

Peter Novak

January 10, 2009
Version 1.20

Abstract

Jazzyk - interpreter of the *Jazzyk* programming language (*In varietate concordia!*)

1 Synopsis

jazzyk [*options*] [*file*]

2 Description

Jazzyk is an experimental, special-purpose programming language for development of knowledge intensive (intelligent) agent systems. *Jazzyk* agents consist of

- a number of knowledge bases, each realized by a separate
- specialized knowledge representation module (plug-in), and an agent program in a form of a set of possibly nested rules of the basic form: `when Query then Update`.

Jazzyk was designed to exploit power of heterogeneous knowledge representation (KR) technologies in a single agent system. Each such a KR technology is encapsulated in a separate *Jazzyk* KR module providing a simple generic interface consisting of a set of query and update operations. Semantics of *Jazzyk* based on *Behavioural State Machines*, an adaptation of computational model of Gurevich's *Abstract State Machines*.

Theory of *Behavioural State Machines*, and in turn also *Jazzyk*, draws a strict distinction between agent's knowledge representational and behavioural aspects. While an agent's deliberation abilities reside in its KR modules, its behaviour are encoded as a Behavioural State Machine.

3 Options

3.1 General options:

–help display the help message and exit

–version output version information and exit

–license display the GNU GPL license information and exit

3.2 Compiler options:

-0 [–stdin] after reading in program files, read also standard input. This option is useful to execute for example automatically generated programs, or programs preprocessed by a specialized filters.

-I [–include *path*] macro preprocessor include path(s); Provided include paths will be passed to the internal macro preprocessor invocation via **-I** option. By default, the macro preprocessor searches only in the current path (path in which the interpreter was invoked (see *pwd*(1))). The source code can include also files from other then the current directory, but it has to use relative paths to find the included file correctly. Otherwise all the paths where include files reside, should be passed to the interpreter using **-I** options.

–no-mp bypass the macro preprocessor; Internal macro preprocessor will not be invoked on the input file(s) and the content will be passed directly to the compiler. This option can be useful in the case the input source code either does not use any higher level macro overlay definitions, or it is already preprocessed by the macro preprocessor.

3.3 Interpreter options:

-L [–libraries *path*] add an absolute system path to the location of external plug-ins; Plug-ins are standard shared dynamically loadable libraries and by default the interpreter searches in the standard system paths where libraries are present. Use this option in the case you have KR plug-ins installed in a non-standard location (w.r.t. your OS), or the interpreter has difficulties to find the requested plug-ins.

-o [–ordered] pick the first applicable rule to apply from a set transformer; By default, when the interpreter executes a set mental state transformer {<transformer> ; <transformer}, it first randomly shuffles the transformer set and then searches for an applicable rule from the beginning. This way a random applicable rule is chosen from the original set transformer. Using this option disables the random shuffling step and lets the interpreter to choose the first applicable rule of the set transformer w.r.t. the ordering as defined in the source code.

-n [–cycles *num*] perform only *num* interpreter cycles and quit; 0 (default) means endless interpreter cycle loop.

-q [–no-check-query] switch off checking sanity of resulting query variable substitutions. Modules are allowed to not substitute all the provided free variables.

3.4 Debug options:

-e [–only-macros] print the output of macro preprocessor run only; do not compile, or interpret; Using this option amounts to the same effect as if *GNU M4* (*m4*(1)) were executed with the default options described later in this manual and the corresponding input file (or standard input stream).

-E [–compile] run preprocessor and compile only; do not interpret; This option is useful for verifying whether the program is interpretable. This means that it is well-formed according to the syntax rules of the *Jazzyk* language and additional semantical constraints hold: each module referenced either in update, query, or notification expression/statement is also declared in the program (Note: it can be declared also after the first use!)

-p [–print] pretty print the program tree structure after compilation stage; This is a convenience feature to check that the compiler understood the structure of the program correctly. Currently it prints only a rough structure of the program. In the future it should print the executable formatted source code of the input program.

4 Diagnostics

When failing, the interpreter provides informative error messages in the standard compiler format with the location in the source file and an error message. The error output can be parsed and processed by IDEs and text editors like *vim*(1), *emacs*(1) and others. Messages are written to the standard error output.

During the internal invocation of the *GNU M4* (*m4*) macro preprocessor, the interpreter forwards whatever error messages of *m4* as well.

4.1 GNU M4 invocation

GNU M4 macro preprocessor is internally invoked with the following default arguments:

```
$ m4 -s -E -I $(PACKAGELIBDIR) ...
```

Where `$(PACKAGELIBDIR)` stands for the default installation directory where internally used shared macros are placed. When the default installation prefix is used, this should be `/usr/lib/jazzyk`. For more details, see the installation instructions of the *jazzyk* package and help message of its *configure* script.

For more details on the *m4* options semantics see *m4*(1).

5 Bugs

As of time of writing this manual, no issues and bugs are known to me. For more details see the `Changelog` file. In the case you will spot any bugs, or problems, or

you have some enhancement/feature request,do not hesitate and contact the author, or maintainer.

6 Author

The *Jazzyk* interpreter was written by Peter Novak `pno at aronde.net` as a result of his research work towards PhD. degree in *Computational Intelligence Group* of *Clausthal University of Technology*, Clausthal, Germany
(`http://cig.in.tu-clausthal.de`).

Peter Novak <`pno at aronde.net`>, `http://peter.aronde.net/`

7 See also

GNU M4 *m4*(1).

8 Publications

The list of papers related to *Jazzyk* until February 2008.

1. *Peter Novak, Juergen Dix*:
   Modular BDI Architecture
   Proceedings of Fifth International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS'06,
   Future University, Hakodate, Hokkaido, Japan, pg. 1009-1016
   Association for Computing Machinery/2006, May 2006

2. *Peter Novak, Juergen Dix*:
   Adding structure to agent programming languages
   Proceedings of Fifth International Workshop on Programming Multi-Agent Systems ProMAS'07,
   Hawai'i, USA, May 2007

3. *Peter Novak*:
   Behavioral State Machines: programming modular agents
   AAAI 2008 Spring Symposium: Architectures for Intelligent Theory-Based Agents, AITA'08
   AAAI Press, Menlo Park, California, USA, March 2008

4. *Peter Novak*:
   Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations
   Proceedings of Sixth International Workshop on Programming Multi-Agent Systems, ProMAS'08
   Estoril, Portugal, May 2008

9 Acknowledgements

Prof. Dr. Juergen Dix participated on development of the underlying theory of the *Jazzyk* programming language. *Bernd Fuhrmann* (at the time a diploma student at Computational Intelligence Group) developed a prototype of the *Jazzyk* interpreter (labelled *Jazyk*) and provided a valuable input in crafting deeper technical details of the *Jazzyk* interpreter. I thank them.

10 Copying permissions