I/O Facilities for XTT Based Systems[*]

Igor Wojnicki
Institute of Automatics,
AGH – University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
wojnicki@agh.edu.pl

Abstract

This paper proposes a method of environment interaction for XTT based rule systems. XTT is a methodology and a set of supporting technologies for implementing rule-based systems. The technologies provide a development environment and the inference engine. XTT is based on attributive logic. There are some attributes designated as communication means. For such attributes appropriate so-called *trigger routines* are provided. The *routines* are written in an external to XTT language to perform I/O operations. Upon referencing such an attribute the inference engine calls appropriate routine. In this way the knowledge base remains highly declarative and it is not focused on technical details of I/O operations.

Introduction

A rule-based system consists of a knowledge base and an inference engine. The knowledge engineering process aims at designing and evaluating the knowledge base, and implementing a proper inference engine (Liebowitz 1998; Jackson 1999; Negnevitsky 2002). However, at some point a rule-based system (RBS) needs to be integrated with underlying OS or other applications. Such integration is necessary in order to enable an RBS to process files, data streams, react to stimuli (in case of control systems), and communicate with users through user interfaces.

This paper presents concepts which allow to integrate an XTT (Nalepa & Ligęza 2005) based system with the environment providing I/O communication means. For the purpose of this paper the environment is defined as all the software the rule-based system interfaces with, including data streams, I/O devices, embedding application and underlying operating system. A term *environment interaction* and *I/O* are used interchangeably.

The *XTT* (*EXtended Tabular Trees*) knowledge representation (Nalepa & Ligęza 2005), has been proposed in order to solve some common design, analysis and implementation problems present in RBS. At the visual level It is composed of decision tables. A single table is presented in Fig. 1 (this

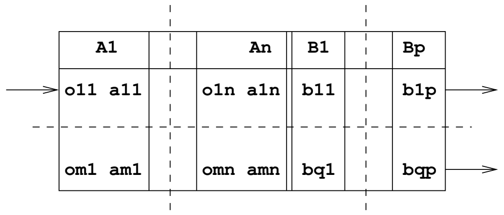

Figure 1: A single XTT table.

is XTT^2, a refined version of the original XTT, more details can be found at hekate.ia.agh.edu.pl, XTT^2 and XTT are used interchangeably in this paper). The table represents a set of rules based on the same attributes. A single rule can be read as follows:

```
IF (A1  o11  a11) and...(An  o1n a1n)
THEN (B1=b11) ,...(Bp=b1p)
```

where A1...An are attributes used in the condition part, o11...o1n are logical operators, B1...Bp are attributes used in the decision part; a11...a1n and b11...b1p are expressions evaluating to single values or sets of values. Every attribute has assigned a type and an indication whether it can hold single or multiple values. A type is identified by its unique name and it defines: a base type (numeric, string, boolean), optional lenght/precision, and a domain which is a set of admissible values.

XTT includes two main extensions compared to the classic RBS: non-atomic attribute values (sets of values are allowed to be used both in conditions and decisions), non-monotonic reasoning support (assignments in the decision part apply to sets of values providing assert/retract like operations). Each table row corresponds to a decision rule. Rows are interpreted top-down. Tables can be linked in a graph-like structure. A link is followed when a rule is fired.

At the logical level a table corresponds to a number of rules, processed in a sequence. If a rule is fired and it has a link, the inference engine processes the rule in another table the link points to. The rule is based on an *attributive language* (Ligęza 2006). It corresponds to a *Horn* clause: $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee h$ where p_i is a literal in SAL (Set Attributive Logic, see (Ligęza 2006)) in a form $A_i(o) \in t$ where $o \in O$ is a object referenced in the system, and

[*]The paper is supported by the *HEKATE* Project funded from 2007–2009 resources for science as a research project.

$A_i \in A$ is a selected attribute of this object, $t \subseteq D_i$ is a subset of attribute domain A_i. Rules are interpreted using a unified knowledge and fact base, that can be dynamically modified during the inference process using set based assignments in the rule decision part. This approach has been successfully used to model classic rule-based expert systems.

XTT has been extended and redesigned several times to make it more robust, flexible and easy to use (Nalepa & Ligęza 2005; Nalepa & Wojnicki 2007; Wojnicki & Nalepa 2007). This paper focuses on issues regarding interactions between an XTT based rule system and the environment. It briefly describes approaches taken by common rule-based systems and proposes a new method of environment interaction for XTT with some illustrative examples.

Interactions Between Rule-based Systems and Their Environment

Rule-based systems offer a wide range of approaches to the environment interactions. There are CLIPS, Jess, Drools and Kheops I/O facilities briefly described in this section. Some summary and analysis are given below as well.

CLIPS (Giarratano & Riley 2005) is a development and delivery tool for building forward chaining expert systems. It can be used as a standalone application or embedded within a C language application.

CLIPS uses a concept of *logical names* to refer to I/O devices. There is a set of predefined *logical names* which refer to the default user input (`stdin`), output (`stdout`), error (`werror`), and warning (`wwarning`) outputs. To perform I/O operations on *logical names* there is a set of functions provided. They allow for creating new *logical names* associated with files, reading from and writing to them. There is also a set of functions which allow to process such incoming or outgoing data in terms of data formatting or adapting it to the requirements of the knowledge base. Such functions can be used within rules to provide actual data processing.

Furthermore there is a way to create *logical names* to communicate with virtually any data source or I/O device. To accomplish that CLIPS introduces so-called *I/O Router System*. It provides infrastructure for defining user defined functions handling I/O operations regarding particular *logical name*. Such a user defined set of functions is called a *router* and it must be programmed in C language (or any other linkable with C). The *I/O Router System* extends I/O operations accessible from CLIPS beyond file operations and built-in set of *logical names*.

Data can be read from or written to the environment upon firing the rule then. The data processing itself is provided through CLIPS code within rules or through user defined functions handling particular *logical name* coded in C (a *router*). Both approaches reveal their imperative nature, there is a set of instructions which needs to be run in order to process incoming or outgoing data. It would be more convenient to keep as much of this processing outside of the knowledge base rules. In this way the knowledge base does not shift its focus towards handling I/O.

From CLIPS perspective interacting with a user provided *router* is more declarative. Data, which is read from or written to the *logical name*, handled by the *router* can be processed by the *router* instead of rules. No processing within the rules is required then.

CLIPS can also interact with the environment through user defined functions. Such a function can provide a value which comes from, or goes to, any I/O device, user interface etc. User defined functions can be called from within rules.

There is also a fourth option available, if CLIPS is embedded within a C language application. It is to use CLIPS API to alter knowledge-base directly from the embedding application.

To summarize. There are four ways the environment interactions can be carried out from CLIPS: file based (with use of *logical names*), *router* based, user defined function based, and API based. The file based is reading from or writing to a file which is delivered by the underlying OS. All data processing have to be provided through CLIPS function calls within the rules. The router based approach provides a set of user defined functions implementing basic primitives for reading from and writing to a logical name. These functions implement communication with the environment. Function calls handling *logical names* are used within the rules to trigger the I/O interactions then. The user defined function approach provides functions available to CLIPS which call corresponding functions written in other languages (such as C, Ada etc.) providing the I/O communication (such functions can be called from within rules). The API based approach assumes that the environment is pushing appropriate data into or reading from the knowledge base directly through using CLIPS API. Rules cannot trigger any environment interactions themselves then. It is entirely up to the programmer which of these approaches are taken.

Jess (Friedman-Hill 2003) is designed as a library allowing to embed a rule-based system into a Java application. Since Jess is inspired by CLIPS it inherits its environment interaction concepts. The main difference between CLIPS and Jess is in their implementations. CLIPS is implemented in C while Jess in Java.

Jess also utilizes the concept of *I/O Router System* and *logical names*. There are several classes implemented as Jess API which support handling *logical names* and *routers* (Friedman-Hill 2003).

User defined functions are also provided. Such functions must be implemented in Java. The user defined function facilities include an ability to call an arbitrary method from a rule.

A direct knowledge-base manipulation from Java through Jess API calls is also possible. The main class of Jess handling the inference process (`jess.Rete`), provides several methods which allow to add, remove and modify facts, rules, and other Jess objects easily.

To summarize, Jess provides similar concepts for I/O communication as CLIPS does. The only major difference regarding I/O is that Jess is implemented in Java while CLIPS is implemented in C. It is also up to the programmer how declarative the I/O handling should be – how much of data processing is implemented by rules and how much by Java methods.

Drools (`www.jboss.org/drools`) is a rule engine for JBoss application server, also known as JBoss Rules. It is a production rule system which can be integrated with business applications complying with the Java Enterprise Edition specification.

A Drools rule can launch any method or function upon firing (function is defined as a named and parametrized set of instructions: Java statements, method calls etc.). Launching such methods provide means for communication with the environment. Drools also allows to pass named instances of objects to the inference engine. In such a case any method of the object can be subsequently called upon firing a rule or checking rule conditions.

Furthermore there is an API which allows to alter rule-base from Java. It also allows to set up listeners which allow to react to certain knowledge-base states and changes.

Comparing with Jess and CLIPS, Drools lacks the router concept. The I/O communication and processing is either implemented within rules or it could also be provided by Java methods. Drools API calls allow to set up listeners reacting to certain knowledge base conditions. It is also up to the programmer which method to choose.

Kheops (Gouyon 1994) is a development environment and a standalone inference engine with a support for designing real-time rule-based systems to be suitable for implementing control systems. The inference engine could also be embedded in a C language application.

Kheops represents facts in terms of attribute values. It relies on a finite set of such attributes being propositional variables. The knowledge base defines a consistent mapping from the *input space* to the *output space* using rules. Kheops I/O is based on declaring attributes belonging to *input* or *output* spaces. Attributes assigned to the *input space* provide input (input attributes), their values are read from the environment, while attributes assigned to the *output space* provide output (output attributes), their values are written to the environment. Attributes which do not belong to the *input* or *output* spaces are assigned to the *intermediate space* which expresses knowledge base state.

Input attribute values have to be known prior to running the inference process. They can be either read from a file or standard input. When the inference process is concluded output attributes hold values being the result of the process. These values can be written to a file, or standard output.

In addition to the above I/O concept any C language function calls can be embedded within a rule.

Interactions with Environment: Conclusions

Rule-based systems presented above offer a wide range of facilities to perform interactions with the environment. I/O operations either trigger the inference process (delivering facts), they are the result of the inference process upon its completion, or they are performed during rule firing as a part of the condition or action. In most cases showed above, it is up to the programmer where the I/O data processing takes place. It can be performed within the rules by the inference engine, or outside of it, performed by the environment (C functions, Java methods, OS calls, services etc.).

There are five types of data processing and interaction with the environment:

- rules,
- routers,
- user defined functions,
- API calls,
- designated attributes.

If I/O data processing is performed by rules the inference engine has to provide appropriate facilities (functions which can be called from within rules) which allow for such processing. As a result rules may contain a lot of code which does not serve the purpose of the system itself: solving a given problem, but instead they implement low level data processing (i.e. writing some inferred data, facts or attribute values, to a file in some given form; in such a case the rule has to perform a set of actions implementing I/O operations: open a file, format data, write data to the file, close the file).

Regarding the *routers* concept (CLIPS, Jess), technical details of the communication are covered by external functions, while the inference engine operates on *logical names* delivered by these functions. The logical names provide handlers to data streams but still the rules have to implement opening, data formatting and closing. The only thing which is covered by the *routers* concept is details regarding where and how data is transmitted.

An I/O operation might be just calling an external function (or method) from a rule. Only a single action is required to trigger such an interaction then: calling appropriate function. Technical details of the I/O operations are carried out by that function instead of the rule. Such a rule base carries the system logic, while technical details of delivering input and output are covered by the environment. The inference engine does not care whether the I/O operation is about writing to a file, data stream, network port, or calling a remote service. Such a an approach increases interoperability and portability of such a rule base.

There are also purely declarative approaches (Kheops) where the I/O is precisely defined and no opening, data formatting, and closing takes place. There are well defined means for I/O communication available through arbitrarily chosen attributes being input or output. Before the inference process takes place all inputs have to be delivered by the environment. When the inference process is concluded all outputs are available to the environment. The inference engine do not take care about I/O details, the environment does. This approach is purely declarative. All details regarding I/O operation are covered by the environment. The inference engine deals with attributes which are declared as input or output and nothing more, it does not care about how data is actually transmitted. Such a clean approach has some disadvantages though. The inference process is run in turns. Inputs are loaded, the inference process is performed, the outputs are saved. No environment interaction is possible during the inference process.

To conclude, assuming that an I/O operation is just a technical mean for data flow, its processing should not be performed within rules themselves, or such processing should be minimized. Putting it within rules makes them focus on actual I/O data processing instead of the core issues the system is meant to deal with.

Depending on programmer the rule base can contain more or less I/O data processing then. It can be more or less readable, more or less declarative, more or less focused on the main goal of the system. There should be a clear separation then. The knowledge base should provide the system logic while appropriate data flow to and from the environment should be established through external routines. The association of these routines with the knowledge base should be as declarative as possible.

XTT Environment Interaction

The first approach to the XTT interaction with the environment (Nalepa & Ligęza 2004; Ligęza 2006) was similar to that of the Kheops system. Each attribute is assigned either to be *input*, *output*, or *middle*. Before the inference process starts all input attribute values are set, passed to the inference engine from the environment. When the inference process concludes, its results are available through output attributes which values are passed to the environment. An attribute designated as *middle* is a regular attribute which is a part of the knowledge base, not being directly involved in I/O operations. This approach is clean and easy to implement, however it does not provide any means for I/O during the inference process.

The second approach was to use *hybrid operators* concept (Wojnicki & Nalepa 2007). The *hybrid operators* is a way to call arbitrary external functions (written in Java or Prolog) which might deal with I/O. This approach introduces the environment interaction during the inference process. The inference process itself can be guided or controlled by the environment then. However, inputs and outputs are not clearly defined anymore. A hybrid operator providing I/O operations can be used anywhere in a rule. It is very flexible but decreases clarity of the logic (rule base) of the system.

The contemporary approach, presented in this paper, defines that the environment interactions are provided through designated attributes only. It is similar to the first approach with *input*, *output* and *state* attributes. There is a strict declaration of inputs and outputs. Each attribute is assigned to a class: *ro*, *wo*, *rw*, or *state*. If the attribute value is to come from the environment it should be assigned to the *ro* (read-only) class. It also indicates that the inference engine is not allowed to change its value as a result of an action, the only way to change its value is to read it from the environment. For each *ro* attribute a routine (a function, method or predicate depending on language and the environment being used) has to be provided, so-called *ro trigger routine*. It implements data transfer from the environment into the attribute. The routine takes no arguments and it returns a new value for the attribute upon calling. An *ro* attributes are allowed in the condition part of a rule only. Upon referencing such an attribute appropriate *ro trigger routine* is called by the inference engine and the attribute value is set. Error conditions regarding reading values into *ro* attributes from the environment have to be covered by valid and defined attribute values which indicate them. If there is only a single error condition it can be indicated by N/D (Not Defined) value.

If the attribute value is to be written to the environment the attribute should be assigned to the *wo* (write only) class. Such an attribute is mainly used in the condition part of a rule. To provide a feedback regarding successful writing operation a *wo* attribute can be used in the condition part of a rule as well. It might not provide actual data then, but values which indicate the state of the previous writing operation. If there is only a single error condition it can be indicated with N/D (Not Defined) value. Particular attribute values, including error codes, are up to the programmer. For each *wo* attribute a routine (a function, method or predicate depending on language and the environment being used) has to be provided, so-called *wo trigger routine*. It implements data transfer from the attribute into the environment. The routine takes attribute value which is to be written as an argument and returns a value which the attribute is set to. It can be subsequently used to detect an error condition. Setting a *wo* attribute value triggers appropriate *wo trigger routine*.

If there is a need for a bi-directional communication it can be established by assigning the attribute to the *rw* class. Its value can be both read from or written to the environment. An *rw* attribute should have assigned two *trigger routines*, one for reading, one for writing: an `ro trigger routine` and a *wo trigger routine*.

A *state* attribute has no *trigger routines* assigned, since such an attribute does not provide any means for communication with the environment.

The *trigger routines* can be written in languages supported by the inference engine. Targeted languages are Prolog and Java.

Such an approach is highly declarative. The knowledge base does not deal with details regarding interfacing with the environment, it is focused on the system logic. Furthermore, it allows to run the inference engine interactively depending on values exchanged with the environment.

Assigning attributes to a class (`ro, wo, rw, state`) is provided while declaring them. However association between particular *ro*, *wo* and *rw* attributes and proper *trigger routines* is provided through additional XML based language: IOML (Input Output Markup Language) which should be available upon run-time. Such an approach separates the rule base (assuming that it holds the system logic) from the environment and makes it possible to exchange the rule base between different run-time environments. For example reading attribute values from a file can be easily replaced by reading them from a stream, a network socket, or a keyboard without altering any data in the rule base. It would require just an assignment of a different `trigger routine`. The rule base is focused on the problem to solve, while IOML allows to connect the inference engine with the environment.

Example

In Fig. 2 there is an example illustrating XTT environment interactions. A goal is to identify somebody's sex based on his or her name. Attributes present in the system are given in

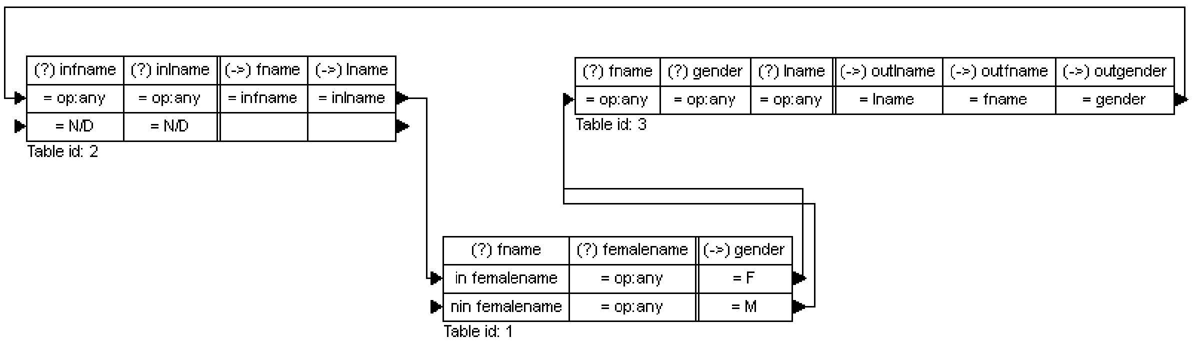

Figure 2: Gender Case, XTT

Name	Type	Value	Class
infname	tname	single	ro
inlname	tname	single	ro
fname	tname	single	state
lname	tname	single	state
femalename	tname	multiple	ro
gender	tgender	single	state
outlname	tname	single	wo
outfname	tname	single	wo
outgender	tgender	single	wo

Table 1: Gender Case, Attributes

Table 1. The attribute types are defined as follows: tname is a string of any length, tgender is a string of any length, allowable values: 'F', 'M'.

The knowledge base consists of three XTT tables (see Fig. 2). The inference process starts with Table id: 2. Attributes: infname, inlname provide actual names which are read from the environment. The first row defines a rule which reads: if infname has a value (op:any means *any value*) and inlname has a value then assign fname a value of infname and assign lname a value of inlname, then follow to Table id: 1. The second row defines a rule which is fired if infname and inlname have assigned no values which takes place if there are no more names coming from the environment. The inference process ends then.

The rules in Table id: 1 identify whether the first name is a female name or a male one. The first rule checks if a value of fname matches at least one value (in operator) in a set of values assigned to femalename – the actual values of femalename come from the environment. If the check is positive then gender is assigned F. Otherwise the second rule fires which assigns gender a value of M. In both cases the inference engine proceeds to Table id:3.

Table id:3 assigns appropriate values to the *wo* attributes generating actual output. The inference engine gets back to Table id:2 to process another input record, which is reading new values from infname and inlname attributes. It is worth pointing out that there is no checking if the writing operation in Table id:3 succeeded.

First and last names are given as a textual data stream of records separated with new lines on the standard input. Each record consists of two strings: first and last names separated with a space. The names from the input stream are available as infname and inlname attributes respectively (*ro* class). Upon referencing (see the condition part of Table id:2, Fig. 2) infname is assigned a first name. The infname *ro trigger routine* reads a string from the standard input until it encounters a space. The inlname *ro trigger routine* reads a string from the standard input until it encounters a new line character, inlname is assigned a last name then. The *ro trigger routines* specified for infname and inlname read from the same data stream.

Another *ro* attribute is femalename. This attribute, upon referencing in Table id:1, holds a set of values which are female first names. It has a *ro trigger routine* assigned which populates femalename with these values. The *trigger* routine providing the values has them hardcoded.

There are also *wo* attributes: outlname, outfname, and outgender which have wo trigger routines assigned. These routines generate an output data stream. Firing the rule in Table id:3 the outlname *wo trigger routine* is called. It outputs the current value of outlname on standard output and a space. Then, the outfname *wo trigger routine* is called which outputs outfname value and a space. Finally, the outgender *wo trigger routine* is called which outputs outgender value and a new line character. The output data stream consists of records separated with new lines then. Each record consists of a sequence of strings: a first name, a last name, and a gender. The gender is a single letter being either `F` (female) or `M` (male), according to Table 1.

All environment interactions are provided through appropriate *trigger routines* then.

Summary and Further Research

This paper proposes an environment communication method for XTT based systems. XTT is a methodology for designing rule-based systems. It is strongly supported with technologies providing the design environment and the inference engine.

XTT is based on attributive logic (Ligęza 2006). There are four classes of attributes: *ro*, *wo*, *rw*, state. Attributes assigned to first three classes are involved in the environment interactions providing input, output and input/output respectively. Each of these attributes has assigned so-called *trigger routines* which implement the communication. The *trigger routines* are written in an external to XTT language (targeted languages are: Java and Prolog). Assignment of the *trigger routines* to the attributes is provided through a separate language IOML. In this way the knowledge base remains highly declarative and it is not focused on technical details of I/O operations.

Upon referencing an *ro* attribute appropriate *trigger routine* is called and the value is delivered to the inference engine. Upon assigning a value to a *wo* attribute appropriate *trigger routine* is called which delivers the value to the environment.

The concept of *trigger routines* is based on observations and analysis of contemporary technologies for developing rule-based systems such as: CLIPS (Giarratano & Riley 2005), Jess (Friedman-Hill 2003), Drools (`www.jboss.org/drools`) and Kheops (Gouyon 1994).

Implementation of the *trigger routines* in Prolog is straightforward since Prolog is the language the inference engine interpreting XTT rules is written in. Providing them as Java methods is under development now.

Further research focuses on implementation and optimization of the communication layer allowing to provide *trigger routines* as Java methods. Furthermore, there is a library of *trigger routines* being formulated. Such a library will allow to handle I/O communication in various ways including: reading from and writing to a plain text file, stream, keyboard, structured documents (such as XML application) or GUI (Graphical User Interface) widgets, as ready to use *trigger routines*.

References

Friedman-Hill, E. 2003. *Jess in Action, Rule Based Systems in Java*. Manning.

Giarratano, J. C., and Riley, G. D. 2005. *Expert Systems*. Thomson.

Gouyon, J.-P. 1994. Kheops users's guide. Technical Report 92503, Report of Laboratoire d'Automatique et d'Analyse des Systemes, Toulouse, France.

Jackson, P. 1999. *Introduction to Expert Systems*. Addison–Wesley, 3rd edition. ISBN 0-201-87686-8.

Liebowitz, J., ed. 1998. *The Handbook of Applied Expert Systems*. Boca Raton: CRC Press.

Ligęza, A. 2006. *Logical Foundations for Rule-Based Systems*. Berlin, Heidelberg: Springer-Verlag.

Nalepa, G. J., and Ligęza, A. 2004. A graphical tabular model for rule-based logic programming and verification. In Bubnicki, Z., and Grzech, A., eds., *Proceedings of 15th International Conference on Systems Science*. Wrocław: Wrocław University of Technology.

Nalepa, G. J., and Ligęza, A. 2005. A graphical tabular model for rule-based logic programming and verification. *Systems Science* 31(2):89–95.

Nalepa, G. J., and Wojnicki, I. 2007. Proposal of visual generalized rule programming model for Prolog. In Seipel, D., and et al., eds., *17th International conference on Applications of declarative programming and knowledge management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007) : Wurzburg, Germany, October 4–6, 2007 : proceedings : Technical Report 434*, 195–204. Wurzburg : Bayerische Julius-Maximilians-Universitat. Institut für Informatik: Bayerische Julius-Maximilians-Universitat Wurzburg. Institut für Informatik.

Negnevitsky, M. 2002. *Artificial Intelligence. A Guide to Intelligent Systems*. Harlow, England; London; New York: Addison-Wesley. ISBN 0-201-71159-1.

Wojnicki, I., and Nalepa, G. J. 2007. Prolog hybrid operators in the generalized rule programming model. In Seipel, D., and et al., eds., *17th International conference on Applications of declarative programming and knowledge management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007) : Wurzburg, Germany, October 4–6, 2007, Technical Report 434*, 205–214. Bayerische Julius-Maximilians-Universitat Wurzburg. Institut fur Informatik.