Decision Making in Manufacturing and Services
Vol. 5 • 2011 • No. 1–2 • pp. 79–89

Separating I/O from Application Logic for Rule-Based Control Systems

Igor Wojnicki*

Abstract. One of the main reasons of using a rule-based approach to program control systems is that they can be formally verified. For such systems communication with the environment is often encoded within the knowledge base. Such inclusion may lead to issues with portability, extendibility, maintainability, and interoperability. The paper proposes a four layer architecture to solve these issues. A proof-of-concept RBS, targeted at control systems, and an example case are also given.

Keywords: rule, rule-based, inference, i/o, control system

Mathematics Subject Classification: 68T35

Revised: 30 May 2011

1. INTRODUCTION AND MOTIVATION

There are several examples of using rule-based systems for control (G2 Gensym: `www.gensym.com`) or diagnostic (Tiger: `www.turbineserviceslimited.com`) purposes. One of the main reasons is that such systems can be formally verified (Ligęza, 2006). Such verification makes sure that the system behavior complies with its design.

A rule-based system (RBS) consists of a knowledge base and an inference engine (Negnevitsky, 2002; Liebowitz, 1998; Jackson, 1999). The knowledge base is the actual application (control) logic, while the inference engine provides general means for interpreting it. However, at some point a rule-based system needs to be integrated with underlying OS, inputs, outputs, or other applications – the environment. Such an integration is necessary in order to enable an RBS to process files, data streams, react to stimuli (esp. in case of control systems), and communicate with users through user interfaces.

Contemporary inference engines provide fairly extended means to interact with the environment. However in most cases, as it is presented below, the rule base is rarely separated from accessing and processing actual Input/Output (I/O). Such a coupling

* AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Department of Automatics, al. A. Mickiewicza 30, 30-059 Krakow, Poland, E-mail: wojnicki@agh.edu.pl

may lead to several problems including issues with portability (to adapt a rule-based application to a new environment, one has to alter its logic which might introduce bugs), maintainability, extendibility (upon altering the application logic one can unintentionally disrupt its communication with the environment) and interoperability with other software components (changing these components require changing the application logic as well – ie. switching to a different user interface).

Solving these problems could be achieved by introducing a clear separation between the application logic and its environment. It constitutes a four layer architecture which consists of: the application knowledge base, the environment knowledge base, the environment routines, and the inference engine. Such an architecture is inspired by the *Model-View-Controller* (MVC) approach (Burbeck, 1992) known from Software Engineering. The application knowledge base resembles the *Model*, the environment knowledge base is the *Controller*, while the environment routines is the *View*. In such a case the application logic (the application knowledge base) does not need to be altered in order to change its communication with the environment i.e. provide a different user interface, provide data from different sources, to port or to embed entire RBS, etc.

For the purpose of this paper the environment is defined as all the software and hardware the RBS interfaces with, including data streams, I/O devices, embedding application and underlying operating system. Terms: environment interaction and I/O are used interchangeably.

2. RULE-BASED SYSTEMS AND THEIR ENVIRONMENT

Rule-based systems offer a wide range of approaches to the problem of environment interaction. There are CLIPS, Jess, Drools and Kheops I/O facilities briefly described in this section. Some summary and analysis are also given. These particular systems were chosen to investigate a diversity of techniques used by rule-based systems designed to handle different domains (CLIPS and Kheops are applicable to create rule-based control systems, while Jess and Drools usually handle business logic).

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 program. One of the concepts used by CLIPS to provide I/O facilities is *logical names*. There are predefined *logical names* which refer to the default user input, output, error output, and warning output. 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 another 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 an 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 the C language (or any other language 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 a rule then. The data processing itself is provided in rule actions (by calling appropriate functions) or through user defined functions handling particular logical name (using the *Routers* mechanism, appropriate user defined functions are called upon referencing particular *logical name*). In both cases information about I/O data processing is kept as a part of the application logic. CLIPS can also interact with the environment through user defined functions without using *logical names* or *routers*. 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 the CLIPS API to alter knowledge-base directly from the embedding application.

To summarize. There are four methods the environment interactions can be carried out from CLIPS: file based (with use of built-in *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 enable 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. The API based approach assumes that the environment is pushing appropriate data into or reading them from the knowledge base directly through using the CLIPS API. 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*. The user defined functions concept is also provided. 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 which is adding, removing and modifying facts, rules, and other Jess objects.

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 would be and 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 call can be embedded within a rule.

3. 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.). In general, data processing and interaction with the environment can be performed through: rules, routers, user defined functions, API calls, or designated attributes.

If I/O data processing is performed by rules the inference engine has to provide appropriate processing facilities, usually functions, which can be called from within rules. 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).

Somewhere in between user defined functions and I/O processing within rules lays 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. An I/O operation might also be provided by calling an external function (or method) from within a rule as a result of its action – an user defined function.

Such an approach keeps the rule base relatively clean, free of I/O processing. 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 functions external to it. 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, within a rule, takes place. There are well defined means for I/O communication available through arbitrarily chosen attributes being input or output. All details regarding I/O operations are covered by means external to the application logic. The inference engine deals with attributes and nothing more, it does not care about how data is actually transmitted. This particular approach, in case of Kheops, 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 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 it should be minimized at least. 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. 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. This constitutes the proposed four layer architecture.

4. PROPOSED ENVIRONMENT INTERACTION

The following components are chosen to implement the four layer architecture:

- the application knowledge base is implemented with the EXtended Tabular Trees (XTT) (Nalepa, Wojnicki, 2007),
- the environment knowledge base is implemented as Input/Output Declarations (IOD),
- the environment routines are written either in Prolog or Java language,
- the inference engine is the Beating HeaRT engine, interpreting XTT based logic.

The *XTT* knowledge representation, is composed of decision tables. A single table is presented in Figure 1 (this is XTT, a refined version of the original XTT, more details can be found at hekate.ia.agh.edu.pl, XTT and XTT are used interchangeably in this paper).

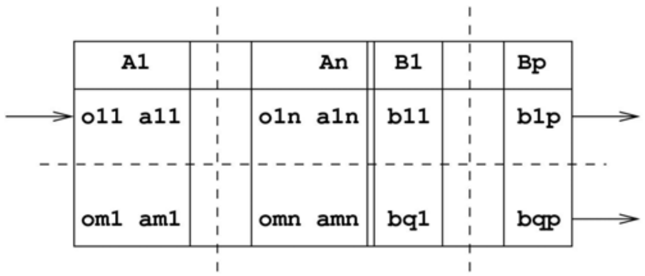


Fig. 1. *A single XTT table.*

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.

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: where is a literal in SAL (Set Attributive Logic) in a form where $o \in O$ is a object referenced in the system, and is a selected attribute of this object, is a subset of attribute domain. 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.

In order to implement the prosed four layer approach, the environment interactions are provided through designated attributes within the application knowledge base only. There is a strict declaration of inputs and outputs. Each attribute is assigned to a class: *ro* (read-only, input data), *wo* (write-only, output data), *rw* (read-write, input or output data), 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 a decision. The only way to change its value is to trigger reading 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.

Furthermore, *ro* attributes are allowed in the condition part of a rule only. Upon referencing an XTT table with 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 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 decision part of a rule. However, to provide a feedback regarding successful writing operation a wo attribute can be used in the condition part of a rule as well.

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 to 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. The returned value 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.

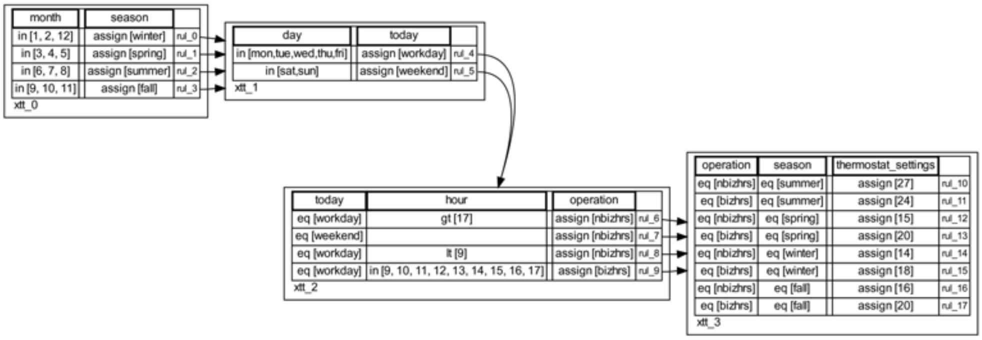
Such an approach is highly declarative. The knowledge base does not deal with details regarding interfacing with the environment, it is focused on the application logic. Furthermore it allows the inference engine to exchange attribute values with the environment during the inference process making it highly interactive.

Assigning attributes to a class (*ro*, *wo*, *rw*, *state*) is provided while declaring them. It is a part of the application knowledge base. However association between particular *ro*, *wo* and *rw* attributes and proper *trigger routines* is provided through I/O *Declarations* (IOD) implementing the environment knowledge base. Such an approach separates the application knowledge base 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* or alteration of the *routine* itself. The rule base is focused on the problem to solve, while IOD allows to establish communication with the environment.

Currently IOD is implemented as Prolog language facts with use of *io/3* predicate. The first argument of the predicate is a unique attribute identifier (name), the second is a trigger class – either *ro_trigger*, *wo_trigger*. The third one identifies the actual trigger routine by its name. The routines are implemented as Prolog predicates or Java methods.

5. EXAMPLE

The following case is a well known thermostat example (Negnevitsky, 2002; Ligęza, 2006). A simple rule-based system designed to control office temperature. XTT tables representing its logic (the application knowledge base) is given in Figure 2. This visualization of XTT diagrams is slightly enhanced comparing to the one presented in Figure 1. These enchantments are targeted toward easier XTT table and rule identification. Each of the XTT tables has a unique identifier displayed in its bottom-left corner. Furthermore, each rule within an XTT table is uniquely labeled (the last column of each XTT table). They serve identification and easier reading purposes only.



month	season	
in [1, 2, 12]	assign [winter]	rul_0
in [3, 4, 5]	assign [spring]	rul_1
in [6, 7, 8]	assign [summer]	rul_2
in [9, 10, 11]	assign [fall]	rul_3

xtt_0

day	today	
in [mon,tue,wed,thu,fri]	assign [workday]	rul_4
in [sat,sun]	assign [weekend]	rul_5

xtt_1

today	hour	operation	
eq [workday]	gt [17]	assign [nbizhrs]	rul_6
eq [weekend]		assign [nbizhrs]	rul_7
eq [workday]	lt [9]	assign [nbizhrs]	rul_8
eq [workday]	in [9, 10, 11, 12, 13, 14, 15, 16, 17]	assign [bizhrs]	rul_9

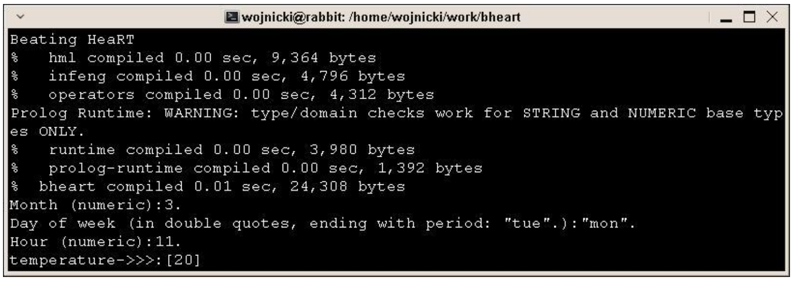
xtt_2

operation	season	thermostat_settings	
eq [nbizhrs]	eq [summer]	assign [27]	rul_10
eq [bizhrs]	eq [summer]	assign [24]	rul_11
eq [nbizhrs]	eq [spring]	assign [15]	rul_12
eq [bizhrs]	eq [spring]	assign [20]	rul_13
eq [nbizhrs]	eq [winter]	assign [14]	rul_14
eq [bizhrs]	eq [winter]	assign [18]	rul_15
eq [nbizhrs]	eq [fall]	assign [16]	rul_16
eq [bizhrs]	eq [fall]	assign [20]	rul_17

xtt_3

Fig. 2. *Thermostat Case, XTT.*

There are the following *state* attributes the thermostat application consists of: *today*, *operation*, *season*. Furthermore, there are *ro class* attributes: *hour*, *month*, *day* and a *wo class* attribute: *thermostat_settings*. The goal of the thermostat system is to set the temperature to the given set-point, based on the current time and date.



```
Beating HeaRT
%   hml compiled 0.00 sec, 9,364 bytes
%   infeng compiled 0.00 sec, 4,796 bytes
%   operators compiled 0.00 sec, 4,312 bytes
Prolog Runtime: WARNING: type/domain checks work for STRING and NUMERIC base typ
es ONLY.
%   runtime compiled 0.00 sec, 3,980 bytes
%   prolog-runtime compiled 0.00 sec, 1,392 bytes
%  bheart compiled 0.01 sec, 24,308 bytes
Month (numeric):3.
Day of week (in double quotes, ending with period: "tue".):"mon".
Hour (numeric):11.
temperature->>>:[20]
```

Fig. 3. *Thermostat Case, Prolog Interface.*

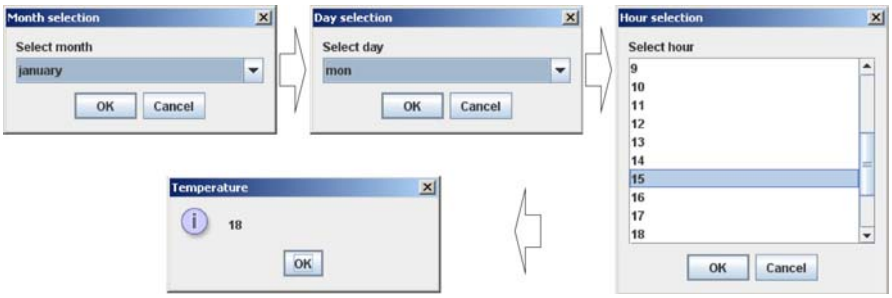


Fig. 4. *Thermostat Case, Java Interface (Leś, Łosiewicz, 2009).*

In general the *xtt_0* table computes current *season* based on *month*. In turn, the *xtt_1* table identifies if current day (*day*) is a workday or weekend day and assigns corresponding value to attribute *today*. Next, *xtt_2* identifies if there are currently business or non-business hours (attribute *operation*, values: *nbizhrs* for non-business hours and *bizhrs* for business hours) based on *hour* and *today*. Finally *xtt_3* assigns appropriate temperature to attribute *thermostat_settings* based on values of attributes: *operation* and *season*.

The inference process is started with the XTT table labeled *xtt_0*. Since *month* is an ro class attribute appropriate trigger is called to obtain its value from the environment. If the application is used in a production environment the trigger should obtain appropriate value from a real-time clock. Then the rule conditions within the XTT table are checked and the conflict set is built. The conditions use in operator which checks if the attribute value belongs to a given set. For example rul_0 condition can be read as: if $month \in 1, 2, 12$. Then the conflict set is resolved and the rules are fired by executing their decision parts. For example, if rul_0 condition is true, its decision is to assign a value of *winter* to attribute *season*. Then the inference engine switches to the XTT table pointed by the link at the fired rule, which is *xtt_1*. Subsequently, the inference process repeats for th *xtt_2* until it reaches *xtt_3* which rules have no outgoing links, then the inference process ends with success. The *xtt_3* decision part contains assignments of values to an wo class attribute which is *thermostat_settings*. For example rul_10, assigns *thermostat_settings* attribute the value of 27. After firing such rule, appropriate wo trigger is called.

The trigger assignment is provided through IOD. An example IOD which enables trigger routines written in Prolog language is given in Figure 5. It assigns attributes *day*, *month*, *hour* and *thermostat_settings* appropriate predicates written in Prolog (*get_day*, *get_hour*, *get_month*, *get_hour*, *set_temperature* respectively). For demonstration purposes these trigger routines read relevant data from standard input and write outcomes of the inference process to the standard output. The results of the inference process can be seen in Figure 3.

Another example of IOD (for the same application knowledge base) is given in Figure 6. It enables trigger routines written in Java. It allows launching appropriate methods which read in values for *day*, *month* and *hour* attributes which are: *Main.getDay, Main.getMonth, Main.getHour* respectively. Additionally there is a method for writing out *thermostat_settings* values: *Main.setTemperature*. For testing purposes these methods implement a graphical user interface which can be seen in Figure 4.

The above separate IODs can be used interchangeably. The same application, in terms of its logic (the application knowledge base), can interact with the environment through Java methods or Prolog predicates. A mixed interaction where some triggers are written in Prolog and some in Java is also possible.

```
io(day,ro_trigger,get_day).
io(month,ro_trigger,get_month).
io(hour,ro_trigger,get_hour).
io(thermostat_settings,wo_trigger, set_temperature).
```

Fig. 5. *Thermostat Case, IOD using Prolog.*

```
io(day,ro_trigger,['Main',getDay]).
io(month,ro_trigger,['Main',getMonth]).
io(hour,ro_trigger,['Main',getHour]).
io(thermostat_settings,wo_trigger,['Main',setTemperature]).
```

Fig. 6. *Thermostat Case, IOD using Java.*

6. SUMMARY AND FURTHER RESEARCH

This paper proposes a four layer architecture for rule-based systems. It allows to separate the application logic from definitions of its interactions with the environment. It introduces an approach similar to that of MVC to RBS. The concept of the proposed architecture is based on observations and analysis of contemporary technologies for developing rule-based systems such as: CLIPS, Jess, Drools and Kheops.

The presented Beating HeaRT RBS is a proof-of-concept implementing the proposed architecture. The application knowledge base is given using XTT knowledge specification. The environment knowledge base is declared with IOD, while the environment routines are implemented in Prolog or Java languages. The approach has been tested with several examples, including the thermostat case presented here. The examples acknowledge the approach to be suitable for both simple one-pass rule-based systems as well as more complex, highly interactive (with interactions during the inference process) rule-based applications.

The proposed architecture increases adaptability, extendibility and maintainability of RBS. Without altering the application knowledge base one can adapt it to a new environment or equip it with different user interface. Furthermore altering the application logic would not unintentionally disrupt its communication with the environment.

The four layer architecture can be applied to other RBS. It can be achieved through consciously splitting the knowledge base into the application knowledge base and the environment knowledge base and documenting the split properly.

Some further research focuses on formulation of a library providing *trigger routines* for variety of purposes. 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 based ones) or GUI (Graphical User Interface) widgets, as ready to use components. It would verify portability and flexibility of the proposed approach.

REFERENCES

Burbeck, S., 1992. *Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc).* Technical report, Department of Computer Science, University of Illinois, Urbana-Champaign.

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

Giarratano, J.C., Gary D. Riley, G.D., 2005. *Expert systems: principles and programming* Thomson Course Technology.

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.

Leś, W., Łosiewicz, M., 2009. *The xtt inference engine and the java virtual machine coupling.* Master's thesis, AGH University of Sience and Technology.

Liebowitz, J. (Ed.), 1998. *The Handbook of Applied Expert Systems.* CRC Press, Boca Raton.

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

Nalepa, G.J., Wojnicki, I., 2007. Proposal of generalized rule programming model. *3rd Workshop on Knowledge Engineering and Software Engineering (KESE 2007) at the 30th annual German conference on Artificial intelligence*, September 2007.

Negnevitsky, M., 2002. *Artificial Intelligence. A Guide to Intelligent Systems.* Addison-Wesley, Harlow, England; London; New York.