AGH University of Science and Technology
Computer Science Laboratory
Department of Automatics
Al. Mickiewicza 30
30-059 Kraków, POLAND

HeaRT Hybrid XTT2 Rule Engine Design and Implementation

Grzegorz J. Nalepa
`gjn@agh.edu.pl`
Szymon Bobek
`szymon.bobek@agh.edu.pl`
Michał Gawędzki
`m.gawedzki@windowslive.com`
Agata Ligęza
`jaaaga2@gmail.com`
AGH University of Science and Technology
Department of Automatics
Kraków, POLAND

Published online: 10.12.2009

CSL Technical Report No. 4/2009

CSL Technical Report No. 4/2009

HeaRT Hybrid XTT2 Rule Engine Design and Implementation[*]

Grzegorz J. Nalepa
gjn@agh.edu.pl

Szymon Bobek
szymon.bobek@agh.edu.pl

Michał Gawędzki
m.gawedzki@windowslive.com

Agata Ligęza
jaaaga2@gmail.com
AGH University of Science and Technology
Department of Automatics
Kraków, POLAND

Abstract. In this paper a new rule engine called HeaRT (HeKatE Run Time) is proposed that solves three main problems of rule-based systems design: efficient rule representation, inference control and logical verification. It uses a custom rule representation called XTT2, that is based on a formalized rule description that allows for a formalized verification of a rule-based systems. The new logic called ALSV(FD) was proposed to allow this. The XTT2 notation provides rules groupping that allows for a structural design of a rule-based systems and more flexible inference-flow control. Proposed in this paper engine is a part of a design environment that provides visual rule design capabilities. The engine supports structured rule bases and custom inference mechanisms. The rule-based logic can be integrated with the system environment using external callback functions in Java, or TCP/IP protocol. In the paper the architecture of the engine is discussed as well as selected implementation issues are given.

Keywords: xtt, rules, inference engine, heart

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

Contents

1 Introduction

Rules are one of the most successful knowledge representation methods [32]. Therefore, systems build on rules found number of applications in the fields of decision support and diagnosis; for a state-of-the art see [9, 6, 13, 8]. Recently, the Business Rules approach advocated rule applications in business software [29].

Rule engines have been the main component of classic expert systems shells, e.g. CLIPS [6]. [1] The recent growth of interest in rule-based systems stimulated the development of newer engine implementations, such as Jess [5] [2] and Drools [4]. [3] Every rule engine provides at least a rule language and inference mechanism. However, practical implementation of rule-based systems still requires solving three main problems: efficient rule representation, rule quality analysis, and system integration with the environment. For some recent studies see [17, 14]. There is no support for structural design of rule-based systems in existing tools, except for very basic implementation of rules groupping in Drools 5. No verification and no inference control for expert systems are provided in aforementioned tools.

In this paper a new rule engine called HeaRT (HeKatE Run Time) is proposed. It uses a custom rule representation called XTT2 [21], that is a second version of XTT representation described in [18]. It is based on a formalized rule description [20] that allows for a formalized analysis of rule quality, rules groupping and inference flow control. In fact, the engine is a part of a design environment [23] that provides visual rule design capabilities. The engine supports structured rule bases and custom inference mechanisms with advanced inference-flow contol. Verification tool is also provided within HeaRT that allows for a formlalized verification of rule-based system designed according to the HeaKatE metodology [21], that was developed to support more efficient rule-based system design.

The rest of the paper is organized as follows: In Sect. 2 the motivation for the HeaRT development is given. It is based on the XTT2 rule representation shortly discussed in Sect. 3. Then, in Sect. 5.1 the architecture of the solution is outlined and the implementation of the engine discussed. The evaluation of the results is given in Sect. 9.1. The paper ends with concluding remarks as well as directions for future work in Sect. 9.3.

2 Motivation

Number of implementations of rule engines for rule-based systems exist, including the classic CLIPS shell, and more recently its Java-based reimplementation – Jess, as well as Drools, tightly integrated with the JBoss application stack. They all share common roots, that results in their sharing of some old problems. The four main problems in the rule implementation include: 1) efficient rule representation and design, 2) rule quality analysis and logical verification, 4) inference control, and 3) rule-based system integration with the environment. These four challenges are still areas of active research. In fact, Drools 5 tries to deliver an integrated implementation environment, and all of the mentioned engines provide some kind of rule grouping that might simplify managing large rule bases. However, none of aformentioned tools provides mechanisms for inference control and formal verification of rule-based system, which makes these solutions not sufficient, with the problems of rule quality and effective design mostly not addressed.

The approach that resulted in the implementation of the HeaRT rule engine introduced in this paper is a result of the *Hybrid Knowledge Engineering Project* (HeKatE, see `hekate.ia.agh.edu.pl`). In this approach the XTT2 rule language, formalized with the use of the ALSV(FD) logic is proposed [20, 21]. The language introduces explicit structure of the rule base, as well as visual rule representation based on the network of decision tables. This allows for an effective visual design on a high level of abstraction, where formal verification of rules is possible. Text representation of the

[1] `clipsrules.sourceforge.net`
[2] `jessrules.com`
[3] `www.jboss.org/drools`

visual XTT2 called HMR was provided to allow easier rules processing. Tools transforming XTT2 representation to HMR and *vice versa* were also developed to improve desing process. In replay to the problem of integration rule-based systems with enwironment, two solution were presented presented in this paper including callbacks mechanism, that allows for knowledge aquisition, and TCP/IP protocol providing standard communication channel with the inference engine. Morover, several libaries in most popular programming languages were written to fasten the integration process with HeaRT.

In the next section the main aspects of the XTT2 rule representation are given, and then the design and implementation of the HeaRT engine is discussed.

3 XTT2 Rule Representation and Inference

3.1 Introduction to the ALSV(FD)

The main motivation behind the ALSV(FD) attributive logic is the extension of the notational possibilities and expressive power of the tabular rule-based systems [13, 20, 21]. Some main concepts of the logic are: attribute, atomic formulae, state representation and rule formulation.

After [13] it is assumed that an *attribute* A_i is a function (or partial function) of the form $A_i\colon O \to 2^{D_i}$. Here O is a set of objects and D_i is the domain of attribute A_i. As we consider dynamic systems, the values of attributes can change over time (or state of the system). We consider both *simple* attributes of the form $A_i\colon T \to D_i$ (i.e. taking a single value at any instant of time) and *generalized* ones of the form $A_i\colon T \to 2^{D_i}$ (i.e. taking a set of values at a time); here T denotes the time domain of discourse.

The *atomic formulae* can have the following four forms: $A_i = d$, $A_i \neq t$, $A_i \in t$, and $A_i \notin t$, where $d \in D$ is an atomic value from the domain D of the attribute. The *semantics* of $A_i = d$ is straightforward – the attribute takes a single value. The semantics of $A_i = t$ is that the attribute takes *all* the values of t (see [21]).

An important extension in ALSV(FD) over previous versions of the logic [13] consists in allowing for explicit specification of one of the relational symbols $=, \neq, \in, \subseteq, \supseteq, \sim$ and $\not\sim$ with an argument in the table.

3.2 State and Rule Representation

The system dinamically changes over the inference process. After firing rules new information may be asserted to the memory, old values may be changed. From the logical point of view the *state* is represented by the current values of all attributes specified within the contents of the knowledge-base, as a formula:

$$(A_1 = S_1) \wedge (A_2 = S_2) \wedge \ldots \wedge (A_n = S_n) \tag{1}$$

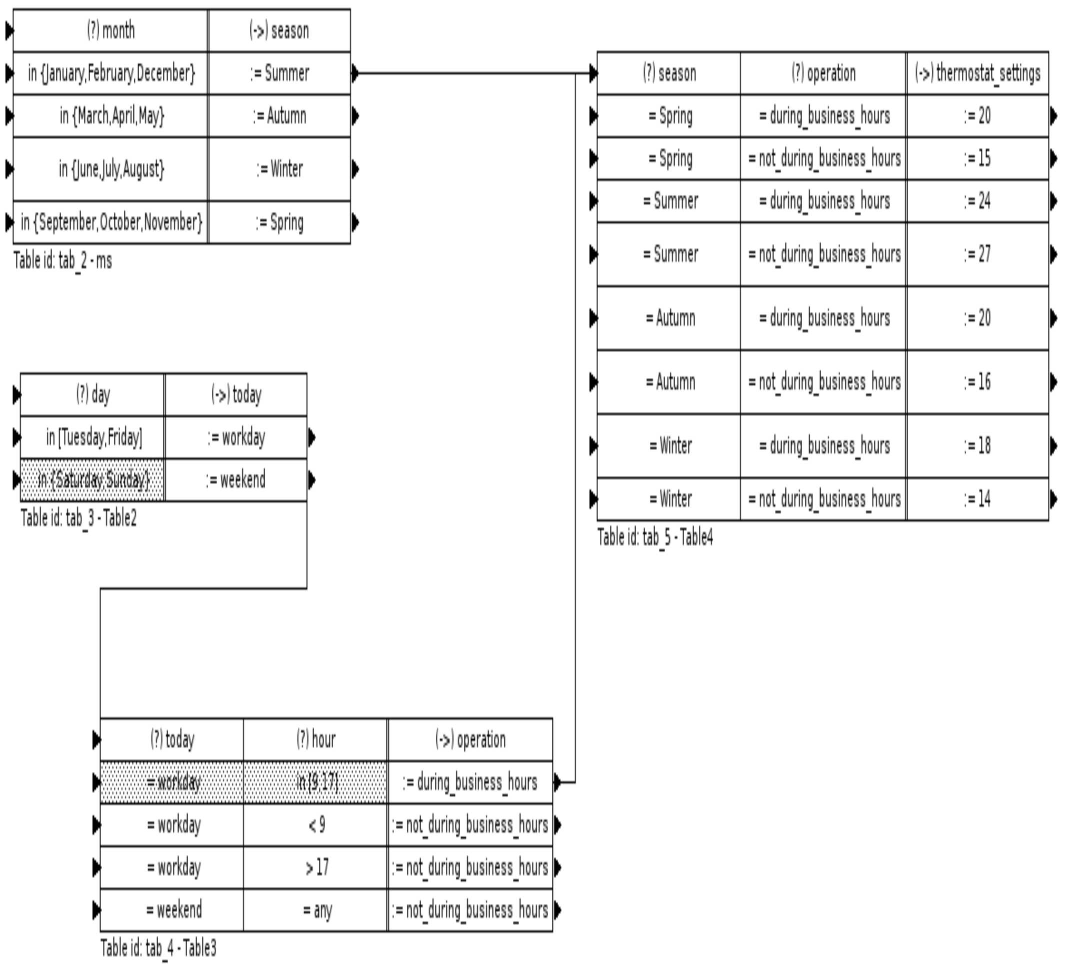
where A_i are the attributes and S_i are their current values; note that $S_i = d_i$ $(d_i \in D_i)$ for simple attributes and $S_i = V_i$, $(V_i \subseteq D_i)$ for generalized ones, where D_i is the domain for attribute A_i, $i = 1, 2, \ldots, n$. An XTT2 rule is of the form:

$$(A_1 \propto_1 V_1) \wedge (A_2 \propto_2 V_2) \wedge \ldots (A_n \propto_n V_n) \longrightarrow RHS \tag{2}$$

where $\propto_i$ is one of the admissible relational symbols, and RHS is the right-hand side of the rule covering conclusions. In practise the conclusions are restricted to assigning new attribute values, thus changing the system state. State changes trigger external callbacks that allow for communication with the environment.

3.3 Visual and Algebraic Rule Representation

XTT2 knowledge representation with incorporates extended attributive table format. Similar rules are grouped within separated tables, and the system is split into such tables linked by arrows representing the control strategy (see Fig. 1) Efficient inference is assured thanks to firing only rules necessary for achieving the goal. It is achieved by selecting the desired output tables and identifying the tables necessary to be fired first. The links representing the partial order assure that when passing from a table to another one, the latter can be fired since the former one prepares an appropriate context knowledge [21]. See Sect. 3.4.



(?) month	(->) season
in {January,February,December}	:= Summer
in {March,April,May}	:= Autumn
in {June,July,August}	:= Winter
in {September,October,November}	:= Spring

Table id: tab_2 - ms

(?) day	(->) today
in [Tuesday,Friday]	:= workday
in {Saturday,Sunday}	:= weekend

Table id: tab_3 - Table2

(?) season	(?) operation	(->) thermostat_settings
= Spring	= during_business_hours	:= 20
= Spring	= not_during_business_hours	:= 15
= Summer	= during_business_hours	:= 24
= Summer	= not_during_business_hours	:= 27
= Autumn	= during_business_hours	:= 20
= Autumn	= not_during_business_hours	:= 16
= Winter	= during_business_hours	:= 18
= Winter	= not_during_business_hours	:= 14

Table id: tab_5 - Table4

(?) today	(?) hour	(->) operation
= workday	in [9;17]	:= during_business_hours
= workday	< 9	:= not_during_business_hours
= workday	> 17	:= not_during_business_hours
= weekend	= any	:= not_during_business_hours

Table id: tab_4 - Table3

Figure 1: Example of XTT2 Visual Representation

The visual representation is automatically transformed into HMR (HeKatE Meta Representation), a corresponding textual algebraic notation, suitable for direct execution by the rule engine. An example excerpt of HMR (see Sec. 4 for details):

```
xschm th: [today,hour] ==> [operation].
xrule th/1:
  [today eq workday, hour gt 17] ==> [operation set not_bizhours].
xrule th/4:
  [today eq workday, hour  in [9 to 17]] ==> [operation set bizhours].
```

The first line defines an XTT2 table scheme, or header, defining all of the attributes used in the table. Its semantics is as follows: "the XTT table *th* has two conditional attributes: *today* and *hour* and one decision attribute: *operation*". Then two examples of rules are given. The second rule can be read as: "Rule with ID *4* in the XTT table called *th*: if value of the attribute *today* equals ($=$) value *workday* and the value of the attribute *hour* belongs to the range ($\in$) $< 9, 17 >$ then set the value of the attribute *operation* to the value *bizhours*".

3.4 XTT2 Inference Strategies

The following specification has been first discussed in [21].

Any XTT table can have one or more inputs. Let T denote a table. By $I(T)$ we shall denote the number of input links to table T. If there are k such input links, they will be denoted as $1.T, 2.T, \ldots, k.T$.

Note that all the links are in fact considered as an AND connection. In order to fire table T all the input tables from which the input links come must be fired to provide the necessary information to fire T.

A similar consideration corresponds to the output. Any table can have one or more output links (from different rules which are placed at the rows of the table), and such an output link can be directed to one or more tables. If there are m such output links, we shall denote them as $T.1, T.2, \ldots, T.m$.

If an output link $T.j$ goes to n tables $T1, T2, \ldots, Tn$, then the links can be denoted as $T.j \rightarrow T1, T.j \rightarrow T2, \ldots, T.j \rightarrow Tn$. In fact, we can have $\Sigma_{j=1}^{m} dim(j)$, where $dim(j)$ is the number of addressed tables (here n).

The XTT tables to which no connections point are referred to as *input tables*. The XTT tables with no connections pointing to other tables are referred to as *output tables*. All the other tables (ones having both input and output links) are referred to as *middle tables*.

Now, consider a network of tables connected according to the following principles:

- there is one or more input table,
- there is one or more output table,
- there is zero or more middle tables,
- all the tables are interconnected.

The problem is how to choose the inference order. The basic principle is that before firing a table, all the immediately preceding tables must have already been fired. The structure of the network imposes a partial order with respect to the order of table firing. Below, we describe three possible algorithms for inference control.

3.4.1 Fixed-Order Approach

The simplest algorithm consists of a hard-coded order of inference, in such way that every table is assigned an integer number; all the numbers are different from one another. The tables are fired in order from the lowest number to the highest one.

In order to ensure executability of the inference process, the assignment of such numbers should fulfill the following minimal requirement: for any table T, the numbers assigned to tables being predecessors of T must all be lower than the one assigned to T.

After starting the inference process, the predefined order of inference is followed. The inference stops after firing the last table. In case a table contains a complete set of rules (w.r.t. possible outputs generated by preceding tables) the inference process should end with all the output values defined by all the output tables being produced.

3.4.2 Token-Transfer Approach

This approach is based on monitoring the partial order of inference defined by the network structure with tokens assigned to tables. A table can be fired only when there is a token at each input. Intuitively, a token at the input is a kind of a flag signalling that the necessary data generated by the preceding table is ready for use.

The tables ready to be fired (with all tokens at the input) are placed in a FIFO queue. The outline of this algorithm is as follows:

- since input tables have 0 inputs, they automatically have all the tokens they need.
- all the input tables are placed in a FIFO queue (in an arbitrary order),
- then the following procedure is repeated; the first table from the queue is fired and removed from the queue, the token is removed from its input and placed at the active output link and passed to all following tables. Simultaneously, if a token is passed to a table, the table is immediately checked if it has tokens at all the inputs; if so, it is appended to the end of the queue,
- the process stops when the queue is empty (no further tables to fire).

Note that this model of inference execution covers the case of possible loops in the network. For example, if there is a loop and a table should be fired several times in turn, the token is passed from its output to its input, and it is analyzed if can be fired; if so, it is placed in the queue.

3.4.3 Goal-Driven Approach

The presented models of inference control can be considered to be *blind* procedures since they do not take into consideration the goal of inference. Hence, it may happen that numerous tables are fired without purpose – the results they produce are of no interest. This, in fact, is a deficiency of most of the forward-chaining rule-based inference control strategies.

A *goal-driven approach* works backwards with respect to *selecting* the tables necessary for a specific task, and then fires the tables forwards so as to achieve the goal. The principles of backward search procedures are:

- one or more output tables are identified as the ones that can generate the desired goal values: these are the tables that must be fired,
- these tables are stored on a stack (LIFO queue) in an arbitrary order,
- the following procedure is repeated: the list of tables from the queue is examined and all the input tables are identified; they are placed on the stack, while the analyzed table is marked as "needed" and removed from queue,
- only unvisited tables are examined,
- for input tables no analysis is necessary; all the input tables necessary to start the process are identified.

The execution is performed forwards using the token-transfer approach. Tables which are not marked as "needed" on the stack are not fired – they are not necessary to achieve the goal.

4 HeKatE Meta Representation

4.1 Introduction

The visual XTT2 model is represented by means of a human readable, algebraic notation, also called the XTT presentation syntax, or HeKatE Meta Representation (HMR).

The representation can be directly run by the HeKatE RunTime environment HeaRT. HMR file is in fact a legal Prolog code that can be interpreted directly (number of custom operators are defined). Therefore, HeaRT prototype is implemented in Prolog and provides the implementation for number of inference solutions including the three described before.

HMR can be used to write rules directly, or it can be generated by the HQEd XTT2 editor from the visual XTT2 model. The same rule-based knowledge can be serialized to XML-based HML.

An example excerpt of HMR and its interpretation in pseudocode is given below:

```
xschm th: [today,hour] ==> [operation].

xrule th/1:
  [today eq workday,
   hour gt 17]
  ==>
  [operation set not_bizhours].

xrule th/4:
  [today eq workday,
   hour  in [9 to 17]]
  ==>
  [operation set bizhours].
```

The pseudocode equivalent for the HMR notation:

```
CREATE SCHEMA th THAT USES today AND hour AND PRODUCES operation;
RULE 1 IN SCHEMA th :
  IF today == workday AND hour > 17 THEN operation = not_bizhours
RULE 4 IN SCHEMA th :
  IF today == workday AND hour IN <9 ; 17> THEN operation = not_bizhours
```

The first line defines an XTT table scheme, or header, defining all of the attributes used in the table.

Its semantics is as follows: The XTT table th has two conditional attributes: today and hour and one decision attribute: operation. This information is determined from the conceptual design phase using the ARD method. Then two examples of rules are given.

The second rule can be read as: Rule with ID 4 in the XTT table called th: if value of the attribute today equals (=) value workday and the value of the attribute hour belongs to the range (in) <9,17> then set the value of the attribute operation to the value *bizhours*.

4.2 HMR language specification

HMR (Hekate Meta represenation) language was created to provide simple and human-readable text representation for XTT diagrams that would be also easily parsed by the machine. Graphical representation is very human-readable, but it is very difficult to parse it by the machine. The popular XML notation can be easily parsed by the machine, but it is hardly readable by human. To address this two issues, the HMR language was design to be a simple and very readable text representation, that is also fully Prolog compatible, which means that it is interpreted directly by Prolog interpreter, without any additional parser. The HMR file consists of the following elements:

- Mandatory types definitions (predicate *xtype*),

- Optional types groups definitions (predicate *xtpgr*),
- Mandatory attributes definitions (predicate *xattr*),
- Optional attributes groups definitions (predicate *xatgr*),
- Mandatory schemas definitions (predicate *xschm*),
- Mandatory rules definitions (predicate *xrule*),
- Optional states definitions (predicate *xstat*),
- Optional callbacks definitions (predicate *xcall*),
- Optional actions definitions (predicate *xactn*),
- Generated by verification plugin predicates containing verifications reports (predicate *xhalv*),
- Generated by inference engine predicates containing information about system states changes during inference process (predicate *xtraj*).

All HMR elements are described below. Following notation was introduced:

- All mandatory fields are preceded by **+** sign,
- All optional fields are preceded by **?** sign,
- If there is more than one possible value of the field following notation will be used {*value1* | *value2* | *value3*}; the notation means that the filed can take either value1, value2, or value3,
- `%STRING%` means a string in Prolog,
- `%NUMBER%` means either integer or floating point number,
- `%LIST%` means a list in Prolog,
- `%CLAUSE%` means any valid Prolog goal,

4.2.1 Types definitions

Types in HMR language are used to define kind, structure and domain for further attributes definitions. Type definition looks as follows:

```
xtype [ +name: %STRING%,
  +base: {numeric | symbolic},
  +domain: %LIST%,
  ?scale: %NUMBER%,
  ?ordered: {yes | no},
  ?desc: %STRING%].
```

- *name* – a field that contains type's name,
- *base* – a field describing base of the type. It can be either numeric, or symbolic,
- *domain* – a filed that contains a list of all acceptable values for this type. For ordered symbolic domains, it is possible to assign weights to the values. For instance in the example below, every day has it's own number assigned, that allows referring to the values using this number instead of symbolic name. If a symbolic domain is marked as ordered, and there are no weights assigned explicitly to the domain's values, default numbering is assumed that starts from 1 and ends on *n*, where *n* is the number of elements in the domain,

- *scale* – a field denoting precision for floating point numbers,
- *ordered* – a field indicating whether domain is ordered or not. Every numeric type has ordered domain by default,
- *desc* – a filed containing longer description of the type.

Type definition may look as follows:

```
xtype [ name: days,
  domain: [mon/1,tue/3,wed/5,thu/7,fri/9,sat/10,sun/20],
  ordered: yes,
  desc: 'Days of the week'].
```

4.2.2 Attributes definitions

Attributes in HMR language denotes instances of defined types. Every attribute in the HMR language must have a type assigned. Only one type can be assigned to one attribute. Type definitions looks as follows:

```
xattr [ +name : %STRING%,
  +class: {simple | general},
  +type: %XTYPE_NAME%,
  +comm: {in | out | inter | comm},
  ?callback: %LIST%,
  ?abbrev: %STRING%,
  ?desc: %STRING%].
```

- *name* – a field contains attribute's name,
- *call* – a field that denotes weather the attribute is simple (represents simple values) or general (represents set values),
- *type* – a field that contains a name of a type defined as *xtype*. This field denotes the attribute's type,
- *comm* – a field that describes attribute's relation to the world. If the attribute is *in* it means that a value of it is supposed to be given by the user; if the attribute is out it means that the value of it should be presented to the user; if attribute is inter it means that its value is set by the system as a result of inference process, but it;s not relevant to the user; if the attribute is comm, it means that it's both in and out,
- *callback* – a field that contains an information about which callback is supposed to be fired for the attribute. The list should look as follows:

  ```
  [callback_name, [callback_parameters]]
  ```

  The `callback_name` is a name of defined callback, and `callback_parameters` is a list of parameters that the callback takes,
- *abbrev* – a field that contains short name of the attribute,
- *desc* – a field that contains longer description of the attribute.

Atribute definition may look as follows:

```
xattr [name: day,
  abbrev: day,
  class: simple,
  type: days,
  comm: in,
  callback: [ask_console,[day]]].
```

4.2.3 Types and attributes groups definitions

Groups contain sets of attributes' or types' names. It is possible to refer to a group of attributes or types using group name instead of each type's or attributes name separately. Groups were designed for future use in HeaRT interpreter.

Groups definitions looks as follows:

```
xtpgr [ +name : %STRING%,
  ?abbrev: %STRING%,
  +types: %LIST%
  ].

xatgr [ +name : %STRING%,
  ?abbrev: %STRING%,
  +types: %LIST%
  ].
```

4.2.4 Schemas definitions

Schemas represent XTT tables in HMR language. However, they cannot be considered as entities filled with rows containing rules. Schema is only a representation of relation between attributes that rules falling into this schema realizes. Definitions of schema looks as follows:

```
xschm +%STRING1%/?%STRING2% : +%LIST1% ==> +%LIST2%.
```

- `%STRING1%` – a field containing obligatory schema name,
- `%STRING2%` – a field containing an optional schema description,
- `+%LIST1% ==> +%LIST2%` – represents relation that given schema describes. First list denotes attributes that are required to be known by the rules that falls in this schema, and the second list denoted attributes that values are calculate by the rules in given schema. For instance: `[month] ==> [season].`.

An example of schema definitions may look as follows:

```
 xschm ms: [month] ==> [season].
```

4.2.5 Rules definitions

Rules in HMR language are defined as follows:

```
xrule %STRING1%/%NUMBER%: %LIST1%
==> %LIST2% **> %LIST3% : %STRING2%.
```

- `%STRING1%` – a field that contains name of the schema that the rule falls into,
- `%NUMBER%` – a field that represents rule ID inside the schema,
- `%LIST1%` – contains a list of firing conditions for the rule. Each condition has to be valid ALSV(FD) expression of the form `attribute_name ALSV_OPERATOR value`,
- `%LIST2%` – contains decisions that has to be fired when conditions are true. Decision is of the form `attribute_name set value`,
- `%LIST3%` – contains list of actions with its parameters of the form `[action_name, [action_parameters]]`. Those actions are fired when rule's conditions are true. Actions cannot change system's state,

- `%STRING2%` – is a schema (table) name that should be given a token when inference mode is set to Token-Driven.

An example of a rule definition is shown below.

```
xrule dt/1:
  [day in [mon,tue,wed,thu,fri]]
  ==>
  [today set workday]
  **>
  [tell_console,['Today is a workday']]
  :th.
```

4.2.6 States definitions

State in HMR language is a set of attributes and corresponding values that describes current status of the syste. State definitions looks as follows:

```
xstat %STRING%: %LIST%.
```

- `%STRING%` – is a name of the state. Because each *xstat* record describes only one attribute's value, the name is not unique. If a system contains *n* attributes, that have to be described by HMR state, there have to be *n xstat* records – each for every attribute.
- `%LIST%` – is always a two-elements list containing attribute's name and value.

An example of *xstat* record may look as follows:

```
xstat s1: [month, january]
```

4.2.7 Callbacks definitions

Callback is an operation that is fired at the certain point of inference process, and can change system state. Callbacks are defined as follows:

```
xcall %STRING% : %LIST% >>> (%CLAUSE%).
```

- `%STRING%` – is a callback name,
- `%LIST%` – is a list of variables that have to be unified with some values in order to reach a goal defined by `%CLAUSE%`,
- `%CLAUSE%` – is any Prolog code describing a goal.

An example of *xcall* record is shown below:

```
xcall ask_console : [AttName] >>> (
  write('Type value for attribute:  '),
  write(AttName),nl,
  read(Answer),
  alsv_new_val(AttName,Answer)).
```

4.2.8 Actions definitions

Action is an operation that is fired when rule's firing conditions are true. Actions cannot change system state.

```
xactn %STRING% : %LIST% >>> (%CLAUSE%).
```

- `%STRING%` – is a action name,
- `%LIST%` – is a list of variables that have to be unified with some values in order to reach a goal defined by
- `%CLAUSE%` – is any Prolog code describing a goal.

An example of *xactn* record is shown below:

```
xactn tell_console : [AttName] >>> (
    write('Attribute '),
    write(AttName),
    write(' output value is '),
    xstat(current:[AttName,V]),
    write(V),nl).
```

4.2.9 Verification reports

Predicates containing versification reports are generated automatically by the HeaRT interpreter. Every record contains only one message generated by verification plugin. All messages generated during verification are grouped within a set of *xhalv/1* predicates of the same ID (name). Definition of *xhalv/1* looks as follows:

```
xhalv %STRING% : %LIST%.
```

- `%STRING%` – is an unique ID (or name of a set of xhalv records) that is automatically generated every time a verification process is fired.
- `%LIST%` – is a message generated by the verification process. The list can look differently depending on which verification was performed:
  - subsumption: [rule_id, description], where *rule_id* is a rule that subsumes other rule; a *description* is text message describing in details anomaly that was detected.
  - contradiction: [rule_id, description], where *rule_id* is a rule that contradicts with other rule; a *description* is text message describing in details anomaly that was detected.
  - reduction: [rule_id, description], where *rule_id* is a rule that with other rule; a *description* is text message describing in details anomaly that was detected.
  - completeness: [description], where *description* is a text message describing in details anomaly that was detected.

4.2.10 Trajectory traces

Predicates containing trajectories of the system's inference process are automatically generated by the HeaRT and saved as *xtraj/1*. A single *xtraj/1* record contains information about one inference process from the start until the end. Definitions of *xtraj/1* are build as follows:

```
xtraj %STRING% : %LIST%.
```

- `%STRING%` – is an unique ID that is automatically generated every time a inference process is fired.
- `%LIST%` – is a list of two-elements list of the form:
  - `[entity_id, state_id]` – where *entity_id* is a name of entity after which a state of the system was capture (it can be name of a table, name of a rule). Trajectory contains as a first element state of the system before inference process; entity for this state is called *start*. The *state_id* is a name of a *xstat/1* record where the state of the system was saved.

5 HeaRT Architecture and Implementation

5.1 Engine Architecture

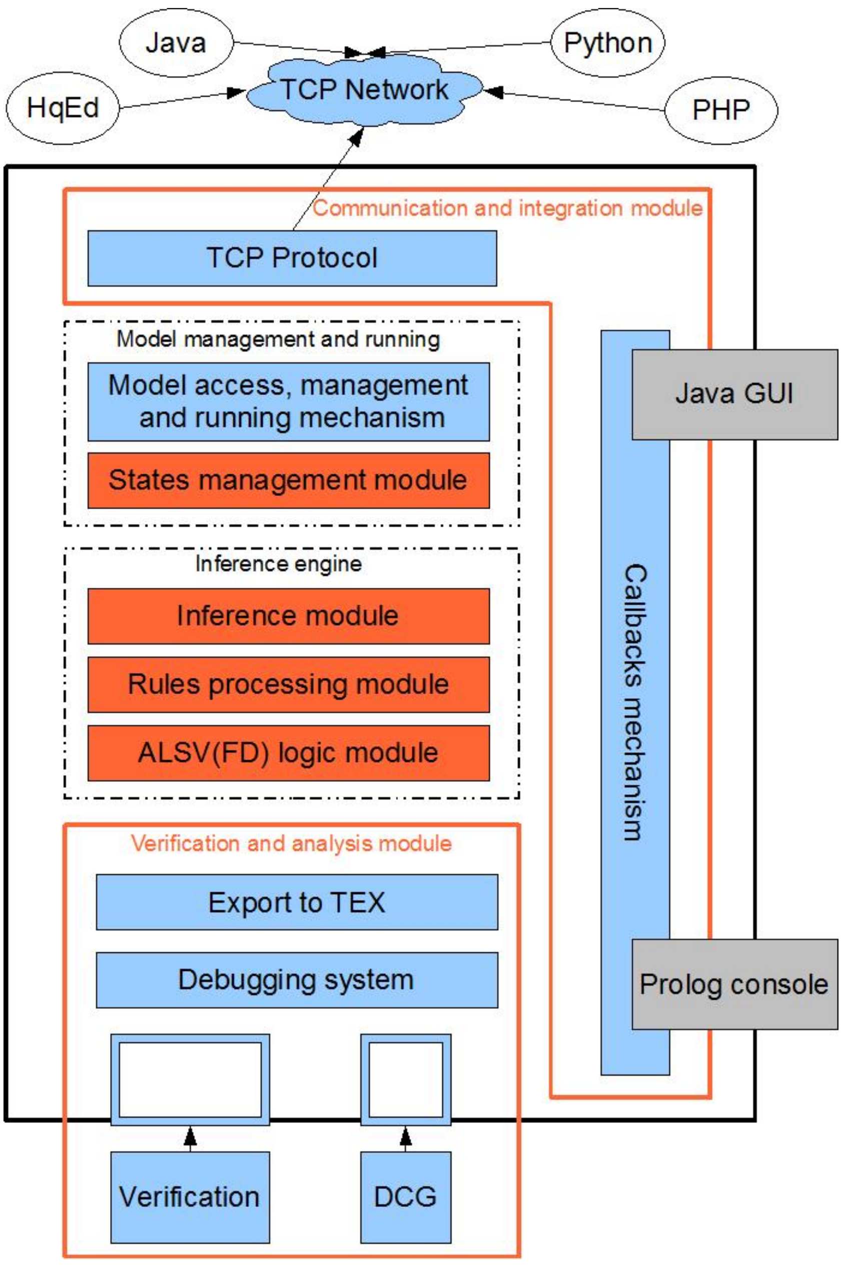


Figure 2: HeaRT Architecture (see [2])

HeKatE RunTime (HeaRT) is a dedicated inference engine for the XTT2 rule base, The architecture of HeaRT can be observed in Fig. 2 The engine is highly modularized. It is composed of the main *inference module* based on ALSV(FD). It supports four types of inference process, Data and Goal Driven, Fixed Order, and Token Driven [21]. The *model management module* allows loading and managing rule models. HeaRT also provides a *verification module*, also known as HalVA (HeKatE Verification and Analysis) [15]. The module implements: simple debugging mechanism that allows tracking system trajectory, logical verification of models (several plugins are available, including completeness, determinism and redundancy checks), and syntactic analysis of HMR files using a DCG grammar of HMR. The verification plugins can be run from the interpreter or indirectly from the design environment using the communication module. The *communication and integration* *module* provides environment integration based on the callbacks mechanism. Another feature is the network-based communication protocol, that allows for both remote access as well as integration with other components of the design environment [23].

5.2 HeaRT Implementation

The engine is implemented in Prolog, using the SWI Prolog stack. The main HMR parser is heavily based on the Prolog operator redefinition [3]. A dedicated forward and backward chaining meta interpreter is provided, implementing custom rule inference modes.

Implementation of modularized architecture (shown on Figure 2) was achieved by dividing HeaRT source code into smaller, partially independent pieces – modules. The following modules were extracted:

- *Inference module* that includes 293 lines of code was implemented in a *heart-infer.pl* file. It is responsible for running models in four offered by the system modes: Data-Driven, Goal-Driven, Token-Driven and Fixed order Driven. Implementaiton of the inference modes was described in details in 5.3 section.
- *ALSV(FD) logic module* that includes 948 lines of code, was implemented in a *heart-alsv.pl* file. It is responsible for evaluating rules conditions according to the ALSV(FD) logic described in details in [22]. The ALSV(FD) logic module implementation was discussed in details in 5.5 section.
- *Communication and integration module* that includes 616 lines of code was implemented in *heart-io.pl* file. It is responsible for providing TCP/IP communication mechanism. The Protocol was described in 5.7 section.
- *Rules processing* that includes 203 lines of code was implemented in a *heart-sules.pl* file. It is responsible for selecting rules that firing conditions are true, and then firing them.
- *States management module* that includes 128 lines of code was implemented in a *heart-state.pl* file provides mechaninsm for printing, adding and removing system states; it also implements mechanism for tracking states of the system during inference process.
- *Verification and analysis module* that includes about 1000 lines of code was implemented as a plugin to HeaRT. It is located in a *varification* directory, and the main file that offers all the verification predicates is called *verification.pl*. More about verification module can be found in 6 section. Within this module couple more tools allowing offline analysis of the system was implemented. That includes export XTT models to TEX, and saving trajectory to file; both located in *heart-io.pl* file, and discussed in details in 5.6.3 section.

5.3 Inference modes

HeaRT implements inference strategies as specified in Sect. 3.4. Implementation of these strategies is located in *heart-infer.pl* file. Every inference mode, except for the Fixed-Order mode) works according to the following schema:

Step 1 Based on the connections between tables determine an order in which tables are supposed to be processed.

Step 2 If Token-Driven mode was selected, determine how many tokens each table one the stack will require to be processed.

Step 3 Process rules in tables that were put on the stack.

The main task of this module is to build a stack of tables to be run by the *rules processing module*. Depending on the inference mode, different predicates are used to build the stack. The inference is controlled by HeaRT differently for each inference mode. See Fig. 3 for examples. [4]

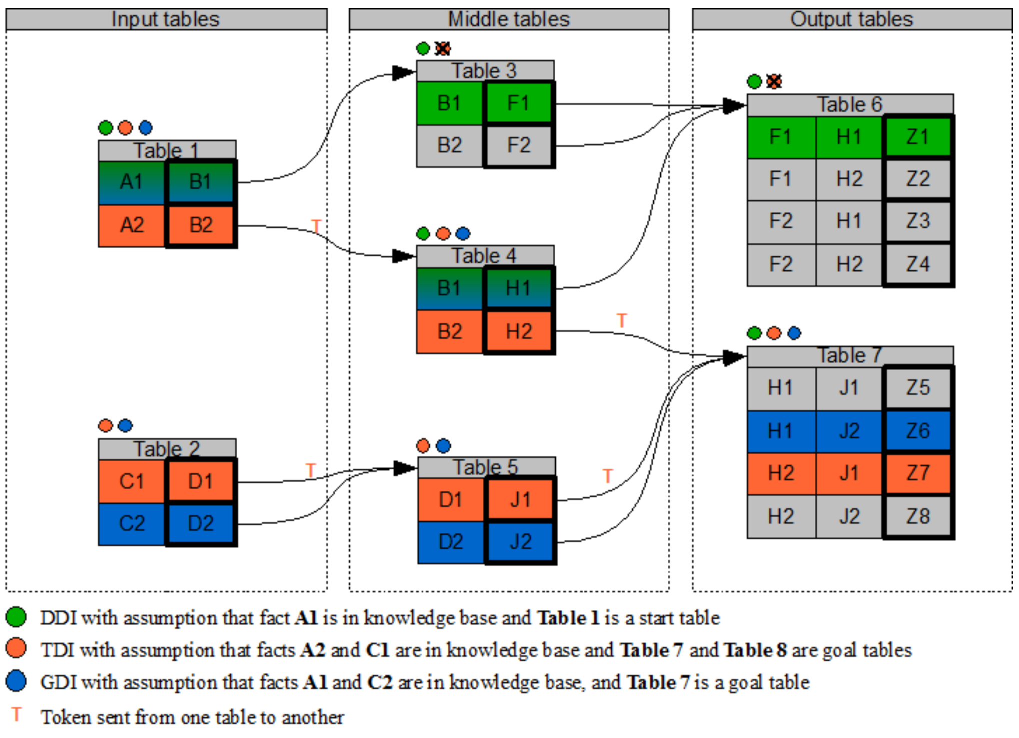

Input tables
Middle tables
Output tables
Table 1
A1
B1
A2
B2
Table 2
C1
D1
C2
D2
Table 3
B1
F1
B2
F2
Table 4
B1
H1
B2
H2
Table 5
D1
J1
D2
J2
Table 6
F1
H1
Z1
F1
H2
Z2
F2
H1
Z3
F2
H2
Z4
Table 7
H1
J1
Z5
H1
J2
Z6
H2
J1
Z7
H2
J2
Z8
T
DDI with assumption that fact A1 is in knowledge base and Table 1 is a start table
TDI with assumption that facts A2 and C1 are in knowledge base and Table 7 and Table 8 are goal tables
GDI with assumption that facts A1 and C2 are in knowledge base, and Table 7 is a goal table
Token sent from one table to another

Figure 3: HeaRT inference modes

The Fixed-Order mode does not require building a stack since it is the user who determines the order, and number of tables to be fired. The list passed by the user is directly run by *rules processing module*. The predicate that runs this kind of inference is called `runtables(+Tables)`.

The Data-Driven mode requires one or more start tables from which the inference should begin. If no tables are specified, all tables without predecessors are treated as start tables. Based on this tables, a stack of successor tables is created, and then processed. The orderer of the tables in the list is not relevant. Successor table is designated as the one that requires attribute value that the other table (the predecessor table) produces. When the stack is created, all tables within the stack are run by the `runtables` predicate.

```
runrules_DDI(Init) :-
   create_ddi_stack(OrderedTables,Init),
   runtables(OrderedTables).

create_ddi_stack([],[]).
create_ddi_stack(Stack,[H|Goals]) :-
   create_ddi_stack_helper(StackPart1, H),
   create_ddi_stack(StackPart2,Goals),
   join_tables(StackPart1,StackPart2,StackDuplicated),
   reverse(StackDuplicated,RStackDuplicated),
   list_to_set(RStackDuplicated,RStack),
   reverse(RStack,Stack),!.
```

The Goal-Driven mode is very similar to data-Driven, but the stack is built backward. The Goal-Driven mode needs at least one goal table, which is supposed to be the final table to fire in the

[4]Circles above tables means that table was put on the stack. Crossed circles means that despite being on the stack, table was not processed.

inference process. If no tables are specified, all tables without successors are treated as finals. Based on the goal table (or tables) a stack of predecessor tables is created and the processed. Predecessor table is designated as the one that produces attribute value that the other table (the successor table) requires. When the stack is created, all tables within the stack are run by the `runtables` predicate.

```
runrules_GDI(Goals) :-
   create_stack(OrderedTables, Goals),
   runtables(OrderedTables).

create_stack([],[]).
create_stack(Stack,[HG|RG]) :-
   create_stack_helper(FrontStack, [HG]),
   create_stack(RareStack,RG),
   join_tables(FrontStack,RareStack,StackDuplicated),
   list_to_set(StackDuplicated,Stack),!.
```

The Token-Driven mode is the most complicated one. It also needs one ore more final tables, which are the final tables to be fired in the inference process. Based on these tables a stack of predecessor tables is created, then required numbers of tokens for each table is calculated and the stack of tables is processed. The order of the tables in the list is not relevant. Predecessor table designation is based on the links between tables. When the stack is created, the tokens for all tables are set to zero, and they are fired by the `run_token_inference/0` predicate.

```
runrules_TDI(Goals):-
   retractall(xtoken(_,_,_)),
   create_token_stack(Stack,Goals),
   fill_token_knowledge(Stack),
   run_token_inference(Stack).
```

No matter which inference mode is chosen, the rules within a XTT table are always executed by the `runrules(+TableName)` predicate, which uses the *rules processing* module.

```
runrules(T):-
   xrule(T/R:C==>D),
   heart_debug(1,['Firing rule: ',T/R]),
   cond_satisfy(C),
   process_decision(D,Go),
   inference_transfer(Go),
   trajectory_projection(T/R,full),
   fail.
runrules(_).
```

5.4 Rules processing

The *rules processing* module implemented in *heart-rules.pl* file is responsible for selecting rules that firing conditions are true, and then firing them.

The following predicates were implemented to support this functionality:

- *formula_satisfy(+Conditions)* – a predicate that checks if given as a parametr condition part of a rule is true.
- *cond_satisfy(+Condition)* – a predicate used by the `formula_satisfy/1` to determine if a single element of a condition expression is true.
- *proces_decision(+Decisions)* – a predicate that processes decision part of a rule.
- *process_action(+Actions)* – a predicate that processes action part of a rule.

- *inference_transfer(+TableName)* – the predicate used by the Token-Driven inference that grants specified as a parameter table a token.
- *xtable(?Name,?Desc,?From,?To)* – a predicate used to extract information about XTT tables present in the model.

5.5 ALSV(FD) logic implementation

The ALSV(FD) logic was presented in details in [22]. This module implements all ALSV(FD) logic operators and provides a large set of predicated that allows validation of attributes values within states and rules conditions and decisions parts. This module is also responsible for evaluating and executing transitions within decision part of the XTT rules.

To allow proper work of all mentioned predicates also for general attributes, a set of special predicates was implemented. Those predicates allow compute intersection, difference, union and complement of a sets defined in the HMR notation.

For evaluating conditions written in ALSV(FD) logic the `alsv_valid(+Expr,+StateVal)` predicate is used.

```
alsv_valid(Att in [L to U], State) :-
   alsv_attr_class(Att,simple),
   alsv_values_get(State,Att,StateValue),
   !,
   alsv_values_check(Att,[StateValue]), alsv_values_check(Att,[L]),
   alsv_values_check(Att,[U]),
   normalize(Att,[StateValue],[NormStateValue]),
   normalize(Att,[L],[NormL]),
   normalize(Att,[U],[NormU]),
   NormStateValue =< NormU,
   NormStateValue >= NormL,
   heart_debug(2,['Valid:',Att,in,[L to U]]).
```

For checking if attributes values used in condition and decision part of the rule falls into a domain of the attribute, defined in HMR file, the `alsv_values_check(+AttName, +AttValue)` predicate is used.

```
alsv_values_check(Att,[LS to US|Rest]) :-
   alsv_domain(Att,Domain,numeric),
   heart_debug(2,['Check:',Att,numericreg,LS to US]),
   math_member([LS to US],Domain),
   alsv_values_check(Att,Rest),
   !.
```

For evaluating a decision part of the rule the `alsv_transition(+AttName, +AttVal)` predicate is used.

```
alsv_transition(Att set Expr) :-
   \+ Expr = [_|_],
   alsv_domain(Att,_,numeric),
   alsv_attr_class(Att,simple),
   alsv_eval_simple(Expr, Val),
   heart_debug(2,['Adding new fact: ',Att,' set to ',Val]),
   alsv_new_val(Att,Val).
```

5.6 HeaRT tracking system

There are two ways of tracking HeaRT operations:

- using the built-in information system that depending on a value of the `debug_flag/1` predicate produces to the Prolog console information about system activity.

- using trajectory projection mechanism that depending on a value of the `trajectory_mode/1` predicate saves system states over the time when inference process takes place.

5.6.1 HeaRT debug mechanism

The HeaRT debug mechanism is a simple information system that reports system current activity *on-the-fly*. The details of information printed to the Prolog console depends on a value of the `debug_flag/1`. The value may vary from 0 up to 3, where 0 is the very quite mode, and 3 is noisy one, printing almost every information about system work.

The value of the `debug_flag` can be changed with `debug_set_value/1` predicate.

5.6.2 Trajectory projection mechanism

States management module implemented in *heart-state.pl* file provides mechanism for printing, adding and removing system states; it also implements mechanism for tracking states of the system during inference process (the trajectory of the system). Every time the inference process is run its trajectory is saved. There are three different modes in which the trajectory projection predicate can work:

- *full* – All information including which rules in which tables were fired, is saved. State of the system is saved after every rule is fired.
- *table* – Information about the tables fired are saved. State of the system is saved after every table is processed.
- *simple* – System state at the beginning and at the end of the inference process is saved.

The mode of trajectory projection system can be changed with `trajectory_set_mode/1` predicate.

Since HeaRT saves trajectory after each inference process, every trajectory has its own unique ID, generated by the `traj_genid(-ID)` predicate. The ID of the latest trajectory is stored within a dynamic `traj_id/1` predicate.

5.6.3 Offline analysis

HeaRT provides mechanisms for offline analysis of the models and system work that includes exporting models to TEX and saving system trajectory to file.

The `io_export_TEX(+Filename)` predicate located in *heart-io.pl* file allows to export the current model to TEX file. Every XTT table is presented as a table formated in TEX notation, and all rules' conditions are translated from HMR notation to more common mathematical one.

The `io_export_trajectory(+Filename, +TrajID)` exports trajectory of the given ID to the given file.

5.7 HeaRT Communication Protocol

To make HeaRT more flexible tool, and to introduce another integration level, TCP/IP protocol was developed to allow remote access to inference engine. In addition to this several libraries in most popular languages were written to support easier access to this feature.

The protocol part of communication module was designed to allow many clients to work with one HeaRT instance acting as an inference server. It means that several different models owned by several different clients can be stored in HeaRT memory. Models are swaped in HeaRT memory transparently to the client.

To allow this feature work properly, following functionality was implemented:

- listing all models that are store in HeaRT memory,

- getting a model specified by modelname and username,
- adding a new model to HeaRT memory,
- deleting a model specified by modelname and username,
- running inference process in one of the four modes, ddi, foi, gdi or tdi for specified state,
- adding a new state to specified model,
- removing a state from the specified model,
- checking if a given modelname that belongs to username is present in HeaRT storage area,
- running verification process in one of four modes, complete, contradict, subsume and reduce.

Following commands that implements aforedmentioned functionality are available:

5.7.1 Protocol commands

- **[hello,+client_name]** – welcome message that client sends to HeaRT in order to obtain information about version of the protocol implemented on the server, and functionality offered by the protocol.
  As a result the client gets `[true,[heart,hello,1.0,5]]`

- **[model, getlist]** – returns list of all models (with users' names that own them) that are store in HeaRT storage area.
  As a result of the the client gets following answer

  ```
  [true,
  [[username1,modelname1],
  [username1,modelname2],
  [username2, modelname3]]]
  ```

  If there is no models in the storage area following answer is returned: `[true,[]]`. If there are errors during reading a model list, following message is returned:

  `[false,'Error occurred while reading a model list']`

- **[model, get, +modelname, +username]** – returns a model specified by modelname and username.
  As a result client gets the following answer: `[true,'MODEL']` where MODEL is the requested model.
  If there is no requested model or user, follwing answer is sent:

  `[false,'Model or username does not exist.']`

- **[model,exists,+modelname, +username]** – checks if a given modelname that belongs to username is present in HeaRT storage area. If it is stored by the interpreter, following answer is sent to the client: `[true,true]` If there is no such file `[true,false]` is sent back. If there is an error during this procedure, following error message is sent to the client:

  `[false,'Error while checking file existence']`

- **[model,add, +modelname, +username, +'MODEL']** – adds new model to HeaRT storage area, overriding any existing models of the same name.
  If adding succeed, `[true]` is returned. In case of an error following error message is sent to the client: `[false,'Error while saving model.']`

- **[model,remove, +modelname, +username]** – removes a model specified by modelname and username. After successful deletion fo the model `[true]` is returned. In case of an error, following error message is sent back to the client
  `[false,'Error while deleting model.']`

- **[model,run, +modelname, +username, +ddi | gdi | tdi | foi, +tables, +{statename | statedef}]** – runs inference process for the specified by specified model in one of the four modes, ddi, gdi, tdi or foi for given tables (schemas) – (see [[gox/3]] for details). State from which the inference should start can be either a name of the state that is located in HMR file, or a full definition of the state. The full definition of the state (statedef) should look as follows: `[[attribute_name1, value],[attribute_name2, value] ... ]`. If the inference process succeed, the following answer is sent back to the client:
  `[true,system_state, system_trajectory]` The *system_state* is final state of the system defined as statedef. The system trajectory is a list of rules that were fired in the form of `[schema_name,rule_id,schema_name, rule_id, ...]`. If there is an error during this procedure, the following error message will be sent back to the user:
  `[false,'Error occurred while running model']`

- **[state, add ,+modelname, +username, +statename, +statedef]** – adds new state to specified model. The statedef is full state definition in a form of:
  `[[attribute_name1, value],[attribute_name2, value] ... ]`. The statename is a Prolog string.

- **[state, remove, +modelname, +username, +statename]** – remove a state from specified model. If the operation succeeds following message is sent back to the client: `[true]`. In case of an error, the following message is sent back:
  `[false,'Error while deleting model.']`

- **[model, verify, +vcomplete | vcontradict | vsubsume | vreduce, +modelname, +username, +schema]** – runs verification process in one of four modes, complete, contradict, subsume and reduce. If the verification succeeds the following message is sent back to the client: `[ [report],[report], ... ]`. Where *report* is one [[xhalv/1]] record. The list contains all reports generated by the verification process.
  In case of an error, following message is sent back to the client:
  `[false,'Error while verifying']`

Character **+** means, that this parameter is obligatory. Format like this *+{ddi | gdi | tdi | foi}* means, that you must specify one of these parameters.

All protocol commands have to be valid Prolog lists followed by a dot. If the command that is is sent to the server, is not a list followed by a dot, following command will be sent back to the client: `[false,'Timeout. Bad or incomplete command.']`. If a valid Prolog list is sent to the server, but it is not recognized by it, following message will be sent back to the user:`[false, 'Command not supported.']`

5.7.2 Protocol examples

- Adding a model

```
 [model,add,'MyName','Username',
'xtype [ name: week_days,
base: symbolic,
ordered: yes,
domain: [moday,tuesday]',
desc: \'This is only one definition\'].'].
```

- Getting a model `[model,get,'MyName','Username'].` As a result following message should be returned:

  ```
  [true,
  [xtype [ name:week_days,
  base:symbolic,
  ordered:yes,
  domain:[moday, tuesday]',
  desc:'This is only one definition'].]]
  ```

- Getting a list of all models stored by the server: `[model,getlist].` Following answer can be returned:

  ```
  [true,
  [[PHP, hqedmodel0], [PHP, temostat],
  [PHP, Termostat2], [PHP, therm-rt],
  [Username, MyName], [XTTd, Termostat1]]]
  ```

- Deleting a model: `[model,remove, 'MyName','Username'].`
- Running a model `[model,run,'themostat','PHP',ddi, [ms,dt], s1].`This command will run a Data-Driven inference from state *s1* on the model *themostat* belonging to *PHP* user. Following return message can be sent by the server:

  ```
  [true,
  [[day, 3.0], [hour, 3.0], [month, [2.0]],
  [season, 3.0], [today, 1.0], [operation, 2.0],
  [thermostat_settings, 16.0]], [ms, 4, dt, 1, th, 3, os, 7]]
  ```

  Instead of state name, clinet can send a full state definition over the protocol:

  ```
  [model,run,'MyName','Username',
  ddi, [ms,dt,th,os],
  [ [day,monday], [hour,13], [month, january] ] ].
  ```

- Verifying a model `[model,verify, vreduce, temostat,'PHP',os].` Following answer was sent by the server:

  ```
  [true, [[4, 'In os rule: 4 can be joined with rule 8'],
  [8, 'In os rule: 8 can be joined with rule 4']]]
  ```

5.7.3 Protocol access libraries

To make HeaRT integration easier, there are three integration libraries, JHeroic, PHeroic or YHeroic. For full implementation of there libraries see appendix A.

JHeroic library was written in Java. Based on JHeroic one can build applets, desktop application or even JSP services. It is also possible to integrate HeaRT with database using ODBC, or Hibernate. Following interface implementation was developed in Java.

```
public interface JHeroicInterface
{
   public static String protocolVersion = "1.0";

   public ArrayList<JHModel> getModelList() throws Exception;
   public String getUserModel(String userName, String modelName)
```

```
                 throws Exception;
   public String addUserModel(String userName, String modelName,
                 String model) throws Exception;
   public String removeUserModel(String userName, String modelName)
                 throws Exception;
   public String runInference(String userName, String modelName,
                 String infType, ArrayList<String> tables, String state)
                 throws Exception;
   public String addStateToModel(String userName, String modelName,
                 String stateName, String stateDef) throws Exception;
   public String removeStateFromModel(String userName, String modelName,
                 String stateName) throws Exception;
   public String getProtocolVersion();
   public String verifyModel(String userName, String modelName, int mode)
                 throws Exception;
}
```

YHeroic is a library created in Python. It has the same functionality as JHeroic but is easier to use because of Python language nature. Following set of methods were developed in Python.

```
class YHeroic:
   def __init__(self, hostname, port):
   def getModelList(self):
   def getUserModel(self, userName, modelName):
   def addUserModel(self, userName, modelName, model):
   def removeUserModel(self, userName, modelName):
   def runSimulation(self, userName, modelName, infType, tables, state):
   def addStateToModel(self, userName, modelName, stateName, stateDef):
   def removeStateFromModel(self, userName, modelName, stateName):
   def getProtocolVersion(self):
   def verifyModel(self, userName, modelName, mode):
```

PHeroic is the same library but created in PHP5. It can be used in a dynamic web page based on PHP. Following class was written in PHP.

```
class PHeroic
{
   public function __construct($hostname, $port)
   public function getModelList()
   public function getUserModel($userName,  $modelName)
   public function addUserModel($userName, $modelName, $model)
   public function removeUserModel($userName, $modelName)
   public function runInference($userName, $modelName,
                             $infType, $tables, $state)
   public function addStateToModel($userName, $modelName,
                             $stateName, $stateDef)
   public function removeStateFromModel($userName, $modelName,
                             $stateName)
   public function getProtocolVersion()
   public function verifyModel($userName, $modelName, $mode)
}
```

5.8 Callbacks Framework

HeaRT supports Java integration based on the callbacks and the SWI Prolog JPL library. Callbacks can be used to create GUI with JPL and SWING in Java. Another option is to use the SWI XPCE GUI.

The callback mechanism is a part of HMR language discussed in Section 4.

To help integrating Java with HeaRT a simple callbacks library was implemented that supports basic data exchange between HeaRT and the user. Serveral different callbacks were written each dedicated to different data types (that includes general, symbolic, and numeric attributes).

Examples of callback functions are as follows:

```
xcall ask_symbolic_GUI : [AttName] >>>
   (jpl_new('skeleton.RequestInterface',[],T),
   alsv_domain(AttName,Domain,symbolic),
   Params = [AttName|Domain],
   term_to_atom(Params, AtomParams),
   jpl_call(T,request,['callbacks.input.ComboBoxFetcher',
     AtomParams],Answer),
   atom_to_term(Answer,Answer2,_),
   alsv_new_val(AttName,Answer2)).

xcall xpce_ask_numeric: [AttName] >>>
   (alsv_domain(AttName,[Min to Max],numeric),
   dynamic(end),
   new(@dialog,dialog('Select a value')),
   send_list(@dialog,append,
     [new(I,int_item(AttName,low := Min, high := Max)),
     new(_,button('Select',and(message(@prolog,assert,end),
       and(message(@prolog,alsv_new_val,AttName,I?selection),
       message(@dialog,destroy)))))]),
   send(@dialog,open),
   repeat,
     send(@display,dispatch),
   end,!,
   retractall(end)).
```

The first uses the SWI Prolog JPL library, that dynamically links the Prolog code with Java classes. The second is a simple Prolog XPCE solution.

Examples of executing a callback are shown on Fig. 4 and 5.

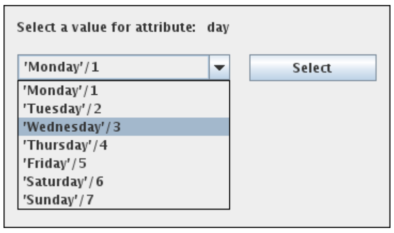


Figure 4: Example of drop-down GUI for callback

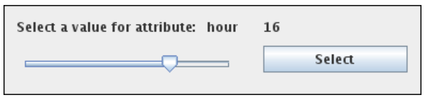


Figure 5: Example of slider GUI for callback

From the network access point of view HeaRT can operate in two modes, stand-alone and as a TCP/IP server, offering network integration mechanism. In particular, it allows for integration with a complete rule design and verification environment [23]. It also makes the creation of console or graphical user interface built on the Model-View-Controler design pattern possible.

6 HalVA Verification Framework

6.1 Formal Analysis of XTT Rules

The quality of a rule-based system is dependent on the quality of a knowledge base. What is more important, anomalies in the set of rules could be a result of serious faults in system responses. Therefore the analysis of knowledge base is significant step during developing rule-based system.

The issue of verification and validation were discussed by many authors. Differences in their approaches to V&V start at the definition level. To unify them, in this paper verification and validation processes are defined as follows:

Verification is a process in the early design phase, aimed at checking if the system meets its constraints and requirements ([31, 1, 26, 28, 30]).

Testing is a process aimed at analysing the system work, by comparing system responses to known responses for special input data ([31]).

Validation is a case of testing, aimed at checking if the system meets user requirements ([31]).

A summary result of analysis techniques and tools is presented in [33].

The classification of potential errors and deformation of knowledge base was also widely discussed. From the formal point of view anomalies of knowledge base could be divided into three main categories: 1) *incompleteness*, 2) *indeterminism*, and 3) *overdeveloped set of rules*.

Let the knowledge base be described by rules:

$$\begin{array}{llcl} r_1\colon & \Psi_1 & \rightarrow & h_1 \\ r_2\colon & \Psi_2 & \rightarrow & h_2 \\ & \vdots & & \\ r_n\colon & \Psi_n & \rightarrow & h_n \end{array} \tag{3}$$

Completeness assures that for any input state the system reacts and produces some response (conclusion, decision or action) ([10]). In other words, the system with the set of rules (3) is *logically complete* if a disjunction of preconditions is a tautology:

$$\models \Psi_1 \ \vee \ \Psi_2 \ \vee \ \ldots \ \vee \ \Psi_n$$

The knowledge base is incomplete, when in the set of rules exist *unreachable clauses*, *dead-end clauses* or some rules are *missing*.

The unreachable clause exists if rule:

$$r\colon \ P(x) \rightarrow Q(x)$$

can be found in in the set (3), and $Q(x)$ is not the system response and do not satisfy preconditions of any other rule.

In (3) a formula $\Psi'_1, \Psi'_2, \ldots, \Psi'_n$ is missing, if

$$\models \Psi_1 \vee \Psi_2 \vee \ldots \vee \Psi_m \vee \Psi'_1 \vee \Psi'_2 \vee \ldots \vee \Psi'_n$$

but a formula:

$$(\Psi_1 \vee \Psi_2 \vee \ldots \vee \Psi_n) \wedge (\Psi'_1 \vee \Psi'_2 \vee \ldots \vee \Psi'_m)$$

is never satisfied.

Determinism guarantees that the system always produces the same reaction for the same input data. In other words for any input state the system finds a unique solution ([10]). From the formal point of view the set (3) is indeterministic "if there exists a state described by formula ϕ, such that simultaneously $\phi \models \Psi_1$ and $\phi \models \Psi_2$ and $h_1 \neq h_2$" ([11]).

The system is indeteministic, if there are *contradictory rules* in knowledge base. Rules r_i and r_j are contradicted, if there exists a state ϕ, such that $\phi \models \Psi_i$ and $\phi \models \Psi_j$, but under the considered interpretation $I \not\models_I h_i \wedge h_j$.

Inconsistency also is a cause of indeterminism. "Two rules r_1 and r_2 are inconsitent if there exists a state described by formula ϕ, such that simultaneously $\phi \models \Psi_1$ and $\phi \models \Psi_2$, but $\not\models h_1 \wedge h_2$" ([11]).

Minimal number of rules indicates a set of rules without *redundant*, *subsumed* rules. What is more, the set of rules should produce the same reactions as a overdeveloped set.

The formal classification of reduncancy is presented in [26]. The redundancy in rule chain is the most general form of redundancy. In this case two rules r_i and r_j are redndant, if

$$\begin{array}{ll} r_i\colon & P(x) \rightarrow Q(x) \rightarrow R(x) \\ r_j\colon & P(x) \rightarrow R(x) \end{array}$$

and $Q(x)$ is not satisfied in any other rule.

All of above features – completeness, determinism, minimal number of rules – should be provided to assure reliability, safety and efficiency of the rule-base system ([10]).

In systems built on XTT^2 rules ([18, 21]), verification and validation procedures are simplified. The design process of the system is based on top-down methodology ([19, 22]). Therefore, every step in the process can be analysed. The model can be built incrementally. This solution allows to provide a *incremental verification* and – in a consequence – save computational resources.

The XTT^2 representation introduces a structurization of knowledge base and simplify pointing out *contexts*. Implicitly the context is identified with an extended decision table. An isolation of contexts allows to provide *local analysis* – contexts can be verified separately.

6.2 HalVA Implementation

The analysis of the XTT knowledge base is provided by HalVA Verification Framework. The main purpose of the framework is the local verification.

HalVA consists of Prolog-based predicates. The verification framework was developed as a plug-in of the HeaRT inference engine [23]. HalVA provides the verification of completeness, contradiction and subsumption. What is more, the number of rules can be reduced. In order that the verification is focused on a local level, the schema of the XTT table is considered.

All of the verification procedures presented bellow are based on inference rules for ALSV(FD) introduced in [22]. For state described by ϕ a precondition $(A_i \propto_i V_i)$ (where $\propto_i \in \{=, \neq, \in, \notin\}$ for the simple attribute and $\propto_i \in \{=, \neq, \subseteq, \supseteq, \sim, \not\sim\}$ for general attribute, $i = 1, \ldots, n$) is satisfied, if simultaneously:

- V_i is a value from a domain of A_i,
- ϕ_{A_i} is a value from a domain of A_i, where ϕ_{A_i} is a value of attribute A_i in formula ϕ,
- a clause $(A_i = \phi_{A_i})$ is a logical consequence of a clause $(A_i \propto_i V_i)$.

In practise, those inference rules are implemented in the HeaRT engine by a predicate `alsv_valid/2`. The predicate has a complex form. Therefore, in this paper only base clauses are presented.

For simple attributes one of the clauses has form:

```
alsv_valid(Att in [L to U], State) :-
  alsv_attr_class(Att,simple),
  xstat State: [Att,StateValue],
  !,
  alsv_values_check(Att,
                    [StateValue]),
  alsv_values_check(Att,[L]),
  alsv_values_check(Att,[U]),
  normalize(Att,[StateValue],
            [NormStateValue]),
  normalize(Att,[L],[NormL]),
  normalize(Att,[U],[NormU]),
  NormStateValue =< NormU,
  NormStateValue >= NormL.
```

This clause is adequate for a condition $(A \in V)$, where A is a simple attribute and V is a value set. In this case the value set V is introduced by lower and upper bounds. It means, the domain of the attribute A is ordered.

An example of clause for general attributes is:

```
alsv_valid(Att sim Set,State) :-
  alsv_attr_class(Att,general),
  xstat State: [Att,StateValue],
  !,
  alsv_values_check(Att,StateValue),
  alsv_values_check(Att,Set),
  normalize(Att,StateValue,
            NormStateValue),
  normalize(Att,Set,NormSet),
  intersection(NormStateValue,
               NormSet,[_|_]).
```

The clause corresponds to preconditions $(A \sim V)$, where A is a general attribute and V is a value set. All clauses for predicate `alsv_valid` are included in HeaRT.

The basic idea of the verification of *completeness* is to check if all possible values of attributes and their combinations are covered by rules conditions in verified XTT table. Domains of attributes are taken into consideration. The Cartesian Product of domains determine all states for the context (table). For every tuple, corresponding to the input state, the algorithm checks, if preconditions of any rule are satisfied. If there is no rule to execute, the considered state is reported as uncovered. Based on all uncovered states, a proposal of a new rule is introduced.

The analysis ends, when all states are checked. Domains of attributes are finite. Therefore, the verification procedure terminates after a finite number of discrete steps.

Verification of *contradiction* is based on a pairwise comparison of rules. Two rules, executable in the same time (for the same state), are taken into consideration. The comparison concerns the right-hand side of rules. If conclusions are inconsistent, the conflict is reported. The verification procedure stops when all possible comparisons are done.

The pairwise comparison of rules is also used to verify *subsumption*. However, the analysis concerns both sides of rules. One rule is subsumed by another, if its preconditions are more specific, but simultaneously conclusions are more general. The verification procedure finds two rules in the context (table), executable for the same state. Then checks, whether there is relation between conclusions.

The algorithm provides all comparisons. This strategy allows to detect identical rules. In this case, a rule is reported as subsuming another and the other subsuming the first one.

HalVA allows to reduce an overdeveloped set of rules. The reduction can be done by using the dual resolution ([11]). If rules produce the same conclusions and in the precondition part exists at least one the same clause, the rest of the clauses are joined into one formula. All possible reductions are reported. What is more, proposals of new rules are introduced.

7 HeaRT Use Scenarios

7.1 HeaRT Shell

The basic way of working with HeaRT is to use the Prolog console. A set of predicates was written to allow running inference process, launching verification, exporting models to TEX, and saving trajectories to files.

7.1.1 Running inference process

It i possible to run inference process in one of four modes:

- Fixed order (FOI),
- Data driven (DDI),
- Goal driven (GDI),
- Token driven (TDI).

All of these modes can be run:

- from a given state, (*gox/3* predicate),
- with assumption that all *in* attributes have callbacks written that will be fired before inference process to collect all required data (*goxio/2* predicate).

To run inference from a given state the **gox(+State, +Tables, +Mode)** predicate was provided. Parameters to the *gox/3* predicates are defined as follows:

1. **State** – is a state name that was defined in HMR file from which the inference suppose to start.
2. **Tables** – is a list of tables names (schemas). A meaning of this parameter depends on th inference mode selected.
    - For *foi* the list denotes tables that have to be processed. The order of the tables is relevant.
    - For *ddi* the list denotes *start* tables from which the inference should begin. Based on this tables, a stack of successor tables is created, and then processed. The orderer of the tables in the list is not relevant. Successor table is designated as the one that requires attribute's value that the other table (the predecessor table) produces. For instance if *table2* will be successor for *table1* if:

      ```
      xschm table1 : [A] ==> [B].
      xschm table2 : [B] ==> [C].
      ```

    - For *gdi* the list denotes goal tables. Based on these tables a stack of predecessor tables is created and the processed. The order of the tables in the list is not relevant. Predecessor table is designated as the one that produces attribute's value that the other table (the successor table) requires. For instance if *table1* will be predecessor for *table2* if

      ```
      xschm table1 : [A] ==> [B].
      xschm table2 : [B] ==> [C].
      ```

    - For *tdi* the list denotes goal tables. Based on these tables a stack of predecessor tables is created, required numbers of tokens calculated and the stack of tables is processed. The order of the tables in the list is not relevant. Predecessor table is designation is based on the links between tables.

3. **Mode** can take one of the following values:
   - *foi* – for Fixed ordered mode
   - *ddi* – for Data driven mode
   - *gdi* – for Goal driven mode
   - *tdi* – for Token driven mode

To run an inference process from a state read by the callback mechanism *goxio(+Tables, +Mode)* predicate was provided. The predicate qorks exactly like a *gox/3*, but instead of taking values of attributes from a given state, callbacks for *in* attributes are fired and data is acquired form the user.

7.1.2 Saving trajectory to a file

Trajectory is the system inference path. After every inference process invocation new trajectory is generated. The trajectory can be identified by ID, to obtain ID of recently generated trajectory use *traj_id(ID)* predicate. The ID variable will be unified with an ID of the last trajectory. To save trajectory projection to a file use *io_export_trajectory(+Filename, +TrajID)* predicate. To save most recent trajectory following predicates has to be called from a Prolog console:

```
traj_id(ID),io_export_trajectory('file.txt', ID).
```

An example file generated by the *io_export_trajectory/2* predicate:

```
TRAJECTROY: Rule start has generated state:
[[day, Tuesday], [hour, 15], [month, February]]
TRAJECTROY: Rule Table2/1 has generated state:
[[day, Tuesday], [hour, 15], [month, February], [today, workday]]
TRAJECTROY: Rule ms/1 has generated state:
[[day, Tuesday], [hour, 15], [month, February], [today, workday],
[season, Summer]]
TRAJECTROY: Rule Table3/1 has generated state:
[[day, Tuesday], [hour, 15], [month, February], [today, workday],
[season, Summer], [operation, during_business_hours]]
TRAJECTROY: Rule Table4/3 has generated state:
[[day, Tuesday], [hour, 15], [month, February], [today, workday],
[season, Summer], [operation, during_business_hours],
[thermostat_settings, 24]]
```

7.1.3 Exporting models to TEX

Every XTT model can be exported to TEX. The *io_export_TEX(+Filename)* predicate creates a TEX file containing all rules organized into tables that coresponds to the XTT tables.

On tables 7.1.3, 7.1.3, and 7.1.3 a sample output for the *io_export_TEX(+Filename)* predicate was presented.

7.1.4 Verifying models

A verification plugin provides following predicates to allow verifications of XTT models.

vsubsume(+Schema)

Checks currently loaded model for subsumtions. Versification reports are printed out to the Prolog's console respectively to the vreport flag that can be set by the *set_vreport_flag/1*. The higher the flag, the more detailed information are printed out.

All information about verification process are saved in the *xhalv/1* predicate not matter what is a value of the vreport flag. Last performed verification report can be obtained by the *get_current_vreport/1* predicate. ID (or name) of the *xhlav/1* record can be retrieved using *vid/1* fact.

Schema is a name of a table (schema) for which the verification is supposed to be performed.

vcomplete(+Schema)

Works exactly the same way as *vsubsume/1*, but checks the model for completeness.

vcontrdict(+Schema)

Works exactly the same way as *vsubsume/1*, but checks the model for contradictions within the given table.

vreduce(+Schema)

Works exactly the same way as *vsubsume/1*, but checks if there are rules that can be reduced to one rule.

7.2 HeaRT Batch Mode

To run HeaRT from a Linux console run a bash script HeaRT located in heart directory. The script can take 4 parameters:

1. HML file to consult.

   Usage: *heart somefile.pl*

   After that, SWI-Prolog will be run, and file given as a parameter will be consulted.

2. Tables to be fired.

   Usage: *heart somefile.pl [table1,table2,table3]*

   After that SWI-Prolog will consult somefile.pl file, and then fire table1, table2 and table3 in DDI mode. If there were callbacks written they also will be fired.

3. Inference mode.

   Usage: *heart somefile.pl [table3] gdi*

   After that SWI-Prolog will consult somefile.pl file, and run GDI inference, treating table3 as a goal table.

4. Initial state.

   Usage: *heart somefile.pl [table3] gdi state1*

   After that SWI-Prolog will run inference in GDI mode treating table3 as a goal table and taking attributes values from state1.

7.3 TCP/IP Integration

TCP/IP protocol developed for HeaRT allows to integrate any application with the inference engine. Three libraries described in Section 5.7.3 can be used to make the integration process easy.

JHeroic

To integrate external application written in Java, teh JHeroic library can be used by simply adding a JHeroic JAR file to the Java classpath.

```
java -cp JHeroic.jar:. TheProgram
```

To create an object of the class that omplements JHeroic interfece, following line should be added to the source code:

```
JHeroic JHeroicInstance = new JHeroic(0,"heart.domain.com",20044");
```

The first argument is a debug flag which indicates if debug information should be displayed. Zero means no debug messages.

YHeroci

To integrate HeaRT with a program written in Python, the YHeroic library can be ued. To include this library to it you must import it

```
import string, sys, YHeroic
```

To create a new instance of YHeroic object following line should be called.

```
myLib = YHeroic.YHeroic('heart.domain.com',20044)
```

PHeroic

To integrate HeaRT with an application or a dynamic webpage written in PHP, a PHeroic library should be included to the project:

```
include_once('PHeroic.php');
```

To create a PHeroic object that will allow to connect and use HeaRT inference engine following line should be added to a PHP code:

```
$pheroic = new PHeroic("heart.domain.com",20044);
```

7.4 HQEd Integration

Two different methods of integration HeaRT and HQEd editor can be defined: *passive*, and *active* one.

In the passive method there is no direct interference between these two tools. HQEd editor can generate HMR file, which is an input for HeaRT. The file can be then parsed by HeaRT and all operations described in 7.1 can be performed from HeaRT shell.

The other method of integration is more active, and allows running and verifying models from within HQEd editor. Ir requires HeaRT to be running in server mode. Communication between the editor and the inference engine is done via TCP/IP Protocol described in Section 5.7.

The example of how to run a XTT model from HQEd was described in Section 5.2.

Table 1: Types definitions

Name	Base	Domain	Scale	Ordered	Description
default	symbolic	$\{none/1\}$	*empty*	yes	*empty*
weekdays	symbolic	$\{'Monday'/1,'Tuesday'/2,$ $'Wednesday'/3,'Thursday'/4,$ $'Friday'/5,'Saturday'/6,'Sunday'/7\}$	*empty*	yes	A set of days of the week
hours	numeric	$\langle 0;23\rangle$	0	*empty*	Hours of a day
months	symbolic	$\{'January'/1,'February'/2,$ $'March'/3,'April'/4,$ $'May'/5,'June'/6, \quad 'July'/7,'August'/8,$ $'September'/9,'October'/10,$ $'November'/11,'December'/12\}$	*empty*	yes	All months of the year
seasons	symbolic	$\{'Winter'/1,'Spring'/2,$ $'Summer'/3,'Autumn'/4\}$	*empty*	yes	Seasons of the year
operating_hours	symbolic	$\{not\_during\_business\_hours/0,$ $during\_business\_hours/1\}$	*empty*	yes	Office operating hours
week_or_end	symbolic	$\{workday, weekend\}$	*empty*	*empty*	Weekend or weekday
thermostat_temperature	numeric	$\langle 1;30\rangle$	0	*empty*	The temperature range of the thermostat

Table 2: Attributes definitions

Name	Calss	Type	Comm	Callback	Abbreviation	Description
day	simple	$weekdays$	in	$[ask\_symbolic\_GUI, [day]]$	day	The current day
$hour$	simple	$hours$	in	$[ask\_numeric\_GUI, [hour]]$	$hour$	The current hour
$month$	simple	$months$	in	$[ask\_symbolic\_GUI, [month]]$	mth	The current month
$operation$	simple	$operating\_hours$	inter	$empty$	$oper$	Operating hours
$season$	simple	$seasons$	inter	$empty$	ssn	Current season
$thermostat\_settings$	simple	$thermostat\_temperature$	out	$[tell\_console, [thermostat\_settings]]$	$therm$	The thermostat setting
$today$	simple	$week\_or\_end$	inter	$empty$	$today$	Part of the week

Table 3: Rules definitions for Table4

Rules ID	Conditions	Decisions	Actions	Links
1	$season = Spring \land operation = during\_business\_hours$	$thermostat\_settings = 20$	$empty$	$'empty'$
2	$season = Spring \land operation = not\_during\_business\_hours$	$thermostat\_settings = 15$	$empty$	$'empty'$
3	$season = Summer \land operation = during\_business\_hours$	$thermostat\_settings = 24$	$empty$	$'empty'$
4	$season = Summer \land operation = not\_during\_business\_hours$	$thermostat\_settings = 27$	$empty$	$'empty'$
5	$season = Autumn \land operation = during\_business\_hours$	$thermostat\_settings = 20$	$empty$	$'empty'$
6	$season = Autumn \land operation = not\_during\_business\_hours$	$thermostat\_settings = 16$	$empty$	$'empty'$
7	$season = Winter \land operation = during\_business\_hours$	$thermostat\_settings = 18$	$empty$	$'empty'$
8	$season = Winter \land operation = not\_during\_business\_hours$	$thermostat\_settings = 14$	$empty$	$'empty'$

8 XTT2 Rule Base System Example

The system is a simple temperature controller. The basic idea is to set the temperature in an office based on time and working hours. The case has been originally taken from [27].

8.1 Conceptual Design

The conceptual design stage is the beginning of the formalized hierarchical process in HeKatE metodology. Based on the concepts, or vocabulary, the ARD [12, 25, 24] properties and attributes are being identified, as well as functional dependencies between them stated. ARD covers requirements specification stage. Its input is a general systems description in the natural language. Its output is a model capturing knowledge about relationships among attributes describing system properties. The model is subsequently used in the next design stage, where the actual logical design with rules is carried out.

On Figure 6 and 7 the output from the conceptual deign stage was presented.

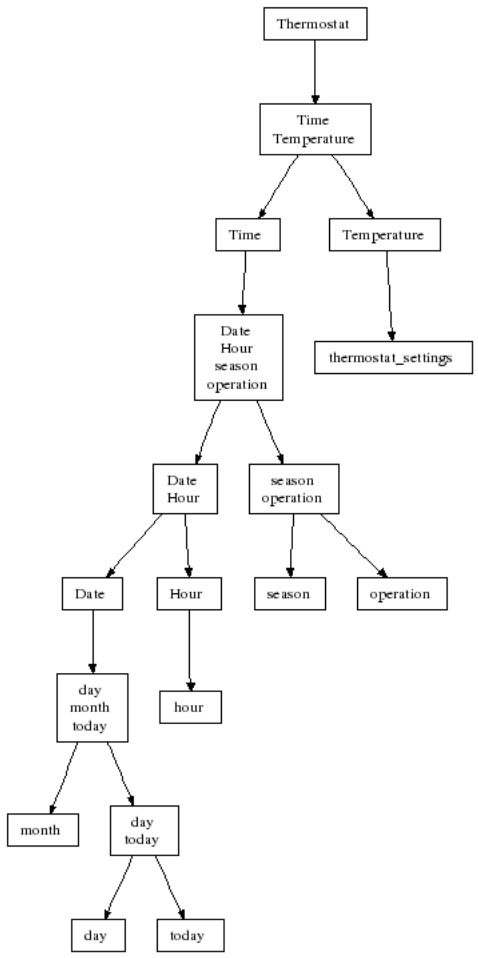


Figure 6: Example of Transformation Process History (TPH) file generated by Varda editor

8.2 Logical Design

In this stage the actual design of the XTT tables is put forward. Tables are filled with rules, and extra, fine-grained links can be added.

The design of the XTT tables can be done manually – by editing a HMR file, or with a HQEd editor. The output of this stage was presented on Figure 8.

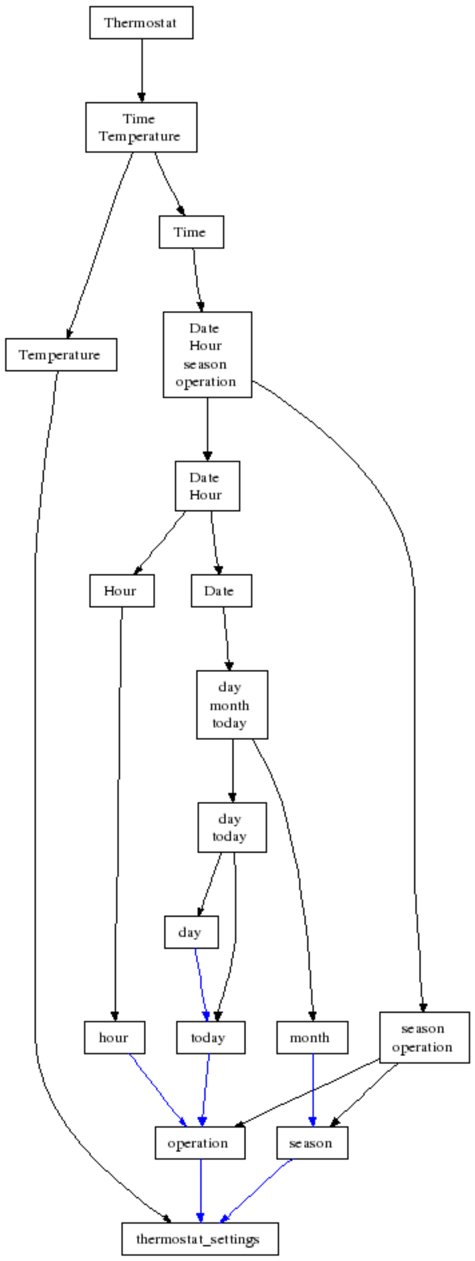
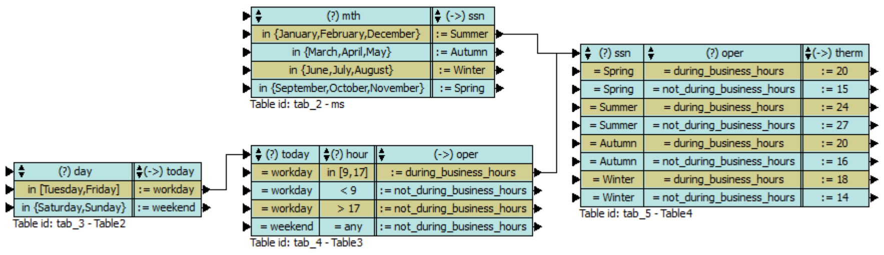


Figure 7: Example of ARD model generated by Varda editor



(?) mth	(->) ssn
in {January,February,December}	:= Summer
in {March,April,May}	:= Autumn
in {June,July,August}	:= Winter
in {September,October,November}	:= Spring

Table id: tab_2 - ms

(?) day	(->) today
in [Tuesday,Friday]	:= workday
in {Saturday,Sunday}	:= weekend

Table id: tab_3 - Table2

(?) today	(?) hour	(->) oper
= workday	in [9,17]	:= during_business_hours
= workday	< 9	:= not_during_business_hours
= workday	> 17	:= not_during_business_hours
= weekend	= any	:= not_during_business_hours

Table id: tab_4 - Table3

(?) ssn	(?) oper	(->) therm
= Spring	= during_business_hours	:= 20
= Spring	= not_during_business_hours	:= 15
= Summer	= during_business_hours	:= 24
= Summer	= not_during_business_hours	:= 27
= Autumn	= during_business_hours	:= 20
= Autumn	= not_during_business_hours	:= 16
= Winter	= during_business_hours	:= 18
= Winter	= not_during_business_hours	:= 14

Table id: tab_5 - Table4

Figure 8: Example of XTT model from HQEd editor

8.3 Implementation

Following file is an output from HQEd editor. It is a HMR file generated from the XTT model shown on fig. 8.

This is only a part of the file. For full file see appendix A.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TYPES DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%

xtype [name: months,
       base: symbolic,
       desc: 'All months of the year',
       domain: ['January'/1,'February'/2,'March'/3,'April'/4,'May'/5,
   'June'/6,'July'/7,'August'/8,'September'/9,'October'/10,
           'November'/11, 'December'/12],
       ordered: yes
      ].
xtype [name: seasons,
       base: symbolic,
       desc: 'Seasons of the year',
       domain: ['Winter'/1,'Spring'/2,'Summer'/3,'Autumn'/4],
       ordered: yes
      ].

...

%%%%%%%%%%%%%%%%%%%%%%%%% ATTRIBUTES DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%

xattr [name: month,
       abbrev: mth,
       class: simple,
       type: months,
       comm: in,
       callback: [ask_symbolic_GUI,[month]],
       desc: 'The current month'
      ].
xattr [name: season,
       abbrev: ssn,
       class: simple,
       type: seasons,
       comm: inter,
       desc: 'Current season'
      ].
...

%%%%%%%%%%%%%%%%%%%%%%%% TABLE SCHEMAS DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%

xschm ms: [month] ==> [season].

...

%%%%%%%%%%%%%%%%%%%%%%%%%%%% RULES DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%

xrule ms/1:
      [month in ['January','February','December']]
    ==>
      [season set 'Summer']
    :'Table4'.
xrule ms/2:
      [month in ['March','April','May']]
    ==>
      [season set 'Autumn'].
xrule ms/3:
      [month in ['June','July','August']]
    ==>
      [season set 'Winter'].
xrule ms/4:
      [month in ['September','October','November']]
    ==>
      [season set 'Spring'].

...
```

A HMR file shown above is a complete XTT model that can be run by HeaRT. To start inference process from a Prolog console, the file has to be consulted. A batch interface described in 7.2 can be used to do this.

After that, one of the predicates described in 7.1.1 can be used to start inference process. Due to the fact that this is a very simple example most modes will work the same way. The only difference may be seen when Token-Driven mode is on.

To start inference in TDI mode, following command shown below show be used. The *goxio/2* predicate uses callbacks mechanism to collect data for input attributes (see fig. 4 5).

```
>> goxio(['Table4'],tdi).
```

The `Table4'` was specified as a final table, since it is the one that produces information about thermostat settings (fig. 8).

The values chosen in this example was:

```
Day: Tuesday
Month: February
Hour: 18
```

The output from the inference process was presented below.

```
[day, Tuesday],[hour, 18],[month, February],[season, Summer],
[today, workday],[operation, not_during_business_hours]
```

According to the Token-Driven mode specification each table has assigned a number of tokens required to fire it. In the thermostat example the *ms* and *Table2* requires 0 tokens to be fired, the *Table3* requires 1 token, and *Table4* 2 tokens. This is why the inference process stops and *Table4* is never fired. The only rules that grant this table tokens are *Table3/1* and *ms/1*. The trajectory presented below shows, that in *Table3'* rule number 3 was fired, and the *Table4* did not earn the required token.

The trajectory of the TDI inference:

```
[[start, s12], [ms/1, s13], ['Table2'/1, s14], ['Table3'/3, s15]].
```

It is also possible to run inference process from HQEd editor. In order to do this HeaRT has to be run in a server mode. After loading a model, and choosing `Execute->HeaRT simulation` a window will pop-up allowing to choose inference mode and values for input attributes.

After launching an inference following information will be printed in an information window of HQEd editor:

```
------------ New model state ------------
Information: : Updated value of attribute 'day' to '2.0'.
Information: : Updated value of attribute 'hour' to '14'.
Information: : Updated value of attribute 'month' to '4.0'.
```

```
Information: : Updated value of attribute 'season' to '4.0'.
Information: : Updated value of attribute 'today' to 'workday'.
Information: : Updated value of attribute 'operation' to '1.0'.
Information: : Updated value of attribute 'thermostat_settings' to '20'.
------------ A sequence of analized rules ------------
Information: in table 'ms', row 2: fired rule.
Information: in table 'Table2', row 1: fired rule.
Information: in table 'Table3', row 1: fired rule.
Information: in table 'Table4', row 5: fired rule.
Information: 13 messages(s) found: 0 error(s), 0 warnings(s),
             13 informations(s).
```

It is also possible to create a WEB interface for the model. In the follwoing example the *PHeroic* library, described in Section 5.7.3 was used to create a simple interface shown on Figure 9. The interface allows running inference process from a given state, and it presents HeaRT responses in a nice graphical way.

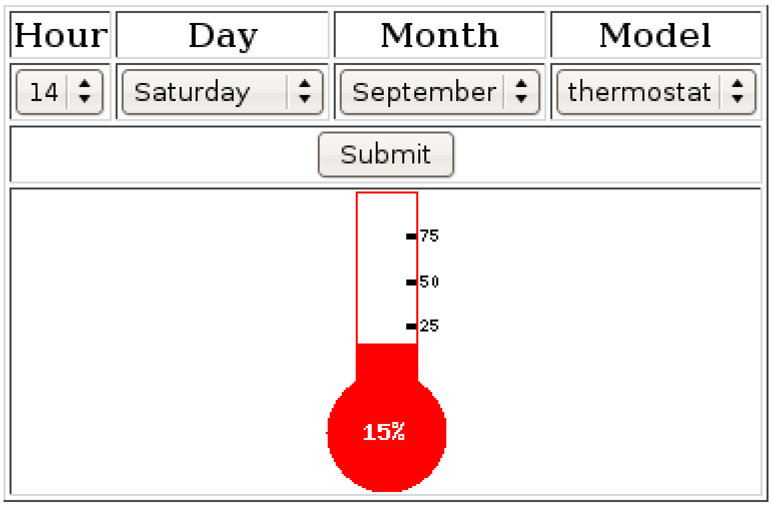


Figure 9: Example of Web interface for Thermostat XTT model

For the full code of the web interface presented on Figure 9 see appendix A.

8.4 Verification

Let the thermostat control system ([11]) be considered. The table-leveled analysis is provided by HalVA Verification Framework. The HMR representation is used.

Incompleteness Let the table *th* be described:

```
xschm th: [today,hour] ==> [operation].
```

The table consists only of two rules:

```
xrule th/1:
  [today eq workday,
   hour  gt 17]
  ==>
  [operation set not_bizhours].
xrule th/2:
  [today eq weekend,
   hour  eq any]
  ==>
  [operation set not_bizhours].
```

The verification of completeness points out uncovered states in the system.

```
>> vcomplete(th).
In th uncover states:
[[today, workday], [hour, 0]]
   [hour, 1]]    [hour, 2]]
   [hour, 3]]    [hour, 4]]
   [hour, 5]]    [hour, 6]]
   [hour, 7]]    [hour, 8]]
   [hour, 9]]    [hour, 10]]
   [hour, 11]]   [hour, 12]]
   [hour, 13]]   [hour, 14]]
   [hour, 15]]   [hour, 16]]
   [hour, 17]]
New rule: [today in [workday],
hour in [17, 16, 15, 14, 13, 12,
         11, 10, 9, 8, 7, 6, 5,
         4, 3, 2, 1, 0]] ==> _
No more uncovered states by table th
```

A rule marked as a new rule is a suggestion.

Contadiction The table *dt* is considered:

```
xschm dt: [day] ==> [today].
```

Adequate rules are defined as follows:

```
xrule dt/1:
  [day in [monday,tuesday,wednesday,
   thursday,friday,saturday]]
  ==>
  [today set workday].
xrule dt/2:
  [day in [saturday,sunday]]
  ==>
  [today set weekend].
```

The verification detects contradiction:

```
>> vcontradict(dt).
Clause [ today set weekend ] is
 conflicted with [ today set workday ]
In dt conflicted rules: 1 <=> 2
 for state  [[day, 6]]
No more conflicts in table dt
```

For state `[day, 6]` rules 1 and 2 are activated. But they produce conflicting result.

Subsumption Let a rule:

```
xrule th/5:
  [today eq weekend,
   hour  in [9 to 17]]
  ==>
  [operation set not_bizhours].
```

be added to the table *th*. The analysis of the subsumption shows that there are two rules, executable for the same state, but one of them produce more general conclusion.

```
>> vsubsume(th).
In th rule 2 subsumes 5
No more subsumption by table th
```

In this case, added rule is subsumed by rule already existed in the context (table).

Minimal set of rules In the control system of thermostat dependency between the season, hours and the thermostat settings are described by *os* table. The table consists of rules:

```
% (...)
xrule os/4:
  [operation eq during_bizhours,
   season eq spring]
  ==>
  [therm_set set 20].
% (...)
xrule os/8:
  [operation eq during_bizhours,
   season eq autumn]
  ==>
  [therm_set set 20].
```

Both of presented rules produce the same conclusion. Therefore, they could be replaced by one, more general rule.

```
>> vreduce(os).
In os rule 4 can be joined
  with rule 8 to new rule:
  [operation in [during_bizhours],
   season in [1, 3]] ==>
   [therm_set set 20]
In os rule 8 can be joined
  with rule 4 to new rule:
  [operation in [during_bizhours],
   season in [3, 1]] ==>
   [therm_set set 20]
No more reduction  in table os
```

As it is shown above, the HalVA algorithm reports, that rules 4 and 8 could be joined. The proposal of a new rule is introduced.

9 Summary

9.1 Evaluation

To evaluate the XTT2 method and the HeaRT engine, a number of benchmark rule systems have been modeled in the HeKatE project, see `https://ai.ia.agh.edu.pl/wiki/hekate:cases:start`. They prove the effectiveness of the design approach and rule structuring. However, at this point a direct comparison of inference effectiveness with other engines is not obvious due to different inference approach.

When it comes to comparing XTT2 and HeaRT to related approaches the focus is on two important solutions: CLIPS and its Java-based incarnation – Jess, as well as Drools, which inherits some of the important CLIPS features, while providing a number of high-level integration features.

XTT provides an expressive, formally defined language to describe rules. The language allows for formally described inference, property analysis, and code generation. Additional callbacks in rule decision provide means to invoke external functions or methods in any language. This feature is superior to those found in both CLIPS/Jess and Drools. On the other hand, the main limitation of the HeKatE approach is the state-base description of the system, where the state is understood as the set of attribute values.

The implicit rule base structure is another feature of XTT. Rules are grouped into decision tables during the design, and the inference control is designed during the conceptual design, and later on refined during the logical design. Therefore, the XTT representation is highly optimized towards rule base structurization. This feature makes the visual design much more transparent and scalable.

In fact all the Rete-based solutions seek some kind of structurization. In the case of CLIPS it is possible to modularize the rule base (see chapter 9 in [7]). It is possible to group rules in modules operating in given contexts, and then provide a context switching logic. Drools 5 offers Drools Flow that allows to define rule set and simple control structure determining their execution. In fact this is similar to the XTT-based solution. However, it is a weaker mechanism that does not correspond to table-based solution.

A complete design process seems to be in practice the most important issue. Both CLIPS and Jess are classic expert system shells, providing rule languages, and runtimes. They are not directly connected to any design methodology. The rule language does not have any visual representation, so no complete visual editors are available. Implementation for these systems can be supported by a number of external environments (e.g. Eclipse). However, it is worth emphasizing, that these tools do not visualize the knowledge contained in the rule base.

Drools 5 is decomposed into four main parts: Guvnor, Expert, Flow, Fusion. It offers several support tools, including an Eclipse-based environment. A "design support" feature, is the ability to read Excel files with simple decision tables. While this is a valuable feature, it does not provide constant syntax checking.

It is crucial to emphasize, that there is a fundamental difference between a graphical user interface like the one provided by generic Eclipse-based solutions, and *visual design support and specification* provided by languages such as XTT for rules, and in software engineering by UML. Other dedicated visual rule design languages include URML [16] that provides a UML-based representation for rules. Here focus is on single rules, not on decision tables, like in XTT.

9.2 Concluding Remarks

The main motivation for the research presented in this paper is to overcome some persistent problems of rule design and implementation including efficient rule representation and design, rule quality analysis, and rule-based system integration with the environment. The paper discusses the implementation of the HeaRT rule engine, a result of the *Hybrid Knowledge Engineering Project* (HeKatE, see `hekate.ia.agh.edu.pl`). In this approach the XTT2 rule language, formalized with the use of the ALSV(FD) logic is proposed [20, 21]. The language introduces explicit structure of the rule base, as well as visual rule representation based on the network of decision tables. This allows for an effective visual design on a high level of abstraction, where formal verification of rules is possible. To evaluate the engine, a number of benchmark systems have been modeled in the HeKatE project.

9.3 Future Work

Future work includes a tighter tool integration, as well as modeling complex cases in order to identify possible limitations of the XTT2 methodology. Providing a comparative studies modelling the same cases in XTT, CLIPS and Drools is planned. This would also allow for a more accurate comparison of the engine efficiency. Another direction is building a complete library of callback functions for common user interface elements in different languages. This would simplify interface integration on different platforms.

A HMR file for Thermostat model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TYPES DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%%%
xtype [name: default,
       base: symbolic,
       domain: [none/1],
       ordered: yes
      ].
xtype [name: weekdays,
       base: symbolic,
       desc: 'A set of days of the week',
       domain: ['Monday'/1,'Tuesday'/2,'Wednesday'/3,
       'Thursday'/4,'Friday'/5,'Saturday'/6,'Sunday'/7],
       ordered: yes
      ].
xtype [name: hours,
       base: numeric,
       length: 2,
       scale: 0,
       desc: 'Hours of a day',
       domain: [0 to 23]
      ].
xtype [name: months,
       base: symbolic,
       desc: 'All months of the year',
       domain: ['January'/1,'February'/2,'March'/3,'April'/4,'May'/5,
       'June'/6,'July'/7,'August'/8,'September'/9,'October'/10,
       'November'/11,'December'/12],
       ordered: yes
      ].
xtype [name: seasons,
       base: symbolic,
       desc: 'Seasons of the year',
       domain: ['Winter'/1,'Spring'/2,'Summer'/3,'Autumn'/4],
       ordered: yes
      ].
xtype [name: operating_hours,
       base: symbolic,
       desc: 'Office operating hours',
       domain: [not_during_business_hours/0,during_business_hours/1],
       ordered: yes
      ].
xtype [name: week_or_end,
       base: symbolic,
       desc: 'Weekend or weekday',
       domain: [workday,weekend]
      ].
xtype [name: thermostat_temperature,
       base: numeric,
       length: 2,
       scale: 0,
       desc: 'The temperature range of the thermostat',
       domain: [1 to 30]
      ].

%%%%%%%%%%%%%%%%%%%%%%%%% ATTRIBUTES DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%%%

xattr [name: day,
       abbrev: day,
       class: simple,
       type: weekdays,
       comm: in,
       callback: [ask_symbolic_GUI,[day]],
       desc: 'The current day'
      ].
xattr [name: hour,
       abbrev: hour,
       class: simple,
       type: hours,
       comm: in,
       callback: [ask_numeric_GUI,[hour]],
       desc: 'The current hour'
      ].
```

```
xattr [name: month,
       abbrev: mth,
       class: simple,
       type: months,
       comm: in,
       callback: [ask_symbolic_GUI,[month]],
       desc: 'The current month'
      ].
xattr [name: operation,
       abbrev: oper,
       class: simple,
       type: operating_hours,
       comm: inter,
       desc: 'Operating hours'
      ].
xattr [name: season,
       abbrev: ssn,
       class: simple,
       type: seasons,
       comm: inter,
       desc: 'Current season'
      ].
xattr [name: thermostat_settings,
       abbrev: therm,
       class: simple,
       type: thermostat_temperature,
       comm: out,
       callback: [tell_console,[thermostat_settings]],
       desc: 'The therostat setting'
      ].
xattr [name: today,
       abbrev: today,
       class: simple,
       type: week_or_end,
       comm: inter,
       desc: 'Part of the week'
      ].

%%%%%%%%%%%%%%%%%%%%%%%% TABLE SCHEMAS DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%

xschm ms: [month] ==> [season].
xschm 'Table2': [day] ==> [today].
xschm 'Table3': [today,hour] ==> [operation].
xschm 'Table4': [season,operation] ==> [thermostat_settings].

%%%%%%%%%%%%%%%%%%%%%%%%%%%% RULES DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%

xrule ms/1:
      [month in ['January','February','December']]
    ==>
      [season set 'Summer']
    :'Table4'.
xrule ms/2:
      [month in ['March','April','May']]
    ==>
      [season set 'Autumn'].
xrule ms/3:
      [month in ['June','July','August']]
    ==>
      [season set 'Winter'].
xrule ms/4:
      [month in ['September','October','November']]
    ==>
      [season set 'Spring'].

xrule 'Table2'/1:
      [day in ['Tuesday' to 'Friday']]
    ==>
      [today set workday]
    :'Table3'.
xrule 'Table2'/2:
      [day in ['Saturday','Sunday']]
    ==>
```

```
      [today set weekend].

xrule 'Table3'/1:
      [today eq workday,
       hour in [9 to 17]]
    ==>
      [operation set during_business_hours]
    :'Table4'.
xrule 'Table3'/2:
      [today eq workday,
       hour lt 9]
    ==>
      [operation set not_during_business_hours].
xrule 'Table3'/3:
      [today eq workday,
       hour gt 17]
    ==>
      [operation set not_during_business_hours].
xrule 'Table3'/4:
      [today eq weekend,
       hour eq any]
    ==>
      [operation set not_during_business_hours].

xrule 'Table4'/1:
      [season eq 'Spring',
       operation eq during_business_hours]
    ==>
      [thermostat_settings set 20].
xrule 'Table4'/2:
      [season eq 'Spring',
       operation eq not_during_business_hours]
    ==>
      [thermostat_settings set 15].
xrule 'Table4'/3:
      [season eq 'Summer',
       operation eq during_business_hours]
    ==>
      [thermostat_settings set 24].
xrule 'Table4'/4:
      [season eq 'Summer',
       operation eq not_during_business_hours]
    ==>
      [thermostat_settings set 27].
xrule 'Table4'/5:
      [season eq 'Autumn',
       operation eq during_business_hours]
    ==>
      [thermostat_settings set 20].
xrule 'Table4'/6:
      [season eq 'Autumn',
       operation eq not_during_business_hours]
    ==>
      [thermostat_settings set 16].
xrule 'Table4'/7:
      [season eq 'Winter',
       operation eq during_business_hours]
    ==>
      [thermostat_settings set 18].
xrule 'Table4'/8:
      [season eq 'Winter',
       operation eq not_during_business_hours]
    ==>
      [thermostat_settings set 14].

%%%%%%%%%%%%%%%%%%%%%%%%% CALLBACKS %%%%%%%%%%%%%%%%%%%%%%%%%%

xcall ask_symbolic_GUI : [AttName] >>> (jpl_new('skeleton.RequestInterface',[],T),
     alsv_domain(AttName,Domain,symbolic),
     Params = [AttName|Domain],
     term_to_atom(Params, AtomParams),
     jpl_call(T,request,['callbacks.input.ComboBoxFetcher',AtomParams],Answer),
     atom_to_term(Answer,Answer2,_),
     alsv_new_val(AttName,Answer2)).
xcall ask_numeric_GUI : [AttName] >>> (jpl_new('skeleton.RequestInterface',[],T),
     alsv_domain(AttName,[Min to Max],numeric),
     concat_atom(['[',AttName,',',Min,',',Max,']'],Parameters),
     jpl_call(T,request,['callbacks.input.SliderFetcher',Parameters],Answer),
     atom_to_term(Answer,Answer2,_),
     alsv_new_val(AttName,Answer2)).
xcall tell_console : [AttName] >>> (write('Attribute '),
    write(AttName),write(' output value is '),
    xstat(current:[AttName,V]),
    write(V),nl).
```

A Web interface for Thermostat model

index.php

```
<HTML>
<HEAD>
<TITLE>A web inteface for thermostat rule system</TITLE>
</HEAD>
<BODY>
<CENTER>
<FORM name="myform" method="post" action="process.php">
<TABLE border=1>
<TR><TH>Hour<TH>Day<TH>Month<TH> Model
<TR>
    <TD>
        <?php
            echo '<select name="hour">';
            for($i = 0; $i < 24; $i++)
                echo "<option value=\"$i\">$i</option>";
            echo '</select>'
        ?>
    <TD>
        <?php
            $days = array('Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday','Sunday');
            echo '<select name="day">';
            foreach($days as $s)
                echo "<option value=\"$s\">$s</option>";
            echo '</select>';
        ?>
    <TD>
        <?php
            $months = array('January','Februaru','March','April','May',
                            'June','July','August',
                            'September','October','November','December');
            echo '<select name="month">';
            foreach($months as $m)
                echo "<option value=$m>$m</option>";
            echo '</select>';
            ?>
    <TD>
        <?php
            // Connect HeaRT and fetch a list of available models
            include_once('PHeroic.php');
            $pheroic = new PHeroic("localhost",6000);
            $modelsList = $pheroic->getModelList();
            $modelsList = str_replace(array('[',']'),'',$modelsList);
            $modelArray = explode(',',$modelsList);
            echo '<select name="model">';
            for($i = 2; $i < count($modelArray); $i += 2)
                echo "<option value=\"$modelArray[$i]\">$modelArray[$i]</option>";
            echo '</select>';
        ?>
<TR><TD colspan="4"> <center> <input type="submit" value="Submit"></center>
<TR><TD colspan="4">
    <?php
        $response = $_GET["therm_set"];
        if(!empty($response)){
            echo '<center>';
            echo '<img border="0" src="thermometer.php?Current='.$response.'&Goal=100
&Width=60&Height=150&Font=1"';
            echo '</center>';
        }
    ?>
</TABLE>
</FORM>
</CENTER>
</BODY>
</HTML>
```

process.php

```
<?php
$hour = $_POST["hour"];
$day = $_POST["day"];
$month = $_POST["month"];
$model = trim($_POST["model"]);
// Arbitrary chosen username
$user = 'Thermostat';
include_once('PHeroic.php');
$pheroic = new PHeroic("localhost",6000);

$state =  '[[hour,'.$hour.'],[day,\''.$day.'\'],[month,\''.$month.'\']]';

$response = $pheroic->runSimulation($user, $model, gdi, '[\'Table4\']', $state);
preg_match('/thermostat_settings, [0-9]{2}/',$response, $matches);

$split_match = split(',', $matches[0]);
$setting = trim($split_match[1]);

header( 'Location: index.php?therm_set='.$setting ) ;


?>
```

thermometer.php

```
<?php

// This work is licensed under the Creative Commons Attribution 2.5 License.
// To view a copy of this license, visit
// http://creativecommons.org/licenses/by/2.5/
// or send a letter to Creative Commons, 543 Howard Street, 5th Floor,
// San Francisco, California, 94105, USA.
//
// Attribution (do not remove):
//    Original Creation of Arkie.Net - http://www.arkie.net/~scripts/
//

function thermGraph( $current, $goal, $width, $height, $font ) {

 $bar = 0.50;

 // create the image
 $image = ImageCreate($width, $height);
 $bg    = ImageColorAllocate($image,255,255,255 );
 $fg    = ImageColorAllocate($image,255,0,0);
 $tx    = ImageColorAllocate($image,0,0,0);

 //  Build background
 ImageFilledRectangle($image,0,0,$width,$height,$bg);

 //  Build bottom bulb
 imagearc($image, $width/2, $height-($width/2), $width, $width, 0, 360, $fg);
 ImageFillToBorder($image, $width/2, $height-($width/2), $fg, $fg);

 //  Build "Bottom level
 ImageFilledRectangle($image,
                      ($width/2)-(($width/2)*$bar),
                      $height-$width,
                      ($width/2)+(($width/2)*$bar),
                      $height-($width/2),
                      $fg );

 //  Draw Top Border
 ImageRectangle( $image,
                 ($width/2)-(($width/2)*$bar),
                 0,
                 ($width/2)+(($width/2)*$bar),
                 $height-$width,
                 $fg);

 //  Fill to %
 ImageFilledRectangle( $image,
                 ($width/2)-(($width/2)*$bar),
                 ($height-$width) * (1-($current/$goal)),
                 ($width/2)+(($width/2)*$bar),
                 $height-$width,
                 $fg );

 //  Add tic's
 for( $k=25; $k<100; $k+=25 ) {

     ImageFilledRectangle( $image,
            ($width/2)+(($width/2)*$bar) -5,
            ($height-$width) - ($height-$width)*($k/100) -1,
            ($width/2)+(($width/2)*$bar) -1,
            ($height-$width) - ($height-$width)*($k/100) +1, $tx );


     ImageString($image, $font,
            ($width/2)+(($width/2)*$bar) +2,
            (($height-$width) - ($height-$width)*($k/100)) - (ImageFontHeight($font)/2),
            sprintf( "%2d", $k),$tx);
 }

 // Add % over BULB
 $pct = sprintf( "%d%%", ($current/$goal)*100 );

 ImageString( $image, $font+2, ($width/2)-((strlen($pct)/2)*ImageFontWidth($font+2)),
    ($height-($width/2))-(ImageFontHeight($font+2) / 2),
    $pct, $bg);


 // send the image
 header("content-type: image/png");
 imagepng($image);
}

 thermGraph(
    $HTTP_GET_VARS["Current"],
    $HTTP_GET_VARS["Goal"],
    $HTTP_GET_VARS["Width"],
    $HTTP_GET_VARS["Height"],
    $HTTP_GET_VARS["Font"] );

?>
```

A Implementation of Heroic libraries

JHeroic

```
package JHeroic;

import java.net.*;
import java.io.*;
import java.util.ArrayList;

/**
 *
 * @author Szymon Bobek
 * @author Michal Gawedzki
 * @version 1.0.1
 *
 * Implementation of JHeroicInterface, allows to communicate with HeaRT using
 * HeaRT protocol in any Java application.
 *
 * 17.06.2009
 */
public class JHeroic implements JHeroicInterface
{
    /**
     * Name of host, where HeaRT server is working
     */
    public String hostName = null;

    /**
     * Debug mode on (1)/off (0)
     */
    public int debugMode = 0;

    /**
     * ipAddress of host, where HeaRT server is working
     */
    public InetAddress ipAddress = null;

    /**
     * port where HeaRT server is working
     */
    public int port = 0;

    /**
     * JHeroic constructor. Set debugMode to 1 if you want to save all
 exceptions to debug file, otherwise set 0.
     * Set hostName to name of the host and port to HeaRT port. This
 constructor does not establish any connection.
     *
     * @param debugMode 1 = Debug mode on, 0 = Debug mode off
     * @param hostName Name of the host, where application can connect
 do HeaRT server
     * @param port HeaRT server working port
     */
    public JHeroic(int debugMode, String hostName, int port)
    {
        this.debugMode = debugMode;
        this.hostName = hostName;
        this.port = port;
    }

    /**
     * JHeroic constructor. Set debugMode to 1 if you want to save all
 exceptions to debug file, otherwise set 0.
     * Set ipAddress to ip of the host and port to HeaRT port. This
 constructor does not establish any connection.
     *
     * @param debugMode 1 = Debug mode on, 0 = Debug mode off
     * @param ipAddress IP Address of HeaRT server
     * @param port HeaRT server working port
     */
    public JHeroic(int debugMode, InetAddress ipAddress, int port)
    {
        this.debugMode = debugMode;
        this.ipAddress = ipAddress;
        this.port = port;
    }

    /**
     * Returns list of all models in HeaRT as JHModel's array
     *
     * @return list of all models in HeaRT
     * @throws java.lang.Exception
     */
    public ArrayList<JHModel> getModelList() throws Exception
    {
        ArrayList<JHModel> modelList = new ArrayList<JHModel>();
        String request = "[model,getlist].";
        String replay = this.performRequest(request);

        if(replay.compareTo("[]") == 0)
            return modelList;

        replay.trim();
        replay = replay.substring(1, replay.length()-1);
        replay.trim();
        if(replay.compareTo("") == 0 || replay == null)
            return modelList;

        String[] models = replay.split(",");

        for(int i = 0; i < models.length;i++)
        {
            models[i] = models[i].trim();
            models[i+1] = models[i+1].trim();
            String userName = models[i];
            String modelName = models[i+1];

            char bChar = userName.charAt(0);
            char eChar = userName.charAt(userName.length()-1);

            if(bChar == '[')
                userName = userName.substring(1,userName.length());
            if(eChar == ']')
                userName = userName.substring(0,userName.length()-1);

            char bCharM = modelName.charAt(0);
            char eCharM = modelName.charAt(modelName.length()-1);

            if(bCharM == '[')
                modelName = modelName.substring(1,modelName.length());
            if(eCharM == ']')
                modelName = modelName.substring(0,modelName.length()-1);

            JHModel model = new JHModel(userName,modelName);
            modelList.add(model);
            i++;
        }

        return modelList;
    }

    /**
     * Returns user specified model as String
     *
     * @param userName Name of the user
     * @param modelName Name of the model
     * @return model as string without [ ] chars
     * @throws java.lang.Exception
     */
    public String getUserModel(String userName, String modelName)
throws Exception
    {
        String request = "[model,get,'" + modelName + "','" + userName + "'].";
        String replay = this.performRequest(request);
        replay = replay.substring(1, replay.length()-1);
        return replay;
    }

    /**
     * Adds new model for specified user
     *
     * @param userName Name of the user
     * @param modelName Name of the model
     * @param model model
     * @return request status
     * @throws java.lang.Exception
     */
    public String addUserModel(String userName, String modelName, String model)
throws Exception
```

```
    {
        String r = java.util.regex.Matcher.quoteReplacement("\\'");
        model = model.replaceAll("'",r);
        String request = "[model,add,'" + modelName + "','" + userName + "','" +
                          model + "']."; 
        String replay = this.performRequest(request);
        return replay;
    }

    /**
     * Removes user specified model
     *
     * @param userName Name of the user
     * @param modelName Name of the model
     * @return request status
     * @throws java.lang.Exception
     */
    public String removeUserModel(String userName, String modelName) throws Exception
    {
        String request = "[model,remove,'" + modelName + "','" + userName + "'].";
        String replay = this.performRequest(request);
        return replay;
    }

    /**
     * This method runs HeaRT with given model, inference type, tables and state
     *
     * @param userName name of the model owner
     * @param modelName model name
     * @param infType inference type, "ddi" = Data Driven, "gdi" = Goal Driven,
 "tdi" = Token Driven, any other string = Data Driven (default)
     * @param tables list of string, one string = one table
     * @param state name or full definition of state
     * @return request status
     * @throws java.lang.Exception
     */
    public String runInference(String userName, String modelName,
String infType, ArrayList<String> tables, String state) throws Exception
    {
        String iType = null;
        String tabStr = null;

        if(infType.compareTo("ddi") == 0)
        {
            iType = "dd";
        }
        else if(infType.compareTo("gdi") == 0)
        {
            iType = "gdi";
        }
        else if(infType.compareTo("tdi") == 0)
        {
            iType = "td";
        }
        else
        {
            iType = "dd";
        }

        for(int i = 0; i < tables.size(); i++)
        {
            if(i == 0)
            {
                tabStr = tables.get(i);
                continue;
            }

            tabStr = tabStr + "," + tables.get(i);
        }

        String request = "[model,run,'" + modelName + "','" + userName +
"','" + iType + "',['" + tabStr + "]','" + state + "'].";
        String replay = this.performRequest(request);
        return replay;
    }

    /**
     * This method adds new state to given model
     *
     * @param userName model owner name
     * @param modelName model name
     * @param stateName new state name
     * @param stateDef new state definition
     * @return request status
     * @throws java.lang.Exception
     */
    public String addStateToModel(String userName, String modelName,
String stateName, String stateDef) throws Exception
    {
        String request = "[state,add,'" + modelName + "','" +
userName + "','" + stateName + "','" + stateDef + "']."; 
        String replay = this.performRequest(request);
        return replay;
    }

    /**
     * This method removes given state from given model
     *
     * @param userName model owner name
     * @param modelName model name
     * @param stateName name of state do delete
     * @return request status
     * @throws java.lang.Exception
     */
    public String removeStateFromModel(String userName, String modelName,
String stateName) throws Exception
    {
        String request = "[stste,remove,'" + modelName + "','" +
userName + "','" + stateName + "']."; 
        String replay = this.performRequest(request);
        return replay;
    }

    /**
     * Returns current protocol version
     *
     * @return protocol version
     */
    public String getProtocolVersion()
    {
        return protocolVersion;
    }

    /**
     * Allows to run model verification in HeaRT engine
     * @param userName model's owner
     * @param modelName model name
     * @param mode mode of verification, "comp" = vcomplete,
 "contr" = vcontradict, "subs" = vsubsume, "redu" = vreduce
     * @return verification status
     * @throws java.lang.Exception
     */
    public String verifyModel(String userName, String modelName, String mode)
throws Exception
    {
        String vType = null;

        if(mode.compareTo("comp") == 0)
            vType = "vcomplete";
        else if(mode.compareTo("contr") == 0)
            vType = "vcontradict";
        else if(mode.compareTo("subs") == 0)
            vType = "vsubsume";
        else if(mode.compareTo("redu") == 0)
            vType = "vreduce";
        String request = "[model,verify,'" + vType + "','" + modelName +
"','" + userName + "']."; 
        String replay  = this.performRequest(request);
        return replay;
    }

    /**
     * Internal connection method
     *
     * @param request
     * @return
     * @throws java.lang.Exception
     */
    private String performRequest(String request) throws Exception
    {
        PrintWriter out = null;
        BufferedReader in = null;
        Socket socket = null;
        String replay = null;

        try {

            if(this.hostName == null && this.ipAddress != null)
            {
                socket = new Socket(this.ipAddress,this.port);
            }
            else if(this.ipAddress == null && this.hostName != null)
            {
                socket = new Socket(this.hostName,this.port);
            }

            out = new PrintWriter(socket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

            out.println(request);
            out.flush();

            replay  = in.readLine();
        }
        catch(Exception e)
        {
            if(this.debugMode == 1)
            {
                FileWriter outFile = new FileWriter("debug.log");
                PrintWriter fileWriter = new PrintWriter(outFile);

                fileWriter.println(e.getMessage());
                e.printStackTrace(fileWriter);
                fileWriter.close();
                outFile.close();

                System.err.println("Error data saved to debug.log");
                throw e;
            }
            else
            {
                throw e;
            }
        }
        finally
        {
            return replay;
        }
    }
}
```

PHeroic

```
<?php

    /**
    * PHeroic, PHP5 library for HeaRT engine
```

```
*
* @author Szymon Bobek
* @author Michal Gawedzki
* @version 1.0
* @license GPL 3
*
* 28.06.2009
 */
class PHeroic
{
    // Current protocol version
    private $protocolVersion = "1.0";

    // Hostname where HeaRT is accessbile
    private $hostname;

    // Port where HeaRT can be find
    private $port;

    /**
    * Constructor, returns instance of PHPHeroic library
    *
    * @param $hostname Hostname where HeaRT server is working
    * @param $port Port at HeaRT server is working
    * @return instance of PHPHeroic class
    */
    public function __construct($hostname, $port)
    {
        $this->hostname = $hostname;
        $this->port = $port;
    }

    /**
    * Allows to get list of all models in HeaRT. [user,model]
    *
    * @return list of models
    */
    public function getModelList()
    {
        $request = "[model,getlist].\n";
        $response = $this->performRequest($request);
        return $response;
    }

    /**
    * Allows to get user specified model
    *
    * @param $userName name of model's owner
    * @param $modelName name of the model
    * @return request status
    */
    public function getUserModel($userName,  $modelName)
    {
        $request = "[model,get,'".$modelName."','".$userName."'].\n";
        $response = $this->performRequest($request);
        return $response;
    }

    /**
    * Allows to add new model specified by user
    * IMPORTANT! Model should not contains " and $ characters
    *
    * @param $userName name of model's owner
    * @param $modelName name of the model
    * @param $model model to add
    * @return request status
    */
   public function addUserModel($userName, $modelName, $model)
    {
        $replaced_model = ereg_replace("'","\'",$model);
        $request = "[model,add,'".$modelName."','".$userName."','".$replaced_model."'].\n";
        $response = $this->performRequest($request);
        return $response;
        }

        /**
        * Allows to remove model from HeaRT
        *
        * @param $userName name of model’s owner
        * @param $modelName name of the model
        * @return request status
        */
       public function removeUserModel($userName, $modelName)
        {
            $request = "[model,remove,’".$modelName."’,’".$userName."’].\n";
            $response = $this->performRequest($request);
            return $response;
        }

        /**
        * Allows to run simulation of specified model
        *
        * @param $userName name of model’s owner
        * @param $modelName name of the model
        * @param $infType type of inference, 1 = Data Driven,
  2 = Goal Driven, 3 = Token Driven
        * @param $state state definition
        * @return state after simulation
        */
       public function runSimulation($userName, $modelName, $infType, $tables, $state)
        {

            $request = "[model,run,’".$modelName."’,’".$userName."’,
’".$infType."’,".$tables.",".$state."].\n";
            $response = $this->performRequest($request);
            return $response;
        }

        /**
        * Allows to add new state to specified model
        *
        * @param $userName name of model’s owner
        * @param $modelName name of the model
        * @param $stateName name of new state
        * @param $stateDef definition of the new state
        * @return request status
        */
       public function addStateToModel($userName, $modelName, $stateName, $stateDef)
        {
            $request = "[state,add,’".$modelName."’,’".$userName."’,’".$stateName."’,’".$stateDef."’].\n";
            $response = $this->performRequest($request);
            return $response;
        }

        /**
        * Allows to remove state from model
        *
        * @param $userName name of model’s ownser
        * @param $modelName name of the model
        * @param $stateName name of state to delete
        * @return request status
        */
       public function removeStateFromModel($userName, $modelName, $stateName)
        {
            $request = "[stste,remove,’".$modelName."’,’".$userName."’,’".$stateName."’].\n";
            $response = $this->performRequest($request);
            return $response;
        }

        /**
        * Returns current protocol version
        *
        * @return protocol version as string
        */
       public function getProtocolVersion()
        {
```

```
        return $this->protocolVersion;
    }

    /**
    * Allows to run verification of the model
    *
    * @param $userName name of model's owner
    * @param $modelName name of the model
    * @param $mode mode of verification, 1 = complete (vcomplete),
      2 = contradict (vcontradict), 3 = subsume (vsubsume), 4 = reduce (vreduce)
    * @return verification status
    */
   public  function verifyModel($userName, $modelName, $mode)
    {
        $verificationMode = "";

        if($mode == 1)
        {
            $verificationMode = "vcomplete";
        }
        else if($mode == 2)
        {
            $verificationMode = "vcontradict";
        }
        else if($mode == 3)
        {
            $verificationMode = "vsubsume";
        }
        else if($mode == 4)
        {
            $verificationMode = "vreduce";
        }

        $request = "[model,verify,'".$verificationMode."','".$modelName."',
                   '".$userName."'].\n";
        $response = $this->performRequest($request);
        return $response;
    }

    /**
    * Private connection status, do not use
    *
    * @param $request request message
    * @return request result
    */
    private function performRequest($request)
    {
        $socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);

        if($socket == true)
        {
            $result = socket_connect($socket, $this->hostname, $this->port);

            if($result == true)
            {
                socket_write($socket, $request, strlen($request));

                $response = "";
                while ($data = socket_read($socket, 1024))
                {

                    $response = $response.$data;
                }

                socket_close($socket);

                return $response;
            }
            else
            {
                echo "Cannot connect";
                return null;
            }
            }
            else
            {
                echo "Cannot create socket";
                return null;
            }
        }
    }

?>
```

YHeroic

```
import socket, string, sys

# YHeroic is class/library for communicatio with HeaRT engine via TCP protocol.
# @author Szymon Bobek
# @author Michal Gawedzki
# @version 1.0.1
# @license GPL 3
#
# 28.06.2009
#
class YHeroic:

    # Constructor, init intance of PHeroic library
    # @argument hostanem: hostname where HeaRT server can be found
    # @argument port: port where HeaRT server is available
    # @return: instance of object
    def __init__(self, hostname, port):
        self.hostname = hostname
        self.port = port
        self.protocolVersion = "1.0"
        print(self.protocolVersion)


    # Returns list of available models [user,model]
    # @return: list of available models
    def getModelList(self):
        request =  '[model,getlist].\n'
        response = self.performRequest(request)
        return response

    # Allows to get user's specified model
    # @argument userName: name of model's owner
    # @argument modelName: name of model you want to get
    # @return: user model as string
    def getUserModel(self, userName, modelName):
        request = "[model,get,'" + modelName + "','" + userName + "'].\n"
        response = self.performRequest(request)
        return response

    # Allows to add new model as specified user.
    # @argument userName: name of model's owner
    # @argument modelName: name of model you want to add
    # @argument model: model as string
    # @return: request status
    #
    # IMPORTANT! Model in last argument must be: one line, without
" characters and with end line char \n changed to \\n (means \n must be seen)
    def addUserModel(self, userName, modelName, model):
        model = model.replace("'","\\'")
        request = "[model,add,'" + modelName + "','" + userName + "','" + model + "'].\n"
        response = self.performRequest(request)
        return response

    # Allows to remove user specified model
    # @argument userName: name of model's owner
    # @argument modelName: name of model to remove
    # @return: request status
    def removeUserModel(self, userName, modelName):
        request = "[model,remove,'" + modelName + "','" + userName + "'].\n"
        response = self.performRequest(request)
        return response

    # Starts simulation of selected model
    # @argument userName: name of model’s owner
    # @argument modelName: name of model you want to simulate
    # @argument infType: type of inference. "ddi" = Data Driven,
"gdi" = Goal Driven, "tdi" = Token Driven
    # @argument tables: list of tables, separate with , char
    def runInference(self, userName, modelName, infType, tables, state):
        inference = ""
        if infType == "ddi":
            inference = "dd"
        elif infType == "gdi":
            inference = "gdi"
        elif infType == "tdi":
            inference = "td"
        else:
            inference = "dd"

        request = "[model,run,’" + modelName+ "’,’" + userName +
"’,’" + inference + "’,’[" + tables + "]’,’" + state + "’].\n"
        response = self.performRequest(request)
        return response

    # Adds new state to selected model
    # @argument userName: name of model’s owner
    # @argument modelName: name of the model
    # @argument stateName: name of the state you want to add
    # @argument stateDef: definition of the state
    # @return: request status
    def addStateToModel(self, userName, modelName, stateName, stateDef):
        request = "[state,add,’" + modelName + "’,’" + userName +
"’,’" + stateName + "’,’" + stateDef + "’].\n"
        response = self.performRequest(request)
        return response

    # Removes state from selected model
    # @argument userName: name of model’s owner
    # @argument modelName: name of the model
    # @argument stateName: name of state that you want to remove
    # @return: request status
    def removeStateFromModel(self, userName, modelName, stateName):
        request = "[stste,remove,’" + modelName + "’,’" +
userName + "’,’" + stateName + "’].\n"
        response = self.performRequest(request)
        return response

    # Returns current protocol version
    # @return: protocol version as string
    def getProtocolVersion(self):
        return self.protocolVersion

    # Runs model’s verification
    # @argument userName: name of model’s owner
    # @argument modelName: name of the model
    # @argument mode: mode of verification, "comp" = complete (vcomplete),
"contr" = contradict (vcontradict), "subs" = subsume (vsubsume),
        "redu" = reduce (vreduce)
    # @return: verification status
    def verifyModel(self, userName, modelName, mode):
        verification = ""

        if mode == "comp":
            verification = "vcomplete"
        elif mode == "contr":
            verification = "vcontradict"
        elif mode == "subs":
            verification = "vsubsume"
        elif mode == "redu":
            verification = "vreduce"

        request = "[model,verify,’" + verification + "’,’" +
modelName + "',' " + userName + "'].\n"
        response = self.performRequest(request)
        return response

    # Internal communication function, do not use it
    def performRequest(self, request):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((self.hostname, self.port))
        sent = s.send(request)

        data = s.recv(1024)
        response = ""
        while len(data):
            response = response + data
            data = s.recv(1024)

        s.close()
        return response
```

References

[1] E. P. Andert. Integrated Knowledge-Based System Design and Validation for Solving Problems in Uncertain Environments. *International Journal of Man-Machine Studies*, 36(2):357–373, 1992.

[2] S. Bobek and M. Gawędzki. Design and implementation of a runtime environment for the xtt^2 rule representation method. Master's thesis, AGH University of Science and Technology, july 2009. Supervisor: G. J. Nalepa.

[3] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, 3rd edition, 2000.

[4] P. Browne. *JBoss Drools Business Rules*. Packt Publishing, 2009.

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

[6] J. Giarratano and G. Riley. *Expert Systems. Principles and Programming*. Thomson Course Technology, Boston, MA, United States, fourth edition edition, 2005. ISBN 0-534-38447-1.

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

[8] A. Giurca, D. Gasevic, and K. Taveter, editors. *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*. Information Science Reference, Hershey, New York, May 2009.

[9] J. Liebowitz, editor. *The Handbook of Applied Expert Systems*. CRC Press, Boca Raton, 1998.

[10] A. Ligęza. Logical Support for Design of Rule-Based Systems. Reliability and Quality Issues. *ECAI'96 Workshop on Validation, Verification and Refinement of Knowledge-Based Systems*, 1996.

[11] A. Ligęza. *Logical Foundations for Rule-Based Systems*. Uczelniane wydawnictwa Naukowo-Dydaktyczne AGH w Krakowie, 2005.

[12] A. Ligęza. *Logical Foundations for Rule-Based Systems*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne AGH w Krakowie, Kraków, 2005.

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

[14] A. Ligęza and G. J. Nalepa. Logical representation and verification of rules. In *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 273–301. Information Science Reference, Hershey, New York, 2009.

[15] A. Ligęza and G. J. Nalepa. Proposal of a formal verification framework for the xtt2 rule bases. In *CMS'09: Computer Methods and Systems 26–7 November 2009, Krakd'ž˝w, Poland*. AGH University of Science and Technology, Cracow, Oprogramowanie Naukowo-Techniczne, 2009.

[16] S. Lukichev and G. Wagner. Visual rules modeling. In *Sixth International Andrei Ershov Memorial Conference Perspectives of System Informatics, Novosibirsk, Russia, June 2006*, LNCS. Springer, 2005.

[17] G. J. Nalepa. Languages and tools for rule modeling. In *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 296–624. Information Science Reference, Hershey, New York, 2009.

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

[19] G. J. Nalepa and A. Ligęza. Prolog-Based Analysis of Tabular Rule-Based Systems with XTT Approach. In *FLAIRS Conference*, pages 426–431, 2006.

[20] G. J. Nalepa and A. Ligęza. Xtt+ rule design using the alsv(fd). In A. Giurca, A. Analyti, and G. Wagner, editors, *ECAI 2008: 18th European Conference on Artificial Intelligence: 2nd East European Workshop on Rule-based applications, RuleApps2008: Patras, 22 July 2008*, pages 11–15, Patras, 2008. University of Patras.

[21] G. J. Nalepa and A. Ligęza. Hekate methodology, hybrid engineering of intelligent systems. *International Journal of Applied Mathematics and Computer Science*, 2009. accepted for publication.

[22] G. J. Nalepa and A. Ligęza. HeKatE Methodology, Hybrid Engineering of Intelligent Systems. *Int. J. Appl. Math. Comput. Sci.*, 18(3):1–15, 2009.

[23] G. J. Nalepa, A. Ligęza, K. Kaczor, and W. T. Furmańska. Hekate rule runtime and design framework. In G. W. Adrian Giurca, Grzegorz J. Nalepa, editor, *Proceedings of the 3rd East European Workshop on Rule-Based Applications (RuleApps 2009) Cottbus, Germany, September 21, 2009*, pages 21–30, Cottbus, Germany, 2009.

[24] G. J. Nalepa and I. Wojnicki. Hierarchical rule design with hades the hekate toolchain. In M. Ganzha, M. Paprzycki, and T. Pelech-Pilichowski, editors, *Proceedings of the International Multiconference on Computer Science and Information Technology*, volume 3, pages 207–214. Polish Information Processing Society, 2008.

[25] G. J. Nalepa and I. Wojnicki. Towards formalization of ARD+ conceptual design and refinement method. In D. C. Wilson and H. C. Lane, editors, *FLAIRS-21: Proceedings of the twenty-first international Florida Artificial Intelligence Research Society conference: 15–17 may 2008, Coconut Grove, Florida, USA*, pages 353–358, Menlo Park, California, 2008. AAAI Press.

[26] D. L. Nazareth. Issues in the Verification of Knowledge in Rule-Based Systems. *International Journal of Man-Machine Studies*, 30(3):255–271, 1989.

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

[28] T. A. Nguyen, W. A. Perkins, T. J. Laffey, and D. Pecora. Checking an Expert Systems Knowledge Base for Consistency and Completeness. In *IJCAI*, pages 375–378, 1985.

[29] R. G. Ross. *Principles of the Business Rule Approach*. Addison-Wesley Professional, 1 edition, 2003.

[30] M. Suwa, C. A. Scott, and E. H. Shortliffe. *Completeness and consistency in rule-based expert system*, pages 159–170. Addison-Wesley, Reading, Massachusetts, 1985.

[31] J. Tepandi. Verification, testing, and validation of rule-based expert systems. *Proceedings of the 11-th IFAC World Congress*, pages 162–167, 1990.

[32] F. van Harmelen, V. Lifschitz, and B. Porter, editors. *Handbook of Knowledge Representation*. Elsevier Science, 2007.

[33] J. A. Wentworth, R. Knaus, and H. Aougab. Verification, Validation and Evaluation of Expert Systems. World Wide Web electronic publication: http://www.tfhrc.gov/advanc/vve/cover.htm.