Hierarchical Rule Design with HaDEs the HeKatE Toolchain

Grzegorz J. Nalepa
Institute of Automatics,
AGH – University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
Email: gjn@agh.edu.pl

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

***Abstract*—A hierarchical approach to decision rule design in the HeKatE project is introduced in the paper. Within this approach a rule prototyping method ARD+ is presented. It offers a high-level hierarchical prototyping of decision rules. The method allows to identify all properties of the system being designed, and using the algorithm presented in this paper, it allows to automatically build rule prototypes. A practical implementation of the prototyping algorithm is discussed and a design example is also presented. Using the rule prototypes actual rules can be designed in the logical design stage. The design process itself is supported by HaDEs, a set of of design tools developed within the project.**

I. Introduction

DESIGNING a knowledge-base for a rule-based system is a non-trivial task. The main issues regard identification of system properties, based on which rules are subsequently specified. This is an iterative process that needs a proper support from the design method being used, as well as computer tools supporting it. Unfortunately there are not many well-established tools for providing a formalized transition from vague concepts provided by the user or expert, to actual rules [1]. Quite often regular software engineering [2] methods are used [3] which are subject to so-called semantic gaps [4].

This paper focuses on transition from system properties to rule prototypes. In the paper ARD+, a design method for decision rules is discussed. The method allows for hierarchical rule prototyping that supports the actual gradual design process. A practical algorithm providing a transition from the ARD+ design to rule design is introduced. Using ARD+ and the algorithm it is possible to build a structured rule base, starting from a general user specification. This functionality is supported by the VARDA design tool implemented in Prolog [5]. A next step is design actual rules which is aided by HQED [6]. Both tools (VARDA, HQED) constitute the Base of HADES, the *HeKatE Design Environment*.

The paper is organized in the following manner. In Sect. II the most important aspects of rule design are summarized, then in Sect. III the hierarchical approach to rule design in the *HeKatE* project is introduced. Within this approach the rule prototyping method *ARD+* is presented in Sect. V. Using the algorithm presented in Sect. VI it is possible to automatically build decision rule prototypes. A practical implementation of the prototyping algorithm is discussed in Sect. VII, with a design example presented in Sect. VIII. The paper ends with the concluding remarks and directions for future work in Sect. IX.

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

II. Rule Design

The basic goal of rule design is to build a rule-based knowledge base from system specification. This process is a case of knowledge engineering (KE) [1], [7]. In general, the KE process is different in many aspects to the classic software engineering (SE) process. The most important difference between SE and KE is that the former tries to model how the system works, while the latter tries to capture and represent what is known about the system. The KE approach assumes that information about how the system works can be inferred automatically from what is known about the system.

In case of rules, the design stage usually consists in writing actual rules, based on knowledge provided by an expert. The rules can be expressed in a natural language, this is often the case with informal approaches such as business rules [8]. However, it is worth pointing out that using some kind of formalization as early as possible in the design process improves design quality significantly.

The next stage is the rule implementation. It is usually targeted at specific rule engine. Some examples of such engines are: *CLIPS*, *Jess*, and *JBoss Rules* (formerly *Drools*). The rule engine enforces a strict syntax on the rule language.

Another aspect of the design – in a very broad sense – is a rule encoding, in a machine readable format. In most cases it uses an XML-based representation. There are some well-established standards for rule markup languages:, e.g. *RuleML* and notably *RIF* (see www.w3.org/2005/rules).

The focus of this paper is the design stage, and the initial transition from user-provided specification that includes some *concepts*, to rule specification that connects rules with these concepts. This stage is often referred to as a *conceptual design*. It is also addressed with some recent representations, such as the *SBVR* [9] in the OMG and business rules communities.

The research presented in this paper is a part of the *HeKatE* project, that aims at providing an integrated rule design and

implementation method for rule-based systems. The HeKatE approach to the design is briefly introduced in the next section.

III. Hierarchical Design Approach

An effective design support for decision systems is a complex issue. It is related to design methods as well as the human-machine interface. Since most of the complex designs are created *gradually*, and they are often *refined* or *refactored*, the design method should take this process into account. Any tools aiding such a process should effectively support it.

It is worth noting that UML, the most common design method used in the SE, does not support the process at all. The process should not be mistaken with version or revision control. Simple cases of revision control are undo features, provided by most of UML tools (which is very rarely combined with real project versioning). More complex version control is usually delegated to some other, external tools in the userspace, such as CVS, Subversion, etc. But a tool supported even by a version control system is still unable to actually *present*, and *visualize* changes in the design, not mentioning its refinement or refactoring. This is related to the fact, that UML has no facilities to express the design process.

In real life the *design* support (as a process of building the design) is much more important then just providing means to visualize and construct the *design* (which is a kind of knowledge snapshot about the system). Another observation can be also made: *designing* is a knowledge-based process, where the *design* is considered a structure with procedures needed to build it (it is often the case in the SE).

In order to solve these problems, the HeKatE project aims at providing both design methods and tools that support the design process. They should be integrated, and provide a hierarchical design process that would allow for building a model, containing the subsequent design stages. Currently HeKatE supports the *conceptual design* with the ARD+ method (*Attribute Relationships Diagrams*) [10]. The main logical design is conducted with the use of XTT method (*eXtended Tabular Trees*) [11], [12].

The ARD+ design method is shortly presented in the subsequent section. It serves as a rule prototyping method for rules. The ARD+ method is a supportive design method for XTT, where the knowledge base is designed using a structured representation, based on the concept of tabular trees [13]. The XTT representation is based on the principle that the rule base is decomposed into a set of decision tables, grouping rules that have the same sets of conditional and decision attributes. It makes the knowledge base hierarchical, combining decision tables and decision trees approaches. While the use of ARD+ is aimed at XTT rules in HeKatE, it is a much more generic solution, allowing for prototyping different types of decision rules.

The HeKatE design process is supported by a dedicated toolchain presented in the next section.

IV. HADES, the HeKatE Toolchain Prototype

The toolchain in the HeKatE project is composed of several elements. Currently two main design tools are available. They are integrated using XML for knowledge representation.

VARDA [5] is an ARD rule prototyping tool. It provides a basic command line interface (CLI) for design creation, refinement and visualization. The CLI is based on the Prolog interactive shell which allows calling application programming interface (API) responsible for design manipulation. It includes system modelling, input/output operations and spawning a visualization tool-chain.

The modelling is provided through several predicates performing property and attribute manipulation, as well as the transformations. The input/output operations regard storing the design as a Prolog knowledge base or exporting it into XML-based format suitable for further processing, which also serves as HQEd input. It also allows to spawn the visualization toolchain based on GraphViz and ImageMagick which graphically presents the design.

VARDA is a crossplatform tool written in pure ISO Prolog, that depends only on the Prolog environment and GraphViz library. It is available under terms of the GNU GPL from https://ai.ia.agh.edu.pl/wiki/hekate:varda.

A. HQEd

HQeD [6] is a complex CASE tool for the main XTT design. The tool supports the visual design process of the XTT tables. Using the rule prototypes generated from the ARD+ design with VARDA, HQEd allows for the actual logical rule design grouped with the XTT tables.

The editor allows for gradual refinement of rules, with an online checking of attribute domains, as well as simple table properties, such as inference related dead rules. In case of simple tables it is possible to emulate and visualize the inference process step-by step. However, the main quality feature being developed is the plugin framework, that allows for integrating Prolog-based analysis plugins to check formal properties of the XTT rule base, including completeness, redundancy, or determinism.

HQEd is a crossplatform tool written in C++, that depends only on the Qt library. It is available under terms of the GNU GPL from https://ai.ia.agh.edu.pl/wiki/hekate:hqed.

The output from the editor is a complete rulebase encoded in Prolog. It can be executed using a Prolog-based inference engine. The rulebase with the inference engine can be integrated into a larger application as a logical core.

B. XML Markup

Knowledge in the HeKatE design process is described in HML, a machine readable XML-based format. HML consists of three logical parts: attribute specification (ATTML), attribute and property relationship specification (ARDML) and rule specification (XTTML).

The attribute specification regards describing attributes present in the system. It includes attribute names and data types used to store attribute values. The attribute and property relationship specification describes what properties the system consists of and which attribute identifies these properties.

Furthermore, it also stores all stages of the design process. The rule specification stores actual structured rules.

These logical parts: ATTML, ARDML and XTTML can be used in different scenarios as:

- pure ATTML – to describe just attributes and their domains,
- ATTML and ARDML combined – to describe the system being designed in terms of properties and dependencies among them,
- ATTML, ARDML and XTTML combined – attributes, dependencies and rules combined, a complete description of the system,
- ATTML and XTTML combined – just rules which are not designed out of properties, it could be used to model ad-hoc rules, or systems described by some predefined rules.

The HML, being an XML application, is formally described through a Document Type Definition (DTD). Its full specification and examples are available at https://ai.ia.agh.edu.pl/wiki/hekate:hekate_markup_language.

V. Introduction To ARD+

The ARD+ method aims at capturing relations between *attributes* in terms of *Attributive Logic* [14]. *Attributes* denote certain system *property*. A *property* is described by one or more attributes. ARD+ captures *functional dependencies* among these *properties*. A simple property is a property described by a single *attribute*, while a complex property is described by multiple *attributes*. It is indicated that particular system property depends functionally on other properties. Such dependencies form a directed graph with nodes being properties.

A typical atomic formula (fact) takes the form $A(p) = d$, where A is an attribute, p is a property and d is the current value of A for p. More complex descriptions take usually the form of conjunctions of such atoms and are omnipresent in the AI literature [15], [7].

Definition 1: Attribute. Let there be given the following, pairwise disjoint sets of symbols: P – a set of property symbols, A – a set of attribute names, D – a set of attribute values (the *domain*).

An attribute (see [14], [16]) A_i is a function (or partial function) of the form

$$A_i \colon P_j \to D_i.$$

A generalized attribute A_i is a function (or partial function) of the form $A_i \colon P_j \to 2^{D_i}$, where 2^{D_i} is the family of all the subsets of D_i.

Definition 2: Conceptual Attribute. A conceptual attribute *A* is an attribute describing some general, abstract aspect of the system to be specified and refined.

Conceptual attribute names are capitalized, e.g.: `WaterLevel`. Conceptual attributes are being *finalized* during the design process, into, possibly multiple, physical attributes, see Def. 8.

Definition 3: Physical Attribute. A physical attribute *a* is an attribute describing an aspect of the system with its domain defined.

Names of physical attributes are not capitalized, e.g. `theWaterLevelInTank1`. By finalization, a physical attribute origins from one or more (indirectly) conceptual attributes. Physical attributes cannot be finalized, they are present in the final rules.

Definition 4: Simple Property. PS is a property described by a *single* attribute.

Definition 5: Complex Property. PC is a property described by *multiple* attributes.

Definition 6: Dependency. A dependency D is an ordered pair of properties $D = \langle p_1, p_2 \rangle$ where p_1 is the independent property, and p_2 is the one that dependent on p_1.

Definition 7: Diagram. An ARD+ diagram G is a pair $G = \langle P, D \rangle$ where P is a set of *properties*, and D is a set of *dependencies*.

Constraint 1: Diagram Restrictions. The diagram constitutes a *directed graph* with certain restrictions:

1) In the diagram cycles are allowed.
2) Between two properties only a single dependency is allowed.

Diagram transformations are one of the core concepts in the ARD+. They serve as a tool for diagram specification and development. For the transformation T such as $T : D_1 \to D_2$, where D_1 and D_2 are both diagrams, the diagram D_2 carries more knowledge, is more specific and less abstract than the D_1. Transformations regard *properties*. Some transformations are required to specify additional *dependencies* or introduce new *attributes*, though. A transformed diagram D_2 constitutes a more detailed *diagram level*.

Definition 8: Finalization. The finalization TF is a function of the form

$$TF : PS \to P$$

transforming a simple property PS described by a conceptual attribute into a P, where the attribute describing PS is substituted by one or more conceptual or physical attributes describing P. It introduces more attributes (more knowledge) regarding particular property.

An interpretation of the substitution is, that new attributes describing P are more detailed and specific than attributes describing PS.

Definition 9: Split. A split is a function TS of the form:

$$TS : PC \to \{P_1, P_2, \ldots P_n\}$$

where a *complex property* PC is replaced by some number of *properties* ($\{P_1, P_2, \ldots P_n\}$), each of them described by one or more attributes originally describing PC. This transformation introduces more properties and defines functional relationships among them.

Constraint 2: Attribute Dependencies. Since PC may depend on other properties $PO_1 \ldots PO_m$, dependencies between these properties and $P_1 \ldots P_n$ have to be stated.

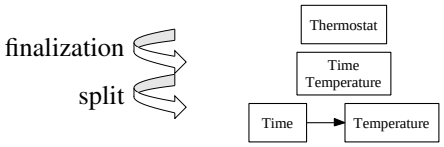

Figure 1. Examples of finalization and split

During the design process, upon splitting and finalization, the ARD+ model grows. This growth is expressed by consecutive diagram levels, making the design more and more specific. This constitutes the *hierarchical model*. Consecutive levels make a hierarchy of more and more detailed diagrams describing the designed system. The implementation of such a hierarchical model is provided through storing the lowest available, most detailed diagram level at any time, and additional information needed to recreate all of the higher levels, the so-called *Transformation Process History* (TPH). It captures information about changes made to properties at consecutive diagram levels. These changes are carried out through the transformations: *split* or *finalization*. A TPH forms a tree-like structure then, denoting what particular property is split into or what attributes a particular property attribute is finalized into.

In Fig. 1 the first and second ARD+ design level of the thermostat example discussed in Sect. VIII are presented. These are examples of finalization and split transformations.

VI. Rule Prototyping Algorithm

The goal of the algorithm is to automatically build prototypes for rules from the ARD+ design. The targeted rule base is structured, grouping rulesets in decision tables with explicit inference control among the latter. It is especially suitable for the XTT rule representation, however, this approach is more generic, and can be applied to any forward chaining rules.

The input of the algorithm is the most detailed ARD+ diagram, that has all of the physical attributes identified (in fact, the algorithm can also be applied to higher level diagrams, generating rules for some parts of the system being designed). The output is a set of *rule prototypes* in a very general format (`atts` stands for attributes):

```
rule: condition atts | decision atts
```

The algorithm is *reversible*, that is having a set of rules in the above format, it is possible to recreate the most detailed level of the ARD+ diagram.

In order to formulate the algorithm some basic subgraphs in the ARD+ structure are considered. These are presented in Figs. 2,3. Now, considering the ARD+ semantics (functional dependencies among properties), the corresponding rule prototypes are as follows:

- for the case in Fig. 2: `rule: e | f, g, h`
- for the case in Fig. 3: `rule: a, b, c | d`

In a general case a subgraph in Fig. 4 is considered. Such a subgraph corresponds to the following rule prototype:

```
rule: alpha, beta, gamma, aa | bb
```

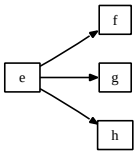

Figure 2. A subgraph in the ARD+ structure, case #1

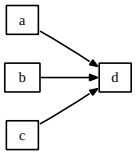

Figure 3. A subgraph in the ARD+ structure, case #2

```
rule: aa | xx, yy, zz
```

Analyzing these cases a general prototyping algorithm has been formulated. Assuming that a dependency between two properties is formulated as: $D(IndependentProperty, DependentProperty)$, the algorithm is as follows:

1) choose a dependency $D : D(F,T), F \neq T$, from all dependencies present in the design,
2) find all properties F, that T depends on: let $F_T = \{F_{T_i} : D(F_{T_i}, T), F_{T_i} \neq F\}$,
3) find all properties which depend on F and F alone: let $T_F = \{T_{F_i} : D(F, T_{F_i}), T_{F_i} \neq T, / \exists T_{F_i} : (D(X, T_{F_i}), X \neq F)\}$
4) if $F_T \neq \emptyset, T_F \neq \emptyset$ then generate rule prototypes:
   `rule: F, FT1, FT2,... | T`
   `rule: F | TF1, TF2,...`
5) if $F_T = \emptyset, T_F \neq \emptyset$ then generate rule prototypes:
   `rule: F | T, TF1, TF2,...`
6) if $F_T \neq \emptyset, T_F = \emptyset$ then generate rule prototypes:
   `rule: F, FT1, FT2,... | T`
7) if $F_T = \emptyset, T_F = \emptyset$ then generate rule prototypes:
   `rule: F | T`
8) if there are any dependencies left goto step 1.

Rule prototypes generated by the above algorithm can be further optimized. If there are rules with the same condition attributes they can be merged. Similarly, if there are rules with

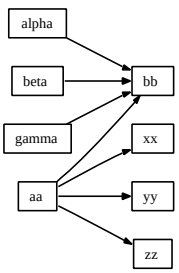

Figure 4. A subgraph in the ARD+ structure, general case

```
ax :-
  ard_depend(F,T),
  F \= T,
  ard_xtt(F,T),
  fail.
ax.

ft(F,T,FT):- ard_depend(F,T),
  ard_depend(FT,T), FT \=F.

tf(F,T,TF):- ard_depend(F,T),
  ard_depend(F,TF), TF \= T,
  \+ ( ard_depend(X,TF), X \= F).

% generate xtt from a dependency: F,T
ard_xtt(F,T):-
  ard_depend(F,T),
  \+ tf(F,T,_),
  \+ ft(F,T,_),
  assert(xtt([F],[T])),
  ard_done([F],[T]).
ard_xtt(F,T):-
  ard_depend(F,T),
  \+ tf(F,T,_),
  findall(FT,ft(F,T,FT),ListFT),
  assert(xtt([F|ListFT],[T])),
  ard_done([F|ListFT],[T]).
ard_xtt(F,T):-
  ard_depend(F,T),
  \+ft(F,T,_),
  findall(TF,tf(F,T,TF),ListTF),
  assert(xtt([F],[T|ListTF])),
  ard_done([F],[T|ListTF]).
ard_xtt(F,T):-
  ard_depend(F,T),
  findall(TF,tf(F,T,TF),ListTF),
  findall(FT,ft(F,T,FT),ListFT),
  assert(xtt([F|ListFT],[T])),
  assert(xtt([F],ListTF)),
  ard_done([F|ListFT],[T]),
  ard_done([F],ListTF).
% retract already processed dependencies
ard_done(F,T):-
  member(FM,F),
  member(TM,T),
  retract(ard_depend(FM,TM)),
  fail.
ard_done(_,_).
```

Figure 5. Algorithm implementation in Prolog

the same decision attributes they can be merged as well. For instance, rules like:

```
rule: a, b | x  ;  rule: a, b | y
```

can be merged into a single rule: `rule: a, b | x, y`

The rule prototyping algorithm has been successfully implemented in Prolog, as presented in the next section.

VII. Algorithm Implementation

The ARD+ is a visual design method, so an appropriate computer tool supporting it should be developed. VARDA is a prototype ARD+ design tool written in Prolog. It is purely declarative and supports automated visualization at any design stage (diagram level) using *GraphViz* (graphiz.org). The prototype consists of an API for low-level ARD+ handling primitives such as defining attributes, properties and dependencies, and high-level primitives supporting the transformations, and visualization primitives as well. The algorithm presented here has been implemented as a part of VARDA.

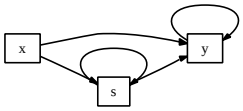

Figure 6. Factorial, ARD+

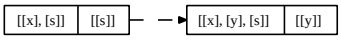

Figure 7. Factorial, XTT prototype

The algorithm implementation (see Fig. 5) consists of: a predicate `ax/0` which spawns the search process, browsing all dependencies (implementing the first step of the algorithm), `ard_xtt/2` predicate which implements steps four through seven, and helper predicates `ft/3`, `tf/3` providing steps two and three. Additional predicate `ard_done/2` removes dependencies which have already been processed. Dependencies are given as `ard_depend/2`.

In addition to the rule prototypes some higher-level relationships can be visualized on the final design. If attributes within a set of XTTs share a common ARD+ property, such XTTs are enclosed within a frame named after the property. It indicates that particular XTTs describe fully the property. Application of this feature is subject to further research, but it seems that it is suitable for establishing rule scope, implementing the XTT *Graphical Scope* feature proposed elsewhere.

The rule generation is reversible. Having rule prototypes it is always possible to recreate the original, most detailed ARD+ level (i.e. for redesign purposes). This functionality is fully implemented in VARDA.

The algorithm has been successfully tested on several examples, including the classic *UServ Product Derby 2005* case study from the Business Rules Forum [17]. This case study includes over 70 rules.

The algorithm covers even more complicated dependencies such these in Fig. 6, including self-dependencies. This example is a model for calculating factorial iteratively. Factorial of x is calculated and its value is stored in y, s serves as a counter. The diagram could be read as follows: y depends on x and s, in addition it also depends on itself, s depends on x and itself.

Using the above algorithm appropriate rule prototypes is generated:

```
rule: x, y, s | y  ; rule: x, s | s
```

which can be read, that a value of y is calculated based on values of x, y, and s. A value of s is calculated based on values of x and s.

These prototype rules are visualized in Fig. 7. Dashed arrows between XTTs indicate, in this case, that a value of s is needed in order to calculate a value of y.

This algorithm can be applied to a design case which is presented in the next section.

VIII. Design Example

A simple thermostat system is considered, see [18]. It regards a temperature control system which is to be used in an office. It controls heating and cooling which depends on several time oriented aspects. These include time of the day, business hours, day of week, month and season. Output of the system (a decision) regards what particular temperature the cooling/heating system should keep.

A. Conceptual Design

The design process includes system property identification and refinement including property relationships, and conceptual and physical attribute identification. The design stages, consecutive ARD+ diagrams, at more and more detailed levels, are given in Fig. 8 with the most detailed diagram at the bottom. Furthermore, the example shows rule prototypes (see Sec. VIII-B) and finally the rules to control the temperature (see Sec. VIII-C). Its TPH, regarding the most detailed level, is given in Fig. 9.

B. Rule Prototyping

A corresponding XTT prototype is given in Fig. 10. Higher level relationships are visualized as labeled frames, i.e.: `rule: day | today` originates from a property `Date`. Similarly, the set of XTT rule prototypes:

```
rule: day          | today
rule: month        | season
rule: today, hour  | operation
```

regards `Time` property. All the prototypes regard `Thermostat`, since it is the top most system property.

Dashed arrows between XTT prototypes indicate functional relationships between them. If there is an XTT with a decision attribute and the same attribute is subsequently used by other XTT as a condition one, there is a dashed arrow between such XTTs. These relationships also represent mandatory inference control, in terms of XTT approach.

C. Rule Design

The main logical rule design stage using HQEd is presented in Fig. 11, where the actual XTT tables corresponding to the prototypes generated using VARDA are visible.

IX. Concluding Remarks

In the paper the hierarchical Hekate design process is discussed. The process is supported by tools constituting the HADEs environment, these are: VARDA and HQED. VARDA is be used to assist a designer with system property identification. It can be applied as early as requirement specification, even to assist in gathering requirements. VARDA

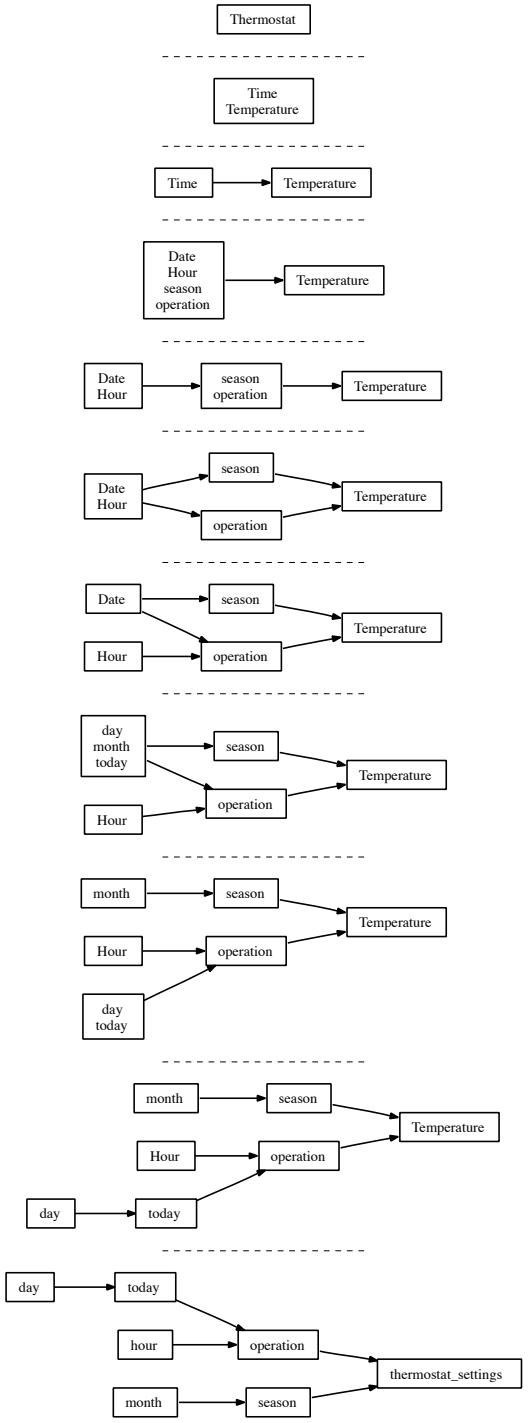

Figure 8. Thermostat design stages, ARD+

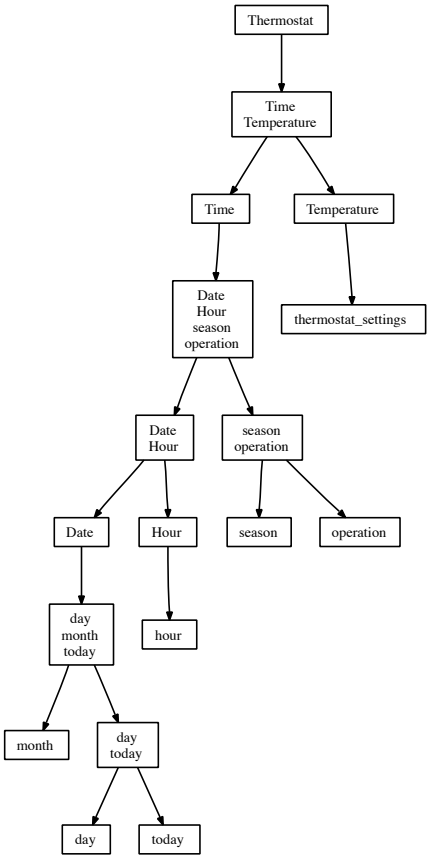

Figure 9. The thermostat, TPH

output is subsequently used as HQEd input being a template for rules. The actual rules are formulated using HQED.

The focus of the paper is on the ARD+, a hierarchical design method for rule prototyping. The original contribution of the paper is the introduction of the automatic rule prototyping algorithm based on properties identified in the ARD+ design process.

Applying ARD+ as a design process allows to identify attributes of the modelled system and refine them gradually. Providing the algorithm, which builds an XTT prototype out of ARD+ design, binds these two concepts together resulting in an uniform design and implementation methodology. The methodology remains hierarchical and gradual, the algorithm works both ways: it generates rule prototypes and it is also capable of recreating the most detailed ARD+ level. Combining the most detailed ARD+ level with the TPH allows to recreate any previous ARD+ level, which provides refactoring capabilities to an already designed or even implemented system.

Future work focuses on the formal definition of the inference process within XTT. There is an initial XTT inference model which requires some extensions and adaptation to cover different use scenarios. A more tight integration of the tools within HADES is anticipated. Upon finishing it and implementing a prototype run-time environment, HADES will provide a uniform environment for design and implementation of rule-based systems. Applications being currently considered include business applications logic [19] as well as control applications for mobile robots.

REFERENCES

[1] J. Liebowitz, Ed., *The Handbook of Applied Expert Systems*. Boca Raton: CRC Press, 1998.

[2] I. Sommerville, *Software Engineering*, 7th ed., ser. International Computer Science. Pearson Education Limited, 2004.

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

[4] D. Merrit, "Best practices for rule-based application development," *Microsoft Architects JOURNAL*, vol. 1, 2004.

[5] G. J. Nalepa and I. Wojnicki, "An ARD+ design and visualization toolchain prototype in prolog," in *FLAIRS2008*, 2008, accepted.

[6] K. Kaczor, "Design and implementation of a unified rule base editor," Master's thesis, AGH Univerity of Science and Technology, june 2008, supervisor: G. J. Nalepa.

[7] A. A. Hopgood, *Intelligent Systems for Engineers and Scientists*, 2nd ed. Boca Raton London New York Washington, D.C.: CRC Press, 2001.

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

[9] OMG, "Semantics of business vocabulary and business rules (sbvr)," Object Management Group, Tech. Rep. dtc/06-03-02, 2006.

[10] G. J. Nalepa and I. Wojnicki, "Towards formalization of ARD+ conceptual design and refinement method," in *FLAIRS2008*, 2008, accepted.

[11] G. J. Nalepa and A. Ligęza, "A graphical tabular model for rule-based logic programming and verification," *Systems Science*, vol. 31, no. 2, pp. 89–95, 2005.

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

[13] A. Ligęza, I. Wojnicki, and G. Nalepa, "Tab-trees: a case tool for design of extended tabular systems," in *Database and Expert Systems Applications*, ser. Lecture Notes in Computer Sciences, H. M. et al., Ed. Berlin: Springer-Verlag, 2001, vol. 2113, pp. 422–431.

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

[15] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall, 2003.

[16] A. Ligęza and G. J. Nalepa, "Knowledge representation with granular attributive logic for XTT-based expert systems," in *FLAIRS-20 : Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference : Key West, Florida, May 7-9, 2007*, D. C. Wilson, G. C. J. Sutcliffe, and FLAIRS, Eds., Florida Artificial Intelligence Research Society. Menlo Park, California: AAAI Press, may 2007, pp. 530–535.

[17] BRForum, "Userv product derby case study," Business Rules Forum, Tech. Rep., 2005.

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

[19] G. J. Nalepa, "Business rules design and refinement using the XTT approach," in *FLAIRS-20 : Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference : Key West, Florida, May 7-9, 2007*, D. C. Wilson, G. C. J. Sutcliffe, and FLAIRS, Eds., Florida Artificial Intelligence Research Society. Menlo Park, California: AAAI Press, may 2007, pp. 536–541.

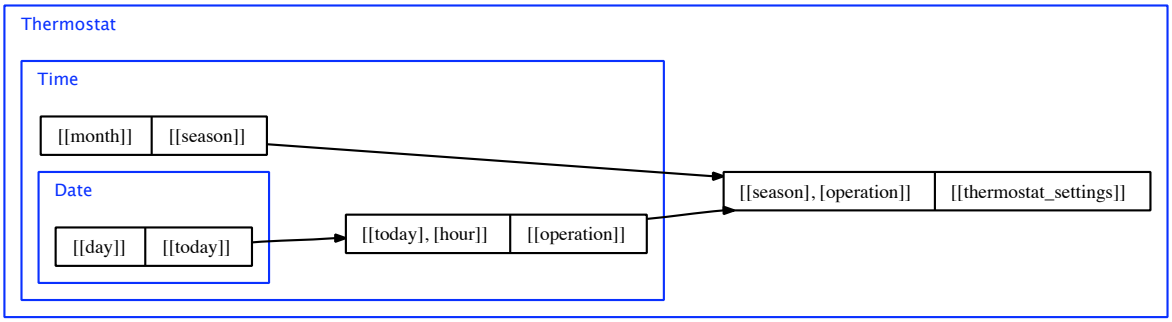

Figure 10. The thermostat, XTT prototype

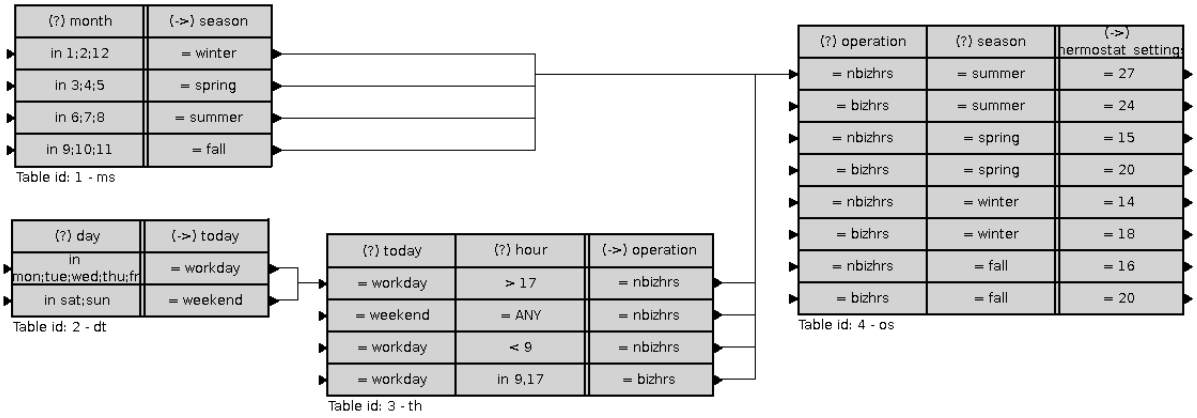

Figure 11. Thermostat rules