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

ARD+ a Prototyping Method for Decision Rules. Method Overview, Tools, and the Thermostat Case Study

Grzegorz J. Nalepa, Igor Wojnicki
AGH University of Science and Technology
Institute of Automatics
Kraków, POLAND
`{gjn,wojnicki}@agh.edu.pl`

Published online: 26.06.2009

CSL Technical Report No. 1/2009

CSL Technical Report No. 1/2009

ARD+ a Prototyping Method for Decision Rules. Method Overview, Tools, and the Thermostat Case Study[*]

Grzegorz J. Nalepa, Igor Wojnicki
AGH University of Science and Technology
Institute of Automatics
Kraków, POLAND
{gjn,wojnicki}@agh.edu.pl

Abstract. This report presents a design method for decision rules called ARD+. It is an extension of a previously proposed ARD method. In ARD+ the emphasis is put on the gradual refinement of the conceptual model of a knowledge-based system described with attributes. 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. The method is supported by a design tool called VARDA. The practical design aspects with ARD+ is are discussed using a thermostat case study.

Keywords: rule-based systems, rule prototyping, conceptual design, attribute relationship

Contents

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

List of Figures

1 Introduction

Providing a coherent design process for Knowledge-Based Systems [4], as well efficient CASE tools supporting it remains a challenge of Knowledge Engineering. Designing a knowledge-base for a rule-based system is a non-trivial task. The main issue regards 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 aiding 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 [5]. Quite often regular software engineering [22] methods are used [3] which are subject to so-called semantic gaps [8].

This paper focuses on the decision rule prototyping phase. In the paper a design method for decision rules called ARD+ (Attribute Relationship Diagrams) [15, 10] 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 [14].

The paper is organized in the following manner. In Sect. 2 the rule design process is discussed in general. Then the main motivation of the work is given in Sect. 3. The HeKatE approach is briefly introduced in Sect. 4. The formulation of the ARD+ method which is the focus of the paper is provided in Sect. 5. Using ARD+ design it is possible to automatically generate rule prototypes by an algorithm described in Sect. 6. In Sect. 7 the VARDA tool that supports the practical ARD+ design is introduced. Following this, the method is presented in Sect. 8 using a Thermostat case study. Sect. 9 provides conclusions and directions for future work.

2 Design Process for Rules

The basic goal of rule design is to build a rule-based knowledge base from a system specification. This process is a case of knowledge engineering (KE) [5, 4]. 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 real life the *design* process support (as a process of building the design) is much more important than 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 in fact a knowledge-based process, where the *design* is often considered a structure with procedures needed to build it (it is at least often the case in the SE).

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 [20]). 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 the 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 (often in natural language) 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* [18] by the OMG and business rules communities.

3 ARD Method Development

The primary motivation for this research is an apparent lack of standard prototyping method for rules, that would support the hierarchical and gradual design aspect covering the entire rule design process. Such a method should be formalized with use of some logic-based calculus in order to allow a formal analysis. The research presented in this paper is a part of the *HeKatE* project, that aims at providing an integrated and hierarchical rule design and implementation method for rule-based systems. The actual rule design is carried out with a flexible rule design method called XTT (eXtended Tabular Trees) [9]. The method introduces a structured rulebase. So one of the main requirements for the new prototyping method should be the support for XTT.

The original ARD method was first proposed by A. Ligęza and later on developed and described by G. J. Nalepa and A. Ligęza in [10, 6]. It was a supportive, and potentially optional, design method for XTT [9]. It provided simple means of attribute identification for the XTT method.

The first version of ARD was applied to simple cases only, and had no practical prototype implementation. The evolution of XTT, as well as larger complexity of systems designed with it, gave motivation for the major rework, and reformulation of ARD, which resulted in the version called ARD+. ARD+ was defined by G. J. Nalepa and I. Wojnicki in [15] and [13].

ARD+ is the main conceptual design method for rules in the HeKatE project (`hekate.ia.agh.edu.pl`). The main concepts of the HeKatE design approach are briefly introduced in the next section.

4 HeKatE Design Approach

The proposed approach introduces strong structuring of the design process. Furthermore, at any stage of partially designed system any knowledge component can be verified and corrected if necessary. The proposed approach follows the structural methodology for design of information systems. Simultaneously, this is a top-down approach, which allows for incorporating hierarchical design. Three stages (see Fig. 1) are considered:

1. The *Conceptual design*, in which system attributes and their functional relationships are identified (ARD+).

2. The *logical design with on-line verification*, during which the system structure is represented as an XTT hierarchy, which can be instantly analyzed, verified (and corrected, if necessary) and even optimized on-line.

3. The *physical design*, which is generated from the *logical design*.

The process is loosely based on the relational database design process [2].

One of the most important features of this approach is the *separation of the logical and physical design*, which also allows for a *hierarchical* design process being applied. The hierarchical conceptual model is mapped to a modular logical structure. The approach addresses three main problems:

- a visual representation,

- a functional dependency and logical structure, and

- a machine readable representation with an automated code generation.

A complete design becomes its implementation upon applying the automated code generation.

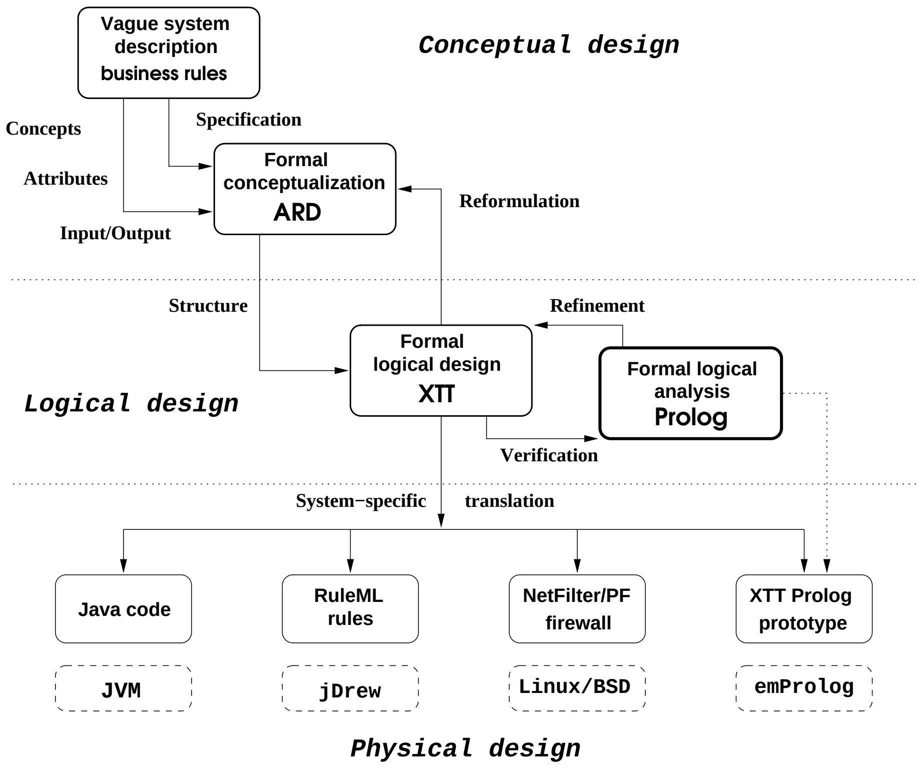

Figure 1: Hierarchical Design, Analysis & Implementation Process with ARD, XTT, and Prolog

5 ARD+ Method Formulation

5.1 Preliminary Concepts

The ARD+ method aims at supporting the rule designer at a very general design level, where the conceptualization of the design takes place [15, 13]. It is a knowledge-based approach, based on some classic AI methods [21]. 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.

The main concepts behind ARD+ are:

attributive logic based on the use of *attributes* for denoting certain properties in a system [6, 7],

functional dependency is a general relation between two or more attributes (or attribute sets), called "dependent" and "independent"; the relation is such as in order to determine the values of the dependent attributes, the values of the independent ones are needed,

graph notation provides simple, standard, yet expressive means for knowledge specification, and transformation,

visualization is the key concept in the practical design support, provided by this method,

gradual refinement is the main design approach, where the design is being specified in number of steps, each step being more detailed than the previous one,

structural transformations are *formalized*, with well defined syntax and semantics,

hierarchical model captures all of the subsequent design steps, with no semantic gaps; it holds knowledge about the system on the different abstraction levels [12],

knowledge-based approach provides means of the *declarative* model specification.

Based on these concepts, a formalization of the method is put forward in the following section.

5.2 Syntax

The ARD+ method aims at capturing relations between *attributes* in terms of *Attributive Logic* (AL) [6, 7, 11]. It is based on the use of *attributes* for expressing knowledge about facts regarding world under consideration. A typical atomic formula (fact) takes the form $A(o) = d$, where A is an attribute, o is an object and d is the current value of A for o. More complex descriptions take usually the form of conjunctions of such atoms and are omnipresent in the AI literature.

Definition 1 *Attribute. Let there be given the following, pairwise disjoint sets of symbols: O – a set of object name symbols, A – a set of attribute names, D – a set of attribute values (the* domains*).*
An attribute (see [6]) A_i is a function (or partial function) of the form:

$$A_i \colon O \to D_i.$$

A generalized attribute A_i is a function (or partial function) of the form $A_i \colon O \to 2^{D_i}$, where 2^{D_i} is the family of all the subsets of D_i, to let the attribute take more than a single value at a time.

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. 10.

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, see Def. 10. Physical attributes cannot be finalized, they are present in the final rules capturing knowledge about the system.

Definition 4 *Property P is a non-empty set of attributes representing knowledge about certain part of the system being designed. Such attributes* describe *the property. Simple Property PS is a property which consists of (is described by) a* single *attribute. Complex Property PC is a property which consists of (is described by)* multiple *attributes.*

Definition 5 *Dependency. A dependency D is an ordered pair of properties $D_{1,2} = \langle P_1, P_2 \rangle$, where P_1 is the independent property, and P_2 depends functionally on P_1.*

Definition 6 *Derivative V is an ordered pair of properties, such as: $V = \langle P_1, P_2 \rangle$, where P_2 is derived from P_1 upon transformation.*

Definition 7 *DPD. A Design Process Diagram is a triple $R = \langle P, D, V \rangle$ where P is a set of properties, D is a set of dependencies and V is a set of derivatives.*

Definition 8 *ARD+. An Attribute Relationship Diagram is a pair $G = \langle P, D \rangle$, where P is a set of* properties*, and D is a set of* dependencies *if there is a* DPD *$R = \langle P, D, V \rangle$.*

An example of ARD+ is given in Fig. 7 and 15.

Constraint 1 *Diagram Restrictions. The diagram constitutes a* directed graph *with possible cycles.*

Definition 9 *TPH. A Transformation Process History is a pair:* $TPH = \langle P, V \rangle$ *if there is a* DPD $R = \langle P, D, V \rangle$.

The TPH can be expressed as a directed graph with properties being nodes and derivatives being edges. An example of TPH is given in Fig. 16.

Let us now discuss the formalization of diagram transformations.

5.3 Transformations

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 $T : R_1 \rightarrow R_2$, where R_1 and R_2 are both *DPDs*, the diagram R_2 carries more system related knowledge, is more specific and less abstract than the R_1. All of the transformations regard *properties*. Some transformations are required to specify additional *dependencies* or introduce new *attributes* though. A transformed diagram R_2 constitutes a more detailed *diagram level*.

Definition 10 *Finalization Transformation. Finalization TF is a function which transforms a* DPD R *into* R_{TF} *by transforming a simple property* PS *consisting of a single conceptual attribute into a property* P*, where the attribute belonging to* PS *is substituted by one or more conceptual or physical attributes belonging to* P*; appropriate dependencies must be transformed as well, and a derivative has to be introduced.*

$$TF: R \rightarrow R_{TF}$$

It introduces more attributes (more knowledge) regarding particular property. An interpretation of the substitution is, that new attributes belonging to P are more detailed and specific than attributes belonging to PS.

Definition 11 *Split Transformation. Split TS is a function which transforms a* DPD R *into* R_{TS} *by transforming a complex property PC into some number of properties (a set)* $P_1, \ldots, P_r$*; appropriate dependencies and derivatives must be introduced.*

$$TS: R \rightarrow R_{TS}$$

This transformation introduces new properties and defines functional relationships among them.

There are several constrains regarding the transformations. They bring certain order to the design process and prevent transformation, property or attribute misuse.

Constraint 2 *Attribute Disjunction. Attribute sets belonging to each of the properties* $P_1 \ldots P_r$ *(regarding Def. 11) have to be disjoint. An attribute cannot belong to more than one property then.*

Constraint 3 *Attribute Matching. All attributes PC consists of have to belong to properties* $P_1, \ldots P_r$ *(regarding Def. 11). No new attributes can be introduced for properties* $P_1 \ldots P_r$*. Such introduction is possible through* finalization *only (see Def. 10).*

Constraint 4 *Attribute Pool. All attributes PC consists of have to belong to* $P_1, \ldots P_r$ *(regarding Def.11). An attribute cannot disappear during the transformation.*

Constraint 5 *Dependency Inheritance. No dependency that involves PC (regarding Def. 11) can disappear during the transformation. Such a dependency must be covered by at least one of* $P_1, \ldots P_r$ *properties introduced by the transformation.*

Constraint 6 *Transformation Limit. A number of transformations in a single design step is limited to* one per property. *It means that a property can be either* split *or* finalized *but not both. It forces gradual step-by-step refinement of the design.*

Let us now discuss the semantics of diagrams and their transformations.

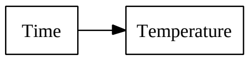

Figure 2: An example of dependency, ARD+

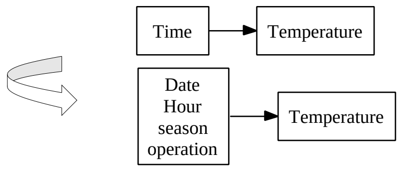

Figure 3: An example of finalization, ARD+

5.4 Semantics

The semantics of ARD+ will be explained and visualized using a reworked *Thermostat example* [17], discussed in [6, 10]. The goal of the system is to set the temperature in the office to the given set-point, based on the current time.

A property always consists of a set of attributes, these attributes identify such a property uniquely. The single most important concept in ARD+ is the concept of the *property functional dependency*. It indicates that in order to evaluate a dependent property, all independent properties have to be evaluated first. It means, that in order to calculate dependent property attribute values, independent properties attribute values have to be calculated first. An example is given in Fig. 2. It indicates that a property described by `Temperature` depends on a property described by `Time`.

Identifying all possible properties and attributes in a system could be a very complex task. Transformations (see Sec. 5.3) allow to gradually refine properties, attributes and functional dependencies in the system being designed. This process ends when all properties are described by *physical attributes* and all functional dependencies among properties are defined.

An example of the *finalization* is given in Fig. 3. The top diagram represents the system before the finalization. The property described by attribute `Time` is finalized. As a result new attributes are introduced: `Date`, `Hour`, `season`, `operation`. The outcome is the bottom diagram. Semantics of this transformation is that the system property described by a *conceptual attribute* `Time`, can be more adequately described by a set of more detailed attributes: `Date`, `Hour`, `season`, `operation`, which more precisely define the meaning `Time` in the design. The two latter attributes are *physical* ones, used in the final rule implementation. Finalization of properties based on such attributes is not allowed.

An example of *simple finalization* is given in Fig. 4. A property described by a *conceptual attribute* `Temperature` is finalized into a property described by a single *physical attribute* `thermostat_settings`. In other words, a general concept of temperature, represented by `Temperature`, is to be represented by an attribute `thermostat_settings`.

Another ARD+ transformation is the *split*. An example is given in Fig. 5. The top diagram shows a situation before, and the bottom one after, the *split transformation*. This *split* allows to refine new properties and define functional dependencies among them. A property described by attributes: `Date`, `Hour`, `season`, `operation`, is split into two properties described by `Date`, `Hour`, and `season`, `operation` appropriately. Furthermore there is a functional dependency defined such as `season` and `operation` depend on `Date` and `Hour`.

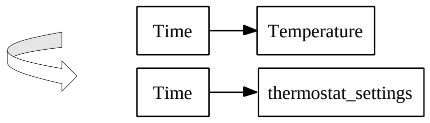

Figure 4: An example of simple finalization, ARD+

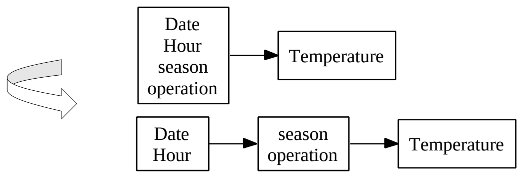

Figure 5: An example of split, ARD+

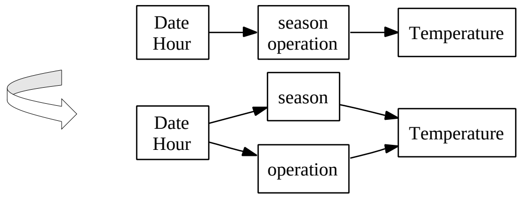

Figure 6: Yet another example of split, ARD+

During the split, some of the properties can be defined as functionally independent. An example of such a split is given in Fig. 6. Properties described by `season` and `operation` are functionally independent.

The ARD+ design process based on transformations leads to a diagram representing properties described only by *physical attributes*. An example of such a diagram is given in Fig. 7. All properties of the designed system are identified. Each of them is described by single *physical attribute*. All functional dependencies are defined as well.

Each transformation creates a more detailed diagram, introducing new attributes (*finalization*) or defining functional dependencies (*split*). These more detailed diagrams are called *levels of detail* or just *diagram levels*. In the above examples, transitions between two subsequent levels are presented. A complete model consists of all the levels capturing knowledge about all *splits* and *finalizations*.

Let us now discuss the practical issues of consistent and compact representation of the hierarchical model.

5.5 Hierarchical Model

During the design process, upon splitting and finalization, the ARD+ model grows, becoming more and more specific. This process constitutes the *hierarchical model*. Consecutive levels make a hierarchy of more and more detailed diagrams describing the designed system.

The purposes of having the hierarchical model are:

- gradual refinement of a designed system, and particularly,
- identification where given properties come from,
- ability to get back to previous diagram levels for refactoring purposes,
- big picture of the designed system.

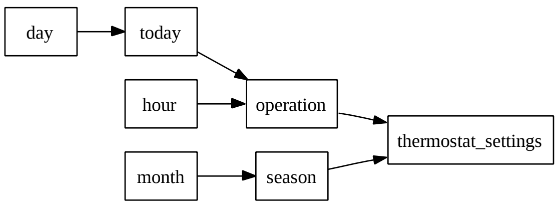

Figure 7: An ARD+ Diagram: all properties are identified

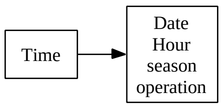

Figure 8: Transformation Process History for the diagrams from Fig. 3

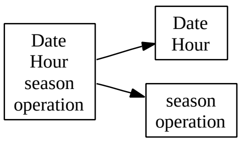

Figure 9: Transformation Process History for the diagrams from Fig. 5

The implementation of such hierarchical model is provided through storing the lowest available, most detailed diagram level, and, additionally, information needed to recreate all of the higher levels, so called *Transformation Process History*, TPH for short (see Def. 9).

The TPH captures information about changes made to properties at consecutive diagram levels. These changes are carried out through the transformations: split or finalization. The TPH forms a tree structure then, denoting what particular properties is split into or what attributes a particular property attribute is finalized into, according to Def. 9.

An example TPH for the transformation presented in Fig. 3 is given in Fig. 8. It indicates that a property described by an attribute `Time` is refined into a property described by attributes: `Date`, `Hour`, `season`, `operation`.

Another TPH example for the split transformation shown in Fig. 5 is given in Fig. 9. It indicates that a property described by attributes: `Date`, `Hour`, `season`, `operation` is refined into two properties described by: `Date`, `Hour` and `season`, `operation` attributes respectively.

Having a complete TPH and the most detailed level (namely ARD+), which constitute the *DPD* (according to Def. 7) it is possible to automatically recreate any, more general, level. A complete TPH for the thermostat example case is given in Sec. 8.

5.6 Refactoring

During the design process some properties or attributes can be missed or treated as not important, hence not included in the diagram. Refactoring allows to incorporate such artifacts or remove unnecessary ones. Refactoring consists in modifying any of the existing transformations: *finalization* or *split* in a particular ARD+ diagram.

5.6.1 Finalization

A *Finalization Refactoring* consist in adding or removing an attribute, modifying a past finalization transformation.

Removing an attribute A_r results in removing it from all already defined complex properties. If there is a simple property which regards such an attribute, it should be removed as well.

The *Finalization Refactoring* with adding an attribute implies that at some point a split has to be performed on a property involving the new attribute. Furthermore, appropriate dependencies between the property which consists of the introduced attribute and other properties have to be stated as well. This constitutes a *Split Refactoring*

5.6.2 Split

In general a *Split Refactoring* regards:

- adding or removing properties upon already defined splits,

- rearranging already defined dependencies.

Removing a property implies that all other properties that were split from it, even transitionally, have to be removed. Adding a property leads to defining its dependencies, between the property and other already defined ones.

In general adjusting dependencies can be based on:

- defining dependencies between the property and other existing properties at the *most detailed* diagram level, or
- adjusting appropriate past split transformations gradually (at previous diagram levels) to take this new property into consideration.

In the first case the dependencies are defined for the most detailed diagram level only. Since there is a hierarchical design process (see Sec. 5.5), these changes are taken into consideration at previous levels automatically.

The second case implies that the designer updates appropriate past splits in a gradual refinement process. Stopping this process at more general level than the most detailed one, generates appropriate splits in all more detailed levels automatically.

6 Rule Prototyping Algorithm

The goal of the algorithm is to automatically build prototypes for rules from the ARD+ design [13]. The targeted rule base is structured, grouping rule sets in decision tables with explicit inference control. It is especially suitable for the XTT rule representation. Moreover, 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. 10, 11. Now, considering the ARD+ semantics (functional dependencies among properties), the corresponding rule prototypes are as follows:

- for the case in Fig. 10: `rule: e       | f, g, h`
- for the case in Fig. 11: `rule: a, b, c | d`

In a general case a subgraph in Fig. 12 is considered. Such a subgraph corresponds to the following rule prototype:

```
rule: alpha, beta, gamma, aa | bb
rule: aa                     | xx, yy, zz
```

Analyzing above 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 $F_T = \{F_{T_i}: D(F_{T_i}, T), F_{T_i} \neq F\}$.

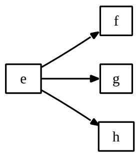

Figure 10: A subgraph in the ARD+ structure, case #1

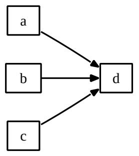

Figure 11: A subgraph in the ARD+ structure, case #2

3. Find all properties which depend on F alone:
   let $T_F = \{T_{F_i} : D(F, T_{F_i}), T_{F_i} \neq T, \nexists T_{F_i} : (D(X, T_{F_i}), X \neq F)\}$.
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 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 practical support form the ARD+ design method, including logical modelling, visualization, and the prototyping algorithm has been implemented in the VARDA presented in the next section.

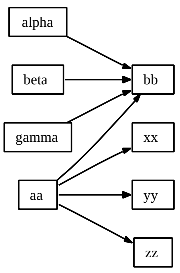

Figure 12: A subgraph in the ARD+ structure, general case

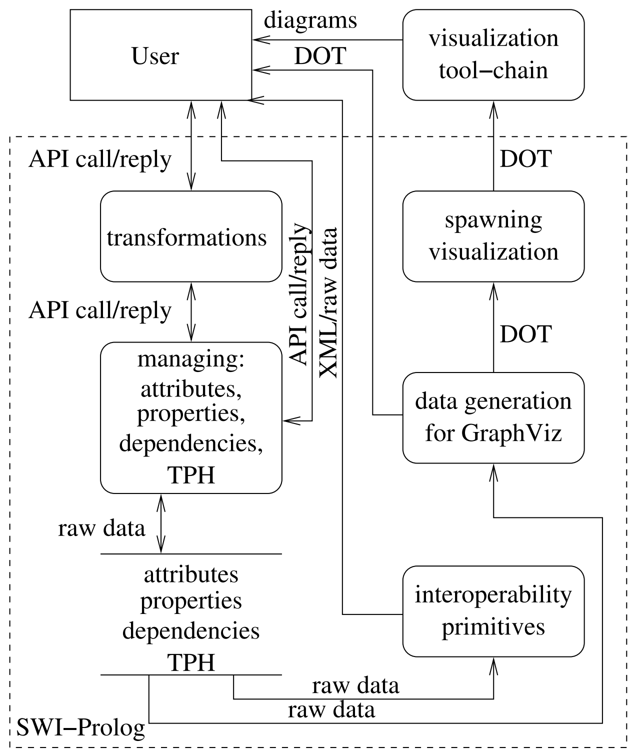

Figure 13: VARDA architecture

7 Design Tool Prototype

7.1 Architecture

A software tool prototype supporting the ARD+ design process (see Sect. 5) has been built. VARDA (*Visual ARD+ Rapid Development Alloy*) [14, 16] is a prototyping environment for ARD+, created with use of the SWI-Prolog environment for the knowledge base building, and Graphviz tool for an on-line design visualization. These tools are combined by the Unix environment, where the ImageMagick tool provides an instant displaying of the prototype at any design stage.

The tool is designed in a multi-layer architecture (see Fig. 13):

- knowledge base to represent the design: attributes, properties, dependencies,
- low-level primitives: adding and removing attributes, properties and dependencies,
- transformations: finalization and split including defining dependencies and automatic TPH creation,
- low-level visualization primitives: generating data for the visualization tool-chain,
- high-level visualization primitives: displaying actual dependency graph among properties and the TPH.

As an implementation environment of choice the Prolog language is used [1]. It serves as a proof of concept for the ARD+ design methodology and prototyping environment. However, switching to other environments such as Java, C++, Ajax, or Eclipse platform is possible. Prolog was chosen because it offers a rapid development environment for knowledge-based systems.

Knowledge base represents attributes, properties, dependencies and TPH as Prolog facts, using dynamic knowledge modification capability.

The low-level primitives are listed below. Additional marks at the arguments mean: `+`: argument has to be given, serves as input, `?`: argument does not have to be given, it serves as input, output or both, which complies with Prolog language notation.

`ard_att_add(+Attribute)` adding the given attribute.

`ard_att_del(?Attribute)` removing the given attribute, multiple attributes can be removed using this predicate as well,

`ard_property_add(+Property)` adding a property, the argument is a list of attributes describing the property, the property is uniquely identified by this list,

`ard_property_del(?Property)` removing a property, multiple properties can be removed using this predicate as well,

`ard_depend_add(+Independent,+Dependent)` adding a dependency between the given properties, the properties have to be defined prior to defining dependencies,

`ard_depend_del(?Independent,?Dependent)` removing a dependency, multiple dependencies can be removed as well.

The primitives regarding transformations are given below. They use the low-level primitives internally:

`ard_transform_finalize(+Property,+ListOfNewAttributes)` finalizing the given property, the second argument is a list of attributes describing a new property, the attributes have to be defined prior to finalization,

`ard_split(+Property,+PropertyList,+DependList)` splitting the given properties into a set of properties given as a second argument, dependencies among new properties are given as a list of dependencies as the third argument.

The low-level visualization primitives generate data for visualization toolchain. The toolchain is based on Unix (or Unix-like) environment and uses SWI-Prolog (`www.swi-prolog.org`), GraphViz (`www.graphviz.org`), and ImageMagick (`www.imagemagick.org`) (see Sect. 7.2).

The low-level Prolog predicates described below, generate input for GraphViz which renders appropriate graphs representing diagrams, and these diagrams are subsequently displayed by ImageMagick. These are:

`dotgen(+Edge)` generating a directed graph from a predicate given as an argument, the graph is described with GraphViz syntax as a text based representation displayed on standard output, it creates a complete directed graphs,

`dotgen(+Edge,+Node)` similar to `dotgen(+Edge)`, the second argument indicates a predicate which identifies nodes, it is capable of creating trivial or edgeless graphs.

Furthermore, there are several high level primitives supporting visualization, which spawn the tool-chain, these are:

`sar` displaying current ARD, and

`shi` displaying current TPH.

These predicates can be used at any time during the design process in the Prolog interactive shell. They launch the visualization tool-chain (see Sec. 7.2) in parallel with the shell.

7.2 Automated Visualization

At the design stage, proper visualization of the current design state, is a key element. It allows to browse the design more swiftly and identify gaps, misconceptions or mistakes more easily.

ARD+ and TPH are directed graphs. Therefore, a graph visualization primitives are needed. Proper graph visualization, node distribution, edge distribution and labeling is a separate scientific domain [19]. Instead of reinventing these concepts, or implementing them from scratch, a tool-chain of well proved tools to provide actual visualization is assembled.

Since, SWI-Prolog is currently used as a design environment platform, it is also used to generate appropriate data for visualization purposes. There are predicates (see Sec. 7) which generate GraphViz readable code from knowledge describing the ARD+ and TPH.

GraphViz (see `graphviz.org`) is a graph visualization software enabling representing structural information as diagrams of abstract graphs and networks. The structural information needs to be expressed in a simple text-based source file. GraphViz renders the source file and generates a visual representation of the structural information taking into account appropriate vertex distribution, edge placement and labeling. The visual representation is provided in many different formats, including but not limited to, bitmap formats such as: JPG, PNG, TIF, scalable formats: SVG, PS, as well as editable ones: FIG, DIA, VRML.

As for the tool-chain, GraphViz generates a PNG bitmap which subsequently is displayed by ImageMagick. ImageMagick does not merely display the diagram, but it also allows for panning, making annotations and saving it as a file in many bitmap formats including PNG and JPG.

There are two scenarios the tool-chain is used:

1. generating diagrams for an already designed system described in Prolog,
2. generating diagrams during the design process.

The first scenario can be executed as follows:

```
swipl -q -f 'ard-design.pl' -t go. | dot -Tpng | display
```

Assuming that `ard-design.pl` file contains the design coded with appropriate Prolog clauses (see Sec. 7), and predicate `go` triggers GraphViz data generation. The generated data is processed by GraphViz (`dot` utility) generating a PNG output which is passed to ImageMagick utility (`display`) which displays it and allows for annotation.

Generating diagrams during the design process is provided by two Prolog predicates: `sar` and `shi` (see Sec. 7) that generate the appropriate GraphViz source code, and spawn both GraphViz and ImageMagick subsequently. These predicates are accessible from the interactive Prolog shell, and display the ARD+ or the TPH. The tool-chain is executed in parallel with the interactive Prolog prompt, which allows to display several diagrams simultaneously.

7.3 Rule Prototyping

The rule prototyping algorithm has also been implemented as a part of VARDA. The algorithm implementation (see Fig. 14) 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.

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.

```
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 14: Prototyping algorithm implementation in Prolog

7.4 Design 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.

As for the ARD-only description including ARD+ and TPH diagrams ARDML with ATTML is used.

8 An Example Design

The ARD+ process can be easily explained using the following example. It is a reworked *Thermostat case*, found in [6, 10]. The main problem described there is to create a temperature control system for an office.

8.1 Conceptual Design

The design process is shown in Fig. 15. First, it is stated that there is a system to be designed which is described by a single conceptual attribute `Thermostat`. It is so called level 0 of the design. Prolog language (see Sec. 7) code providing this statement is given below:

```
  ard_att_add('Thermostat'),
  ard_property_add(['Thermostat']),
```

Refining knowledge about what is the purpose of the system makes a transition to the diagram at level 1; it is a finalization. It is stated that the thermostat controls temperature and this control has something to do with time. That is why `Thermostat` is finalized into `Time` and `Temperature`. Prolog language code providing this statement is given below:

```
  ard_att_add('Time'),
  ard_att_add('Temperature'),
  ard_finalize(['Thermostat'],['Time','Temperature']),
```

Furthermore, at level 2, it is stated that `Temperature` depends on `Time`. There are two properties identified in the system and a functional dependency between them. Prolog language code providing this statement is given below:

```
ard_split(['Time','Temperature'],
          [['Time'],['Temperature']],
          [[['Time'],['Temperature']]]),
```

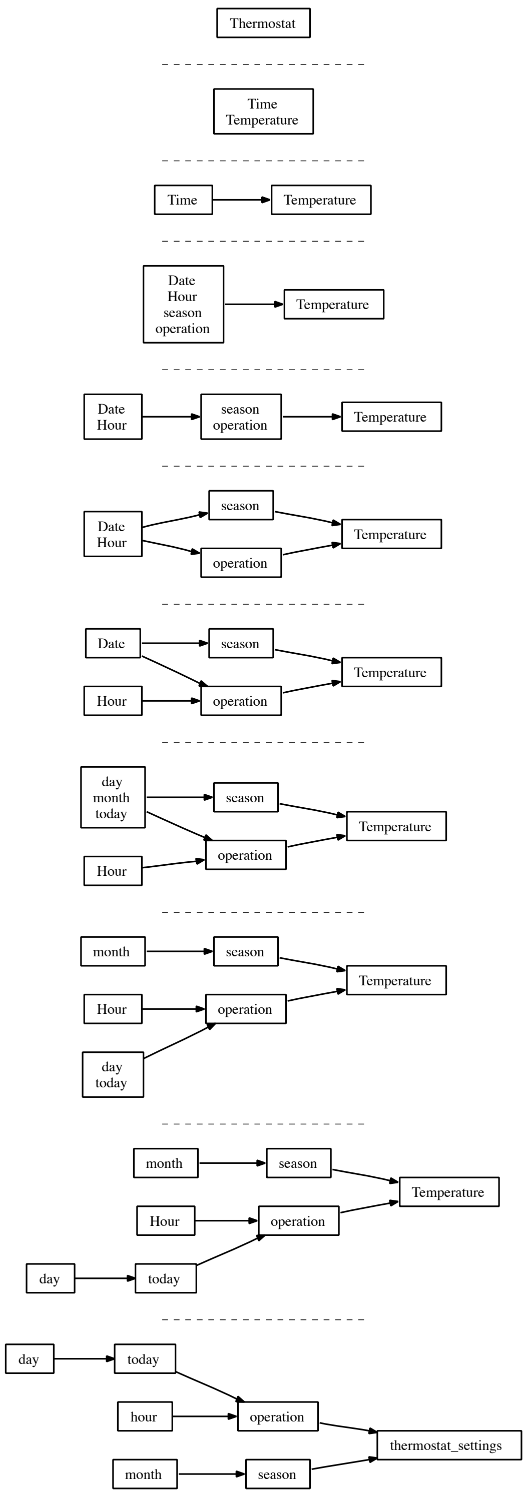

Figure 15: Thermostat design stages, ARD+

Then, at level 3, a general concept of time is refined. It is described by four attributes: `Date`, `Hour`, `season`, `operation`, through the finalization transformation. Attributes `Date` and `Hour` remain conceptual ones, since it is not precisely known at this point what exactly hour and date is in terms of their representation and perhaps further processing. `season` and `operation` are physical attributes. Their domains and constraints are defined to be used subsequently by the XTT process [9], they are not showed in the diagram though. Properties described by such physical attributes cannot be finalized any more. These attributes denote current season and whether the system operates within business hours or not. Prolog language code providing this statement is given below:

```
ard_att_add('Date'),
ard_att_add('Hour'),
ard_att_add(season),
ard_att_add(operation),
ard_finalize(['Time'],
             ['Date','Hour',season,operation]),
```

At level 4, it is specified that there is a functional relationship between properties described by attributes `Date`, `Hour`, and `season`, `operation`. It implies that values of `season` and `operation` are going to be calculated based on `Date` and `Hour`. However, it is not specified what exactly `Date` and `Hour` are. Prolog language code providing this statement is given below:

```
ard_split(['Date','Hour',season,operation],
          [['Date','Hour'],[season,operation]],
          [[['Date','Hour'],[season,operation]],
           [[season,operation],['Temperature']]]),
```

It is stated that `Date` and `Hour` are not functionally dependent at level 5. However, a property described by `season` and `operation` depends on both of them. Prolog language code providing this statement is given below:

```
ard_split(['Date','Hour'],
          [['Date'],['Hour']],
          [[['Date'],[season,operation]],
           [['Hour'],[season,operation]]]),
```

There is a finalization of `Date` at level 6. It is stated that `Date` is expressed by physical attributes: `day`, `month`, `today`. There is no `year` since it is stated that such an information is useless for the temperature control system. Prolog language code providing this statement is given below:

```
ard_att_add(day),
ard_att_add(month),
ard_att_add(today),
ard_finalize(['Date'],[day,month,today]),
```

Furthermore, there are functional dependencies defined at level 7. Prolog language code providing this statement is given below:

```
ard_split([day,month,today],
         [[month],[day,today]],
         [[[month],[season,operation]],
         [[day,today],[season,operation]]]),
```

Finally, there are more functional dependencies defined through splits at level 8 regarding attributes `day`, `today`, `season`, and `operation`. Prolog language code providing this statement is given below:

```
ard_att_add(thermostat_settings),
ard_finalize(['Temperature'],[thermostat_settings]),

ard_att_add(hour),
ard_finalize(['Hour'],[hour]).

ard_split([season,operation],
          [[season],[operation]],
          [[[month],[season]],
           [[day,today],[operation]],
           [[hour],[operation]],
           [[season],[thermostat_settings]],
           [[operation],[thermostat_settings]]]),

ard_split([day,today],
          [[day],[today]],
          [[[day],[today]],
           [[today],[operation]]])
```

The design process ends if all properties of the system are described by single physical attributes.

8.2 Design Process History

A complete Transformation Process History diagram is given in Fig. 16. It takes into consideration every transformation. Transformations are expressed by directed edges. The direction indicates the transformation direction. Vertices represent property state before and after the transformation according to the indicated direction.

Upon a *finalization* transformation number of attributes describing a property increases. For a *split* transformation number of properties increases since the source property is split into some number of properties which is indicated by many vertices in the diagram.

8.3 Rule Prototypes

A corresponding XTT prototype is given in Fig. 17. Higher level relationships are visualized as labeled frames, i.e.: `rule: day | today` originates from a property `Date`.[1]. 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.

The arrows between XTT tables 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 an arrow between such XTTs. These relationships also represent mandatory inference control, in terms of the XTT approach.

8.4 XML representation

Below the XML representation of the Thermostat ARD+ design is given.

[1]The semantics of the labeled frame is that the given set of XTT table prototypes is derived form a single conceptual property, whose name is the frame label.

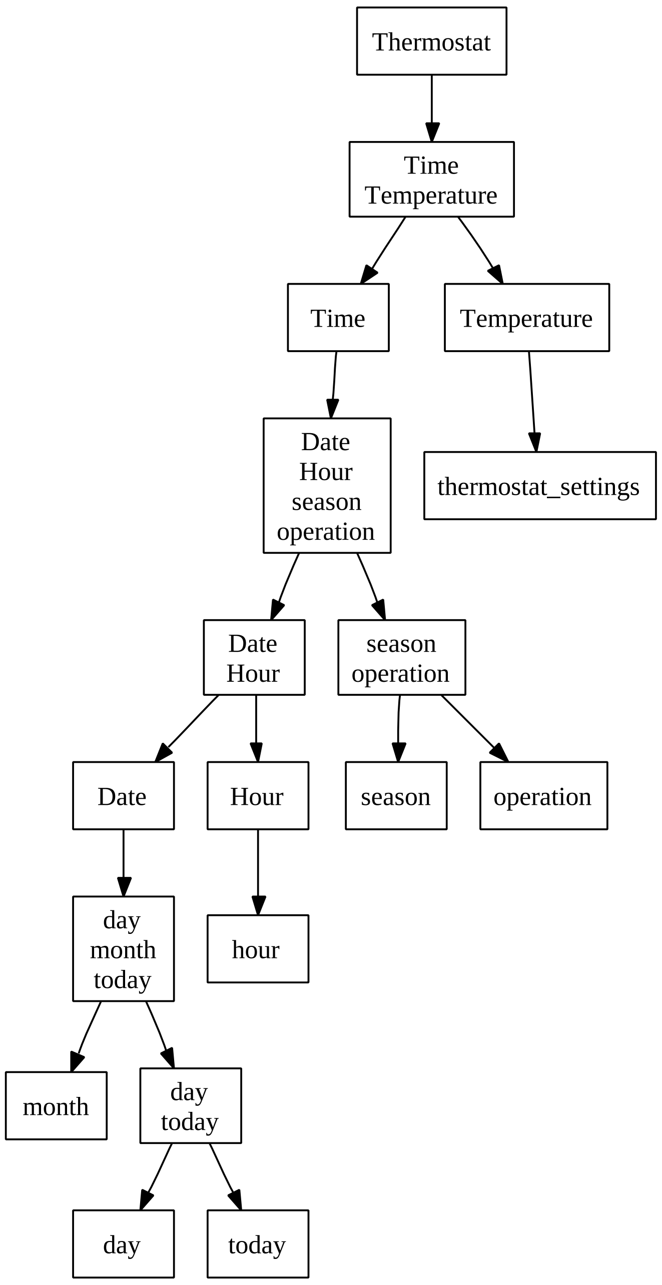

Figure 16: Thermostat TPH diagram

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE hml SYSTEM "hml.dtd">
<hml>
    <type_set>
    </type_set>
    <attribute_set>
        <att name="Thermostat" id="att_0" value="single" class="ro"/>
        <att name="Time" id="att_1" value="single" class="ro"/>
        <att name="Temperature" id="att_2" value="single" class="ro"/>
        <att name="Date" id="att_3" value="single" class="ro"/>
        <att name="Hour" id="att_4" value="single" class="ro"/>
        <att name="season" id="att_5" value="single" class="ro"/>
        <att name="operation" id="att_6" value="single" class="ro"/>
        <att name="day" id="att_7" value="single" class="ro"/>
```

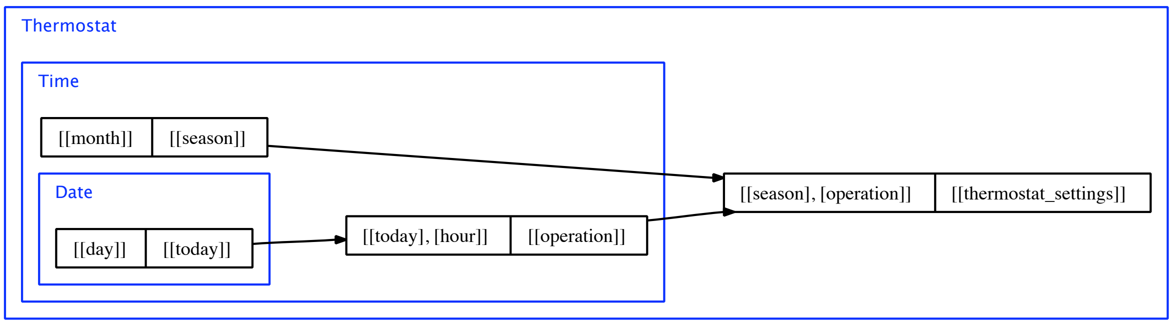

Figure 17: The thermostat, XTT prototype

```
        <att name="month" id="att_8" value="single" class="ro"/>
        <att name="today" id="att_9" value="single" class="ro"/>
        <att name="hour" id="att_10" value="single" class="ro"/>
        <att name="thermostat_settings" id="att_11" value="single"
             class="ro"/>
    </attribute_set>
    <property_set>
        <property id="prp_0">
            <attref ref="att_0"/>
        </property>
        <property id="prp_1">
            <attref ref="att_1"/>
            <attref ref="att_2"/>
        </property>
        <property id="prp_2">
            <attref ref="att_1"/>
        </property>
        <property id="prp_3">
            <attref ref="att_2"/>
        </property>
        <property id="prp_4">
            <attref ref="att_3"/>
            <attref ref="att_4"/>
            <attref ref="att_5"/>
            <attref ref="att_6"/>
        </property>
        <property id="prp_5">
            <attref ref="att_3"/>
            <attref ref="att_4"/>
        </property>
        <property id="prp_6">
            <attref ref="att_5"/>
            <attref ref="att_6"/>
        </property>
        <property id="prp_7">
            <attref ref="att_5"/>
        </property>
        <property id="prp_8">
            <attref ref="att_6"/>
        </property>
        <property id="prp_9">
            <attref ref="att_3"/>
        </property>
        <property id="prp_10">
            <attref ref="att_4"/>
        </property>
        <property id="prp_11">
            <attref ref="att_7"/>
            <attref ref="att_8"/>
            <attref ref="att_9"/>
        </property>
        <property id="prp_12">
            <attref ref="att_8"/>
        </property>
        <property id="prp_13">
            <attref ref="att_7"/>
            <attref ref="att_9"/>
        </property>
        <property id="prp_14">
            <attref ref="att_7"/>
        </property>
        <property id="prp_15">
            <attref ref="att_9"/>
        </property>
        <property id="prp_16">
            <attref ref="att_10"/>
        </property>
        <property id="prp_17">
            <attref ref="att_11"/>
        </property>
    </property_set>
    <tph>
        <trans src="prp_0" dst="prp_1"/>
        <trans src="prp_1" dst="prp_2"/>
        <trans src="prp_1" dst="prp_3"/>
        <trans src="prp_2" dst="prp_4"/>
        <trans src="prp_4" dst="prp_5"/>
        <trans src="prp_4" dst="prp_6"/>
        <trans src="prp_6" dst="prp_7"/>
        <trans src="prp_6" dst="prp_8"/>
        <trans src="prp_5" dst="prp_9"/>
        <trans src="prp_5" dst="prp_10"/>
        <trans src="prp_9" dst="prp_11"/>
        <trans src="prp_11" dst="prp_12"/>
        <trans src="prp_11" dst="prp_13"/>
        <trans src="prp_13" dst="prp_14"/>
        <trans src="prp_13" dst="prp_15"/>
        <trans src="prp_10" dst="prp_16"/>
        <trans src="prp_3" dst="prp_17"/>
    </tph>
    <ard>
        <dep independent="prp_12" dependent="prp_7"/>
        <dep independent="prp_14" dependent="prp_15"/>
        <dep independent="prp_15" dependent="prp_8"/>
        <dep independent="prp_16" dependent="prp_8"/>
        <dep independent="prp_7" dependent="prp_17"/>
        <dep independent="prp_8" dependent="prp_17"/>
    </ard>
</hml>
```

This representation is used by VARDA to store the model.

8.5 Prolog Representation

The same model is internally represented by VARDA in Prolog as:

```
:- dynamic ard_att/1.

ard_att('Thermostat').
ard_att('Time').
ard_att('Temperature').
ard_att('Date').
ard_att('Hour').
ard_att(season).
ard_att(operation).
ard_att(day).
ard_att(month).
ard_att(today).
ard_att(hour).
ard_att(thermostat_settings).

:- dynamic ard_property/1.

ard_property(['Thermostat']).
ard_property(['Time', 'Temperature']).
ard_property(['Time']).
ard_property(['Temperature']).
ard_property(['Date', 'Hour', season, operation]).
ard_property(['Date', 'Hour']).
ard_property([season, operation]).
ard_property([season]).
ard_property([operation]).
ard_property(['Date']).
ard_property(['Hour']).
ard_property([day, month, today]).
ard_property([month]).
ard_property([day, today]).
ard_property([day]).
ard_property([today]).
ard_property([hour]).
ard_property([thermostat_settings]).

:- dynamic ard_depend/2.

ard_depend([month], [season]).
ard_depend([day], [today]).
ard_depend([today], [operation]).
ard_depend([hour], [operation]).
ard_depend([season], [thermostat_settings]).
ard_depend([operation], [thermostat_settings]).

:- dynamic ard_hist/2.

ard_hist(['Thermostat'], ['Time', 'Temperature']).
ard_hist(['Time', 'Temperature'], ['Time']).
ard_hist(['Time', 'Temperature'], ['Temperature']).
ard_hist(['Time'], ['Date', 'Hour', season, operation]).
ard_hist(['Date', 'Hour', season, operation], ['Date', 'Hour']).
ard_hist(['Date', 'Hour', season, operation], [season, operation]).
ard_hist([season, operation], [season]).
ard_hist([season, operation], [operation]).
ard_hist(['Date', 'Hour'], ['Date']).
ard_hist(['Date', 'Hour'], ['Hour']).
ard_hist(['Date'], [day, month, today]).
ard_hist([day, month, today], [month]).
ard_hist([day, month, today], [day, today]).
ard_hist([day, today], [day]).
ard_hist([day, today], [today]).
ard_hist(['Hour'], [hour]).
ard_hist(['Temperature'], [thermostat_settings]).
```

9 Future Work

In this paper the HeKatE design process is discussed. The process is supported by a design tool called VARDA. The tool is used to assist the designer with system property identification. It can be applied as early as requirement specification, or even to assist in gathering requirements.

The focus of the paper is on the ARD+, a hierarchical design method for rule prototyping. 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 system.

In the future work the development of several design tools is considered. A native, interactive, full-fledged shell is being designed. The shell is to support ARD+ design process based on a text-oriented environment with heavy hinting. The hinting will support the designer with available choices at each design (diagram) level and it will also provide refactoring capabilities for already designed systems. There is also an interactive graphical environment being designed. Several implementations are under consideration, including a Prolog native, possibly XPCE-based (the SWI-Prolog GUI library), as well as Gtk-Server-based (`www.gtk-server.org`), or Java-based one.

References

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

[2] T. Connolly, C. Begg, and A. Strechan. *Database Systems, A Practical Approach to Design, Implementation, and Management*. Addison-Wesley, 2nd edition, 1999.

[3] 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.

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

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

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

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

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

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

[10] G. J. Nalepa and A. Ligęza. *Software engineering : evolution and emerging technologies*, volume 130 of *Frontiers in Artificial Intelligence and Applications*, chapter Conceptual modelling and automated implementation of rule-based systems, pages 330–340. IOS Press, Amsterdam, 2005.

[11] 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.

[12] G. J. Nalepa and I. Wojnicki. Filling the semantic gaps in systems engineering. In P. Scharff, editor, *52. IWK : Internationales Wissenschaftliches Kolloquium = International Scientific Colloquium : computer science meets automation : 10–13 September 2007 : proceedings*, volume 1, pages 107–112, Ilmenau : TU Ilmenau. Universitätsbibliothek, 2007. Technische Universität Ilmenau. Faculty of Science and Automation.

[13] 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.

[14] G. J. Nalepa and I. Wojnicki. An ARD+ design and visualization toolchain prototype in prolog. 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 373–374. AAAI Press, 2008.

[15] 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.

[16] G. J. Nalepa and I. Wojnicki. Varda rule design and visualization tool-chain. In A. R. Dengel and et al., editors, *KI 2008: Advances in Artificial Intelligence: 31st Annual German Conference on AI, KI 2008: Kaiserslautern, Germany, September 23–26, 2008*, volume 5243 of *LNAI*, pages 395–396, Berlin; Heidelberg, 2008. Springer Verlag.

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

[18] OMG. Semantics of business vocabulary and business rules (sbvr). Technical Report dtc/06-03-02, Object Management Group, 2006.

[19] K. A. Ross and C. R. Wright. *Discrete Mathematics*. Prentice Hall, 5th edition, 2002.

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

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

[22] I. Sommerville. *Software Engineering*. International Computer Science. Pearson Education Limited, 7th edition, 2004.