An ARD+ Design and Visualization Tool-Chain Prototype in Prolog

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

Abstract

The paper presents a prototype design tool-chain, called VARDA for the ARD+ conceptual design method for rules. The tool-chain is implemented in a Unix environment with the use of Graphviz visualization tool and SWI-Prolog. It supports the hierarchical design feature of the ARD+ method, as well as an automated, on-line visualization. This solution allows for rapid prototyping of ARD+ models, that can be integrated with the XTT-based rules. Tool-Chain features are presented using a complete rule-based system example.

Introduction

An effective design support is a complex problem. It is related to the design methods, as well the as the human-machine interface. What is often not emphasized, is the role of the design *process*. Since most of the complex designs are created *gradually*, and are often *refined* or *refactored*, the design method should take this process into account, and the tools used should effectively support it. This issue is important in both software engineering (SE) and knowledge engineering (KE).

It is worth noting, that the most common design method used in the SE, which is UML, does not support the process at all. This makes it difficult to use in real-life for both SE and KE (Nalepa & Wojnicki 2007). Most UML tools provide only a limited undo function, which is very rarely combined with real project versioning. This task is usually delegated to some other tools in the userspace, such as CVS. But the design tool is still unable to actually *present*, and *visualize* the design change. This is also related to the fact, that UML has no facilities to express the design process.

The fact is, that in real life, the *design* (as a process of building the design) support is much more important then just providing means to visualize, 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 as the *design* is often considered a kind of structure with procedures needed to build it (that is at least 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. These methods and tools should be integrated, and provide a hierarchical design process, that would allow for building a hierarchical model, containing the subsequent design stages. Currently HeKatE supports the preliminary *conceptual design* with the ARD+ method (Attribute Relationships Diagrams). The main, so-called logical design, is conducted with the use of XTT method (eXtended Tabular Trees). While originally these methods were developed with the design of knowledge-based systems in mind, they are currently being enhanced to support SE.

In this paper the ARD+ design method is shortly presented. The main focus of the paper is to present the prototype of VARDA, the Visual ARD Rapid Development Alloy. VARDA is a rapid prototyping environment for ARD+, built 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 visualization of the prototype at any design stage. The paper discusses VARDA architecture, focusing on the Prolog implementation, and interface to Graphviz. The features of VARDA are presented using a complete rule-based system design example from (Negnevitsky 2002). Since implementation of VARDA is in an early stage, directions for future work are also given.

Conceptual Design for Rules with ARD+

The original ARD method has been presented in (Nalepa & Ligęza 2005a; Ligęza 2006). It is a supportive design method for XTT (Nalepa & Ligęza 2005b). XTT is a logical design method for rule-based systems, where the knowledge base is designed using a structured representation, based on the concept of tabular trees (Ligęza, Wojnicki, & Nalepa 2001). ARD provides means of attribute identification for the XTT method. The evolution of XTT, as well as larger complexity of systems designed with it, gave motivation for the major rework, and reformulation of ARD, presented for the first time in (Nalepa & Wojnicki 2008).

Below, some basic concepts on which ARD+ is based are presented. They are in fact a refinement of the concepts of the original method. The refined ARD+ method is still referred to as simply ARD in this paper.

The ARD method aims at capturing relations between *attributes* in terms of *Attributive Logic* (Ligęza 2006; Ligęza

& Nalepa 2007). *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.

There are two kinds of attributes adapted by ARD: *Conceptual Attributes* and *Physical Attributes*. A *conceptual attribute* 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. A *physical attribute* is an attribute describing a well-defined, atomic aspect of the system. Names of physical attributes are not capitalized, e.g. `theWaterLevelInTank1`. Physical attributes cannot be finalized, they are present in the final rules capturing knowledge about the system.

There are two transformations allowed during the ARD+ design. These are: *finalization* and *split*. *Finalization* transforms a simple property described by a conceptual attribute into a property described by one or more conceptual or physical attributes. It introduces a more specific knowledge about the given property. *Split* transforms a complex property into a number of properties and defines functional dependencies among them. No new attributes can be introduced upon *splitting*, all attributes from the source property have to be used in the destination properties. Attributes are unique, the same attribute cannot describe more than a single property. The number of transformations in a single design step is limited to *one per node*. It means that a node can be either *split* or *finalized* but not both.

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 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.

Prolog Prototype

A software prototype providing the ARD+ design and visualization method has been built. It is designed as a multilayer architecture (see Fig. 1):

- knowledge base to represent the design,
- low-level primitives: adding and removing attributes, properties and dependencies,

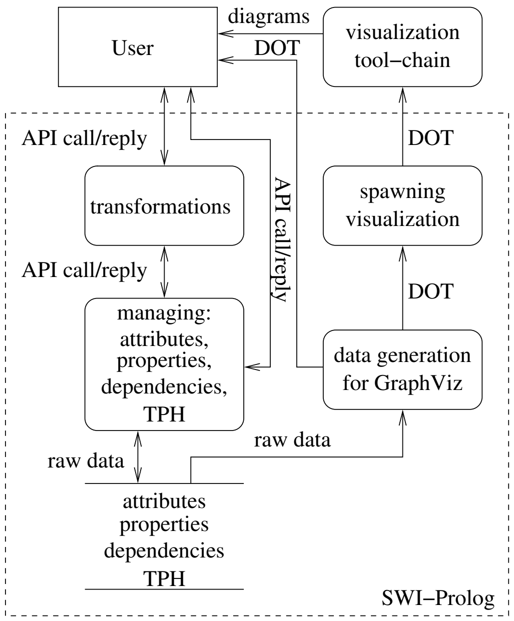

Figure 1: Prolog prototype architecture

- transformations: finalization and split including defining dependencies and automatic TPH creation,
- low-level visualization primitives: generating data for the visualization tool-chain, so-called DOT data,
- high-level visualization primitives: drawing actual dependency graph between properties and the TPH.

As an implementation environment of choice the Prolog language is used (Bratko 2000). It serves as a proof of concept for the ARD+ design methodology and prototyping environment. 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.

Attributes, properties, dependencies and TPH (knowledge base) are represented as Prolog facts, using dynamic knowledge modification capability. The low-level primitives are listed below:

`ard_att_add(+Attribute)` adding the given attribute.[1]

`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,

[1]Marks at the arguments mean: `+`: the argument has to be given, it serves as input, `?`: the argument does not have to be given, it serves as input, output or both, which complies with Prolog language documentation notation.

`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_transform_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 the visualization tool-chain. The tool-chain 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`).

The low-level Prolog predicates described below, generate input for GraphViz. 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 complete directed graphs,

`dotgen(+Edge,+Node)` like `dotgen(+Edge)` but 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. The tool-chain renders appropriate graphs representing the diagrams. The primitives are:

`sar` displaying the current ARD, and

`shi` displaying the 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 in parallel with the shell.

Automated Visualization

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

Both ARD and TPH diagrams are directed graphs. Therefore, a graph visualization primitives are needed. Proper graph visualization, node distribution, edge distribution and labeling is a separate scientific domain (Ross & Wright 2002). Instead of reinventing these concepts, or implementing them from scratch, a tool-chain of well proved tools to provide actual visualization is assembled. The tool-chain is based on three components:

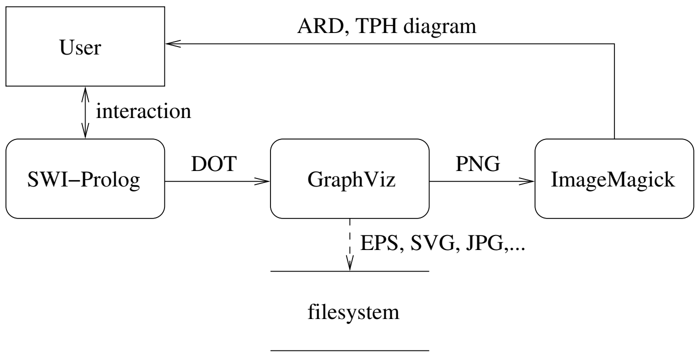

Figure 2: Visualization tool-chain

- SWI-Prolog (`www.swi-prolog.org`),
- GraphViz (`www.graphviz.org`),
- ImageMagick (`www.imagemagick.org`).

An interaction between the visualization tool-chain and the rest of the system is given in Fig. 1. The detailed interaction between the tool-chain components is given in Fig. 2.

SWI-Prolog is an advanced ISO-compliant compiler and interpreter, available under the terms of the LGPL license. It is widely used in research and education as well as for commercial applications. 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 which generate GraphViz readable code from knowledge describing the ARD.

GraphViz 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, so-called DOT data. GraphViz parses the source file and generates a visual representation of the structural information the file consists of. It takes 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.

ImageMagick is a GPLed software suite allowing to create, edit, and compose bitmap images. It can read, convert and write images in a variety of formats. It also allows to modify and transform bitmaps. ImageMagick functionality is typically utilized from the command line or from programs written in almost any programming language.

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, 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 (`display`) which displays it and allows for annotation. In addition to the functionality described above, GraphViz can be successfully applied to generate the diagrams in other formats and store them in the file system. It is indicated as a dotted flow between `GraphViz` and `filesystem` in Fig. 1.

Generating diagrams during the design process is provided by two Prolog predicates: `sar` and `shi` 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.

An Example Design

The ARD process can be easily explained using the following example. It is a reworked *Thermostat case*, found in (Ligęza 2006; Nalepa & Ligęza 2005a). The main problem described there is to create a temperature control system for an office.

The design process is shown in Fig. 3. 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 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:

```
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']]
  ]).
```

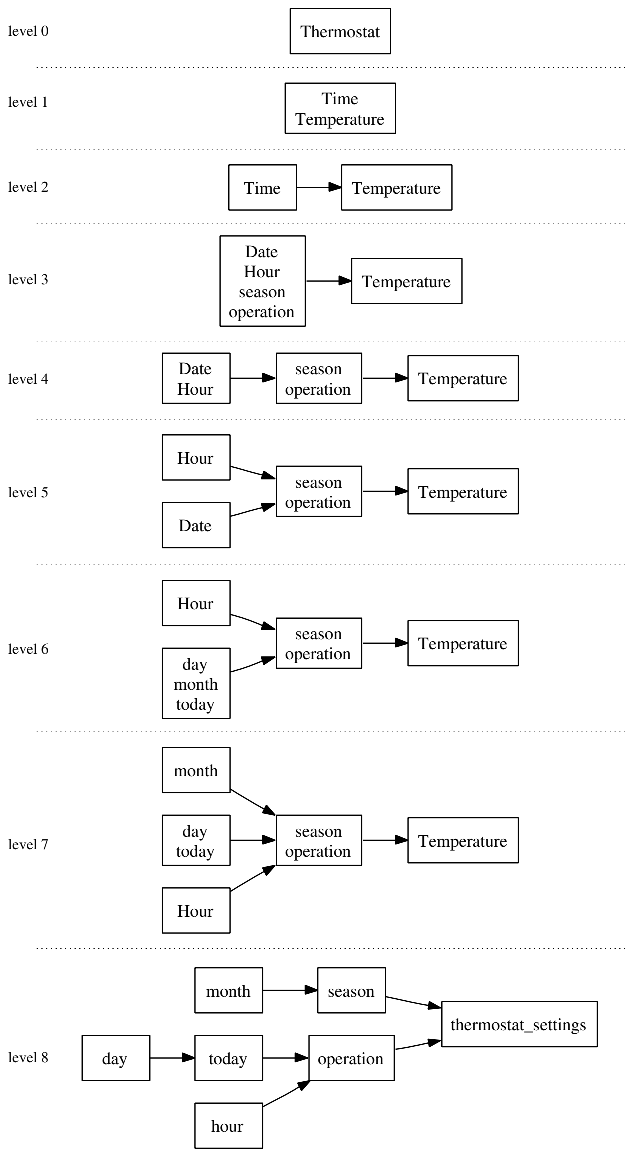

Figure 3: Thermostat example design

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 they mean, in terms of their representation and perhaps further processing, whereas `season` and `operation` are physical attributes. Their domains and constraints are defined to be used subsequently by the XTT process (Nalepa & Ligęza 2004) (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 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:

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

There are functional dependencies defined at level 7:

```
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.

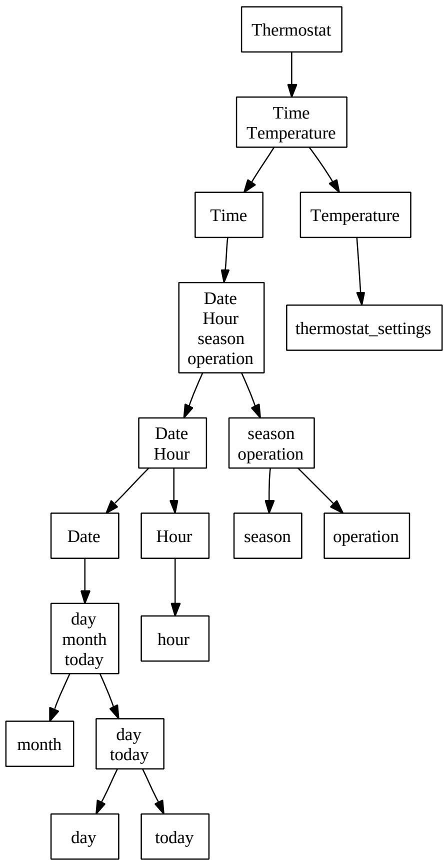

Figure 4: Thermostat TPH diagram

A complete Transformation Process History diagram is given in Fig. 4. It takes into consideration every transformation. Transformations are expressed by directed edges. 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 multiple vertices in the diagram. At any design step current ARD and TPH diagram can be visualized by triggering appropriate predicate.

An example GraphViz source file (DOT file) is given below. It represents level 5 in Fig. 3.

```
digraph {
  rankdir=LR
  node [shape=box]
  "season\noperation\n" -> "Temperature\n"
  "Date\n" -> "season\noperation\n"
  "Hour\n" -> "season\noperation\n"
}
```

The DOT file format is a powerful yet simple text format describing the diagram. GraphViz offers number of advanced features allowing for different graph visualizations.

Future Work

The original contribution of this paper is the presentation of a Prolog-based prototype tool-chain for the refined ARD+ method. The tool-chain uses the Graphviz visualization tool and the SWI-Prolog environment. It allows for a rapid prototyping of the ARD model, with an automated, real-time visualization. The tool-chain is presented using a classic thermostat example from (Negnevitsky 2002).

The tool presented in this paper is just a part of an active research within the HeKatE project. The project aims at developing a range of design, implementation, and analysis tools, basing on the XTT and ARD. Future work will be carried out on both the conceptual and practical level. On the conceptual level fully formalizing the transition from ARD+ to XTT is a priority. Another research aims at translating and representing the ARD knowledge base with the use of XML, through the ATTML and ARDML formats. This should provide means for tool integration between ARD and XTT editors. In order to allow exchange with other attribute specification formalisms, translations from ATTML and ARDML to other formats are considered; these include R2ML, RDF, and ontologies. Depending on the target language different translations are considered. Ontologies may be a source of attributes, where as R2ML, as rule language is considered on the attribute level only. The RDF description may be both applied to the model, as well as attributes only. These translations can be carried out using dedicated XML parsers, Prolog-based parsers, or pure XSLT.

On the practical level, several tools are 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. The next step is developing a visual design environment for ARD is also considered. Several implementations are under consideration, including a Prolog native, possibly XPCE-based (the SWI-Prolog GUI library), as well as Gtk-Server-based (`http://www.gtk-server.org`), or Java-based one.

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

References

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

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

Ligęza, A.; Wojnicki, I.; and Nalepa, G. 2001. Tab-trees: a case tool for design of extended tabular systems. In et al., H. M., ed., *Database and Expert Systems Applications*, volume 2113 of *Lecture Notes in Computer Sciences*. Berlin: Springer-Verlag. 422–431.

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

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

Nalepa, G. J., and Ligęza, A. 2005a. Conceptual modelling and automated implementation of rule-based systems. In Krzysztof Zieliński, T. S., ed., *Software engineering : evolution and emerging technologies*, volume 130 of *Frontiers in Artificial Intelligence and Applications*, 330–340. Amsterdam: IOS Press.

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

Nalepa, G. J., and Wojnicki, I. 2007. Using uml for knowledge engineering 2̆013 a critical overview. In Baumeister, J., and Seipel, D., eds., *3rd Workshop on Knowledge Engineering and Software Engineering (KESE 2007) at the 30th annual German conference on Artificial intelligence : [September 10, 2007, Osnabrück, Germany]*, 37–46.

Nalepa, G. J., and Wojnicki, I. 2008. Towards formalization of ard+ conceptual design and refinement method. In *FLAIRS2008*. submitted.

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

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