Towards Formalization of ARD+ Conceptual Design and Refinement Method

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 discusses a proposal of a formal description of ARD+ a conceptual design and hierarchical refinement method for knowledge-based systems. Formalization of method syntax is given, as well as description of its semantics. ARD+ is a rework of the classic ARD presented elsewhere, that provides a conceptual design for rule-based systems. ARD+ is based on the concept of gradual design where consecutive design steps correspond to different knowledge abstraction levels in the design. Application examples, along with directions for future research are given.

Introduction

Providing an effective development methodology for complex systems remains a challenge, especially in software engineering. In some cases to build conceptual models of business logic software engineers apply Knowledge Engineering methods. These methods originate from the design of knowledge-based systems (KBS), an important class of intelligent systems in AI (Russell & Norvig 2003). In AI, *rules* are probably the most popular choice for building knowledge-based systems, that is the so-called rule-based expert systems (RBS) see (Ligęza 2006).

Applied Software Engineering (SE) is derived as a set of paradigms, and procedures, from practical programming. Historically, when the modelled systems became more complex, SE became more declarative, in order to model the system in a more comprehensive way. It made the design stage independent of programming languages which resulted in number of approaches. So, while programming itself remains mostly sequential, designing becomes more declarative. Since there is no direct bridge between declarative design and sequential implementation, a substantial work is needed in order to turn a design into a running application. This problem is often referred to as a *Semantic Gap* between the design and the implementation (Mellor & Balcer 2002).

The research presented in this paper aims at providing a new software development methodology based on formal RBS design and analysis methods. It is a continuation of the research in the field of RBS, where the logical design method called XTT (*eXtended Tabular Trees*) has been developed (Nalepa & Ligęza 2005b). Subsequently, a conceptual design method called ARD (*Attribute Relationship Diagrams*) has also been proposed (Nalepa & Ligęza 2005a). These methods provided a proof of concept for an integrated design process for simple RBS. This paper presents a proposal of *ARD+*, a largely enhanced and reworked version of the original ARD. Moreover, it provides fully formalized description of the method syntax compared to the original description of ARD found in (Nalepa & Ligęza 2005a; Ligęza 2006). In this paper a formalized notion of diagram transformations is also presented. The ARD+ method allows for *gradual refinement* of the conceptual design with a fully hierarchical system model being built. The method is based on graphs, so a discussion of the related research is also given. Since it is a work in progress, directions for the future development are presented in the final section.

Design Process for Rules

The methods and tools discussed in this paper are aimed at practical design of rule-based expert systems, as well as business rules. Ultimately they should also be applicable to general business software development. This research is carried out within the HeKatE project (hekate.ia.agh.edu.pl). These methods should support the design from the system description in the natural language, through a formalized multilevel design, to a final, automated implementation. The process should be supported by intelligent software tools helping in the design, as well as translation and integration of the model with other solutions.

The HeKatE approach is based upon a concept of a three-level hierarchical design, including conceptual, logical, and physical stages. At each stage another design method is used. The stages are integrated, with the integration formally described, in order to evade well-known *semantic gaps* that are common in the software engineering. The project is based on the previous experiences with design and implementation of RBS in the area on AI (Liebowitz 1998).

Currently, the HeKatE project provides *ARD* (*Attribute Relationship Diagrams*) conceptual design method for attributes, and *XTT* (*eXtended Tabular Trees*) logical design method for rules. While ARD allows to identify system attributes and their functional dependencies, XTT provides means to build decision rules using these attributes. An im-

portant development within the project is the area of supporting tools. At the moment, the ARD+ design is supported by a prototype Prolog toolchain (VARDA), that allows for an automated visualization of the ARD+ model using the *GraphViz* (`www.graphviz.org`) tool.

The original ARD method has been presented in (Nalepa & Ligęza 2005a; Ligęza 2006). It has been proposed as 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 for attribute identification for the XTT method. The first version of ARD (as of (Nalepa & Ligęza 2005a; Ligęza 2006)) was based on the concept of diagrams that contained two sets of attributes, dependent and independent ones. The ARD model was a hierarchical one, with the hierarchy developed with use of two split transformations, the vertical one – specifying the dependency, and horizontal one discovering an independency. This provided a clear and transparent formalism. However, this formalism proved to be quite limited, when it came to more complex design cases. Some of the most important limitations, were:

- insufficient split expressiveness – in complex cases the two basic transformations were not satisfactory, and the ARD formalism would not allow to define other splits,
- unclear attribute origins – the original method would allow to introduce new attributes during the design process without means to formally relate them to existing attributes, making the global attribute model inconsistent, and possibly incomplete (see Fig. 1),
- representation redundancy – there was a notion of functional attribute dependency (which had a clear formulation) and an apparent functional diagram dependency in the model, which were unclear and misleading; at the same time it would enforce attribute redundancy in the neighbor diagrams (see Fig. 1),
- lack of formally described XTT integration – even though ARD supported the XTT, the transition from the ARD model to the XTT was not fully formally described, and suffered from particular notation problems.

These shortcomings stimulated a major rework, and reformulation of the ARD method. At the same time, XTT is also being enhanced towards a general rule programming model.

ARD+ Proposal

The first version of ARD has been applied to simple cases, and had no practical prototype implementation. The evolution of XTT, as well as larger complexity of systems designed, gave motivation for the major rework, and reformulation of ARD, presented for the first time in this paper.

Below, some basic concepts on which ARD+ is based are presented. They are in fact a refinement of the concepts of the original method. ARD+ is a formal method, so the syntax of the method is given along with structural transformations and model semantics. In this paper the refined ARD+ method is still referred to as simply ARD.

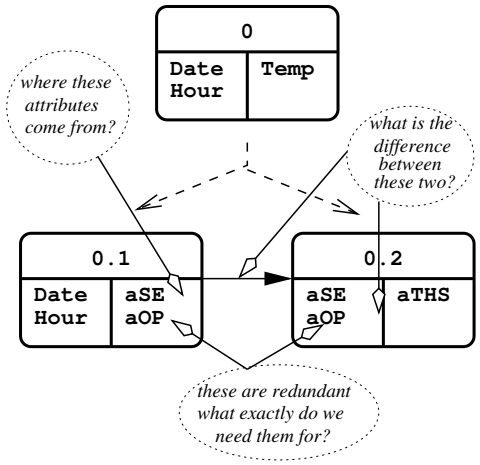

Figure 1: Selected ARD limitations

Preliminary Concepts

The ARD method aims at supporting the designer at a very general design level, where the conceptualization of the design takes place. ARD is used in the first design stage, being a type of *requirements specification* method. Its *input* is a general system description in the natural language. Its *output* is a model capturing knowledge about relations between 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 (Ligęza 2006; Ligęza & Nalepa 2007).

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.

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

Based on these concepts, a new enhanced and formalized version of the ARD is given. Let us now discuss the formal syntax of ARD diagrams.

Syntax

The ARD method aims at capturing relations between *attributes*. *Attributive Logic* (Ligęza 2006; Ligęza & Nalepa 2007) (AL, for short) is one based on the use of *attributes* for denoting certain system *properties*. A *property* is described by one or more attributes. ARD+ captures *functional dependencies* among these *properties*. Capturing these concepts and dependencies is called the *ARD Design Process*.

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 (Russell & Norvig 2003; Hopgood 2001).

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* domains*).*

An attribute (see (Ligęza 2006; Ligęza & Nalepa 2007)) A_i is a function (or partial function) of the form

$$A_i \colon P \rightarrow D_i.$$

A generalized attribute A_i is a function (or partial function) of the form $A_i \colon P \rightarrow 2^{D_i}$, 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 a well-defined, atomic aspect of the system.*

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.*

Let us now discuss the formalization of diagram transformations.

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 $T : D_1 \rightarrow D_2$, 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. Finalization TF is a function of the form*

$$TF : P_1 \rightarrow P_2$$

transforming a simple property P_1 described by a conceptual attribute into a P_2, where the attribute describing P_1 is substituted by one or more conceptual or physical attributes describing P_2.

An interpretation of the substitution is, that new attributes describing P_2 are more detailed and specific than attributes describing P_1.

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

$$S : PS \rightarrow \{PS_1, PS_2, \ldots PS_n\}$$

where a complex property *PS is replaced by n* properties*, each of them described by one or more attributes originally describing PS.*

Constraint 2 *Attribute Dependencies. Since PS may depend on other properties $PO_1 \ldots PO_m$, dependencies between these properties and $PS_1 \ldots PS_n$ have to be stated.*

Constraint 3 *Attribute Disjunction. Attribute sets describing features $PS_1 \ldots PS_n$ have to be disjoint.*

Constraint 4 *Attribute Matching. All of the attributes describing PS have to describe properties $PS_1 \ldots PS_n$. No new attributes can be introduced for properties $PS_1 \ldots PS_n$.*

Such an introduction is possible through *finalization* (see Def. 8) only.

Constraint 5 *Attribute Pool. All attributes describing PS_s have to describe properties $PS_1 \ldots PS_n$.*

Constraint 6 *Transformation Limit. The number of transformations in a single design step is limited to* one per node *– a node can be either* split *or* finalized *but not both.*

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 would be modifying any of the existing transformations: *finalization* or *split* in a particular ARD diagram.

Finalization A *Finalization Refactoring* would be adding or removing an attribute, modifying a past finalization. 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. If P_r functionally depends on a set of properties S_p, even transitionally, and they depend on one another only, properties from S_p needs to be removed too. Additionally all attributes which are not being used any more need to 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 described by the introduced attribute and other properties have to be stated as well. This constitutes the *Split Refactoring*.

Split The *Split Refactoring* can take place on its own. It does not have to be preceded by a *Finalization Refactoring*. In such a case it would be changing functional dependencies among already defined properties only. In general, the *Split Refactoring* regards:

- adding or removing properties upon 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 provided in the following ways:

- define dependencies between the property and other existing properties at the *most detailed* diagram level, or
- adjust appropriate past split transformations (at previous diagram levels) to consideration the new property.

In the first case, the dependencies are defined for the most detailed diagram level only. Since there is a hierarchical design process, 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.

Split refactoring cases discussed here are the generic ones. However, this is a complex issue, crucial for the actual design refactoring. Currently this area of development is an intensive work in progress.

Semantics

The semantics of ARD will be explained and visualized using a reworked *Thermostat example*, found in (Ligęza 2006; Nalepa & Ligęza 2005a).

A property is always described by a set of attributes, these attributes uniquely identify the property . The single most important concept in ARD is the concept of the *property functional dependency*. Properties are described by attributes. It means, that in order to determine dependent property attribute values, independent properties attribute values have to be determined first. An example is given in Fig. 2.

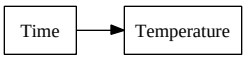

Figure 2: An example of dependency

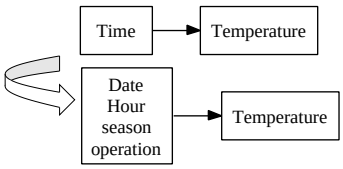

Figure 3: An example of finalization

It indicates that a property described by `Temperature` depends on a property described by `Time`.

Identifying all possible properties and attributes in a system is a very complex task. Transformations allow to gradually refine properties, attributes and functional dependencies in the system being designed. This process ends when all the properties are described by *physical attributes* and all of the functional dependencies 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`. 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 *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`, `operation`

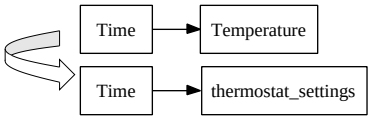

Figure 4: An example of simple finalization

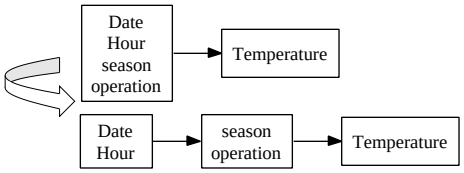

Figure 5: An example of split, dependent

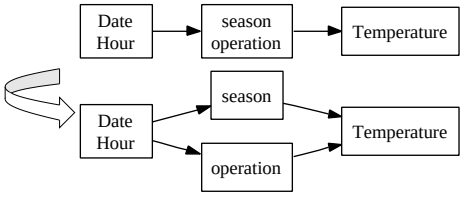

Figure 6: Another example of split, independent

depend on `Date`, `Hour`.

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 features described by *physical attributes* only. 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 a 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.

Hierarchical ARD+ Model

During the design process, upon splitting and finalization, the ARD diagram 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.

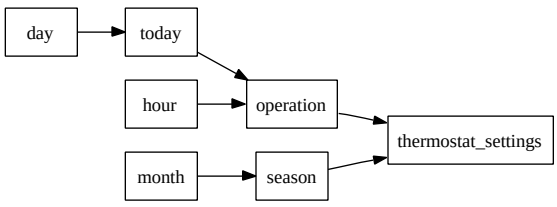

Figure 7: An ARD Diagram: all features are identified

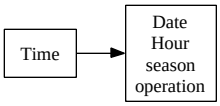

Figure 8: TPH for the diagrams from Fig. 3

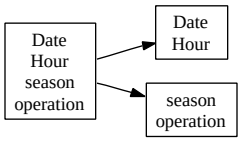

Figure 9: TPH for the diagrams from Fig. 5

The purposes of having the hierarchical model are:

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

The implementation of such a hierarchical model is provided through storing the lowest available, most detailed diagram level, and additionally information needed to recreate all of the higher levels, the so-called *Transformation Process History*, TPH for short. TPH 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, denoting what particular property is split into or what attributes a particular property attribute is finalized into.

An example TPH for the transformation presented in Fig. 3 is given in Fig. 8. It indicates that a feature described by an attribute `Time` is refined into a feature 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 feature described by attributes: `Date`, `Hour`, `season`, `operation` is refined into two features described by: `Date`, `Hour` and `season`, `operation` attributes respectively. Having a complete TPH and the most detailed diagram level, it is possible to automatically recreate any, more general, level, performing a roll-back. Using this feature it is possible to recreate the ARD design at any abstraction level previously defined; it allows further refactoring.

Using the last level of the ARD+ rule prototypes can be generated. The input of the algorithm is the most detailed ARD+ diagram, the output is the set of *rule prototypes* in a very general format:

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

The complete description of the algorithm is out of scope of this paper. The rules for the ARD diagram in Fig. 7, are:

```
season, operation | thermostat_settings
month | season
day | today
today, hour | operation
```

Related Research

The formulation of ARD+ is simpler and more compact, than the original one. To some extent it is influenced by some classic concepts from AI and mathematics.

A first most obvious inspiration is the graph theory. It is a very important and well developed topic in mathematics. The ARD+ formulation should be enhanced in the future, to include some graph notions, especially for transformations, and possibly split patterns. On the other hand it was a conscious design decision *not* to use graph-related syntax names for ARD+. So even though for a mathematician, an ARD+ model is a directed graph, dependencies are edges, and properties are nodes; we try to avoid these "visual" names, simply to indicate their general meaning *within* ARD+.

Graph-based representations can be extended in number of ways. An important formalism, that was also considered are the AND/OR graphs. However, it seems that its semantics is too strong and specific for what is needed in ARD+.

ARD+ also bears similarities to the causal graphs formalism, used mainly in diagnosis and fault detection. At this point ARD+ was invented with forward chaining rule-based systems in mind. However, it is a generic approach for rule prototyping that could also be applied to backwards chaining, which is omnipresent in the domain of diagnosis.

To some extent, the ARD+ is similar to classic AI methods such as semantic networks and ontologies. Again, so far it was a conscious decision *not* to include some of the strong semantical mechanism present in ontologies, such as classic abstraction (though the ARD+ attribute specification is similar) or composition. It seems that at this stage level these tools could be actually too restrictive.

ARD+ attribute finalization concept is similar to a classic concept of *production* in the terms of the formal grammars. Conceptual attributes are semantically close to non-terminal symbols, whereas the physical ones to terminals. So, a more algebraic notation of attribute finalization using BNF, or BNF-like notation is considered.

Summary and Future Work

The paper presents a preliminary formal description of the ARD+ conceptual design method for rules. The original contribution of the paper is the refined formalization of the method, including a gradual hierarchical design process, and a compact hierarchical ARD+ model. The solutions contained in the paper are presented with practical examples.

While ARD+ identifies attributes describing the system, specifying actual rules is provided by the XTT method, building upon attributes specified in the ARD+ process. There is an ongoing research which would allow for an integration of ARD+ and XTT.

Future work will be mainly focused of the more complex design cases, that would allow for developing practical *refactoring* solutions within ARD+. One of the approaches is to formulate semi-formalized *design minipatterns*, used at the ARD+ level transitions. These should ultimately allow to identify and describe some typical split patterns (in the original ARD there has been only two split types possible).

Since the design process should be supported with a convenient tool, there is a prototype ARD+ design tool (VARDA) written in the Prolog language. It is purely declarative and supports automated visualization at any diagram level. 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.

Currently this prototype design tool stores knowledge as Prolog language clauses. While it is an efficient approach for the design stage, it is not portable in terms of data and knowledge exchange standards. So there is an ongoing research regarding ARD+ representation using XML. This should ultimately allow for attribute exchange with other specification formalisms, including RIF on the general level, R2ML, and possibly ontologies.

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

References

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

Liebowitz, J., ed. 1998. *The Handbook of Applied Expert Systems*. Boca Raton: CRC Press.

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.

Mellor, S. J., and Balcer, M. 2002. *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

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.

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