Dissertation

FuzzyARDEN: Representation and Interpretation of Vague Medical Knowledge by Fuzzified Arden Syntax

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter der Leitung von

Univ.-Prof. Dipl.-Ing. Dr. techn. K.-P. Adlassnig
Abteilung für Medizinische Experten- und Wissensbasierte Systeme am
Institut für Medizinische Computerwissenschaften

eingereicht an der Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

Dipl. Inform. Sven Tiffe
Sven.Tiffe@web.de

Wien, im Januar 2003

Zusammenfassung

Die Wissensrepräsentation ist ein grundlegender Forschungsbereich im Gebiet der Künstlichen Intelligenz. Ein Ansatz zur Repräsentation medizinischen Wissens liegt in der "Arden Syntax for Medical Knowledge Systems". Mit der Arden Syntax können modulare Entscheidungsregeln definiert werden, deren zweiwertige Entscheidungslogik durch Algorithmen beschrieben wird.

Medizinisches Wissen enthält häufig Unsicherheiten im Sinne linguistischer Vagheit. Soll vages Wissen durch Regeln repräsentiert werden, welche auf binären Entscheidungen basieren, kann es in Grenzfällen vorkommen, dass die Ergebnisse nicht mehr intuitiv nachvollziehbar sind. Durch Anwendung der Fuzzy Mengentheorie und der Fuzzy Logik kann vages Wissen präzise repräsentiert und verarbeitet werden. Diese Arbeit stellt Ideen und Lösungen vor, mit denen die Arden Syntax um Konzepte der Fuzzy Mengentheorie und der Fuzzy Logik erweitert wird, um Regeln mit inhärenter linguistischer Vagheit adäquat darstellen zu können.

Ein Grundprinzip der Fuzzy Mengentheorie, der graduelle Zugehörigkeitsgrad eines Elementes zur Menge, wird auf konditionale Elemente der Arden Syntax angewandt. Konditionale Ausdrücke, die häufig Teil der Entscheidungslogik sind, nutzen normalerweise Vergleichsoperatoren, welche eine Bedingung durch vordefinierte Grenzwerte modellieren und einen Booleschen Wahrheitswert ermitteln. Durch die Erweiterung ausgewählter Vergleichsoperatoren um eine graduelle Erfüllbarkeit der Bedingung ergeben diese Operatoren in Grenzfällen einen unscharfen, mehrwertigen Wahrheitswert. So können Wissenskonzepte mit vagem Definitionsbereich durch "unscharfe" Vergleiche repräsentiert werden.

Die sich aus der Nutzung unscharfer Wahrheitswerte ergebenden Probleme von 'Fuzzy Algorithmen' werden behandelt und durch eine funktionale Erweiterung von Entscheidungselementen der Arden Syntax sowie durch eine Erweiterung des Datenmodells gelöst. Die Einbindung des Konzeptes 'linguistischer Variablen' schliesst die Erweiterungen der Arden Syntax ab. Variablen dieses Typs abstrahieren die Beziehungen zwischen linguistischen Ausdrücken und numerischen Messwerten und können den intuitiv lesbaren Anteil der Entscheidungslogik erhöhen.

Die vorgestellten Erweiterungen werden in einer Java basierten Arden Syntax 'rules engine' implementiert, welche "fuzzifizierte" Arden Syntax Regeln lesen und ausführen kann. Die 'rules engine' wurde in zwei klinischen Projekten verwendet. Die Wissensbasis des Expertensystems CADIAG-II zur Unterstützung der Differentialdiagnostik in der internen Medizin wurde erfolgreich in Fuzzy Arden repräsentiert. Ein Fuzzy Control basierter Klassifizierer zur Erkennung von Glaukom wurde durch Verwendung von linguistischen Variablen implementiert.

Abstract

The representation of knowledge is one of the fundamental research areas in the realm of artificial intelligence. One approach to this subject was provided by the Arden Syntax for Medical Logic Systems. The Arden Syntax is a hybrid knowledge representation format that allows one to define rules as single modules that use algorithms in their conditional part. It includes typical procedural programming language elements used to define a crisp decision logic based on binary decisions.

However, medical knowledge usually contains uncertainty in terms of linguistic fuzziness. If such knowledge is represented by rules that require binary decisions, the results of the rules might be unintuitive when they touch borderline cases. As such uncertainty can be precisely represented and manipulated by concepts of fuzzy set theory and fuzzy logic, this thesis presents ideas and solutions for extending Arden Syntax by concepts of fuzzy set theory, fuzzy logic, and fuzzy control to adequately represent rules with inherent linguistic uncertainty.

Fuzzy sets define a gradual degrees of membership of their elements to the set instead of crisp ones. This principle is applied to conditional expressions that are part of the decision logic: Such expressions usually include comparison operators that compare a fact to predefined thresholds. With traditional Arden Syntax these conditional expressions return a Boolean truth value; by applying the concept of a gradual degree of membership (or, in other words, a gradual degree of compatibility of the fact to the condition) to a subset of the comparison operators, the conditional expressions return in borderline cases fuzzy truth values instead of crisp ones. By this, fuzzily defined concepts can be represented by fuzzified comparisons.

The definition of fuzzy truth values has an impact on the algorithmic elements of the Arden Syntax, as operations, such as branches in program flow, may depend on a truth value that is neither ‘true’ nor ‘false’. The problem of fuzzy algorithms is addressed and resolved by extending the functionality of the corresponding statements and by extending the data model of the Arden Syntax. As further extension, the thesis proposes a new type of Arden Syntax module, the ‘linguistic variable’ module. This type defines the relationship of linguistic concepts to real-world facts and can be used to increase the amount of intuitively readable parts of the decision logic in an Arden Syntax module or to realize fuzzy control knowledge bases.

All extensions have been realized by a Java based Arden Syntax rules engine that is able to read and execute the fuzzified Arden Syntax modules. It has been used for two clinical projects: The knowledge base of the existing expert system CADIAG-II for diagnostic decision support in internal medicine has been successfully realized by Fuzzy Arden. The syntax has been also used to realize a fuzzy control based glaucoma classifier.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I wish to thank my parents Anna Maria and Ingo without whom I would have been unable to write these words here - in every sense of the term.

Of course I also wish to thank all friends and colleagues who helped me in writing this thesis.

I am indebted to Professor Klaus-Peter Adlassnig from the University of Vienna for his expert guidance of this thesis and his persistence - he never stopped teaching me until he was absolutely certain I had understood everything he wanted me to know. Thank you very much for your patience (and, occasionally, your persistence). I was indeed able to learn a lot from you.

I'd also like to thank Dr. Gudrun Zahlmann and Mr. Werner Striebel from Siemens Medical Solutions for their subject-based as well as organizational support (and of course for their complete financial assistance) of my paper. The same applies to Dr. Siegfried Schneider, Dr. Harm Scherpbier and Mr. Pat Lyons of Siemens Medical Solutions for their willingness to discuss various subjects with me as well as their organizational support. These persons assisted me on both sides of the Atlantic (their pragmatic viewpoint, in part, skilfully steered me from my flights of fancy to the facts of reality).

I received subject-based assistance and also answers to my numerous gnawing questions from Ms. Andrea Rappelsberger and Mr. Dieter Kopecky at the University of Vienna - their contributions ranged from correction and constructive criticism of my paper to the resolution of irregularities in the knowledge bases of our expert systems.

Without the colleagues at HL7 I would have missed out on a lot of inspiration. The discussions at our meetings validated me for my work and also provided the right measure of criticism and suggestions for improvement. Special thanks to my first contact persons, Dr. Robert Jenders and Dr. John Dulcey, as well as Dr. Mike Jones, for their active interest in my work. Of course, a big thank you to all the other members of the Arden Syntax Special Interest Group and HL7 for their contribution to Fuzzy Arden and for their "post-working-hour program" at our meetings.

Without the colleagues at my department in Siemens Medical Solutions I probably would have lacked the inspiration, but I certainly would have missed out on the pleasure of my work. Sincere thanks to my colleagues and co-students at GT/BD/THS for creating a congenial atmosphere at the office, for the many laughs and the delicious coffee.

Finally I wish to thank the person who had to put up with me at home, who gave me her back-up and instilled confidence, and especially gave me a lot of support over the last few months - my dear Juliane. A great many things would have been difficult to tolerate without you. I'm glad you were by my side.

My most heartfelt thanks to all of you.

Chapter 1

Introduction

Daily work in health care requires an ongoing assessment of facts and decisions in diagnostic and therapeutic processes. Medical staff are often compelled to overlook a large body of facts owing their very magnitude. As result, decision making is rendered difficult by incomplete or erroneous information. The workload is further increased by new methods of therapy and observation, such as monitoring the data of patients with chronic diseases. Computer-based systems that support decision-making by automatically processing huge quantities of data could assist specialists in this everyday task and reduce the burden on clinical staff. Uncommon situations that exceed the confines of daily routine may be supported by systems that make the knowledge and experience of experts accessible to non-experts.

In the past decades the development of such *knowledge-based systems* or, more specifically, medical *expert systems*, made significant progress. Knowledge-based systems are part of the science of artificial intelligence (AI), which has been variously defined in the past 50 years.

In an early report in 1963 Minsky defined AI as "the science of making machines do things that would require intelligence if done by men". In this early stage of artificial intelligence, which can be characterized as '*power-based approach*, one goal was to achieve very general and domain independent methods for solving problems by knowledge-based systems [GRS00].

As such generalized systems could not be easily realized, in the beginning of the 70s the focus in AI shifted to the development of systems that work with domain specific knowledge (the '*knowledge-based approach*). Such systems are generally computer programs characterized by the use of an inference mechanism and a knowledge base that explicitly represents the knowledge which is required for solving a specific problem. In contrast to knowledge that is represented implicitly as part of the program code, an explicit knowledge base can be redefined, extended, or exchanged between the knowledge-based systems without having to alter their program code [GRS00].

To create such a knowledge base it is necessary to define a formal knowledge representation that preferably is in a human understandable form and enables the system to act as if it "knew" what to do. Brian Smith formulated these requirements as the "*knowledge representation hypothesis*:

> "Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits,

> and b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge." [Smi85]

Knowledge representation formats may be regarded as descriptions of formal models[1] that should be interpretable by machines as well as by humans. They include formal constructs for the representation of knowledge and can (implicitly) include instructions as to how the represented models can be evaluated. These formats therefore play a central role in any knowledge-based system.

Different representation formats for knowledge have been developed over the years. The choice of a specific knowledge representation format for solving a given problem may directly influence the simplicity of doing it. For instance, humans may find the multiplication of two Arabic numerals easier than the multiplication of the same numbers represented in Roman script, whereas a software-based system could easily solve both problems as long as the necessary algorithms are formalized. On the other hand, natural language can be easily interpreted by humans, whereas it still is a sophisticated task for a computer. For instance, it is not difficult for humans to summarize a short novel while it would be a task of significant magnitude for a software. Thus, a knowledge representation format that seems to be perfect for one specific problem and one specific audience might be less applicable for other problems or other audiences.

This thesis deals with one specific format that has been explicitly developed to model medical knowledge, namely the "*Arden Syntax for Medical Logic Systems*. The Arden Syntax was designed as a format that can be easily understood by persons who are not computer experts or programmers, yet to define (and understand) formal medical rules such as "if the patient has an allergy to penicillin, never administer this drug". Generally the formalized rules are evaluated in cases of pre-defined events, to decide whether the current situation requires further actions or not. The result is therefore usually either 'true' or 'false'.

However, in contrast to industrial systems where products are nearly identical to their templates and production processes can be described quite accurately, medical procedures such as the definition of diseases and the process of diagnosing or the definition and application of therapies largely include uncertain knowledge. Medical knowledge itself is often described originally by natural language and therefore includes linguistic vagueness. Furthermore, findings can be uncertain if they depend on descriptions given by the patient. Owing to the uncertainty and vagueness of knowledge, many facts cannot be properly formalized by conditions that finally yield 'true' or 'false'. The present paper defines extensions for the Arden Syntax that include concepts of fuzzy set theory and fuzzy logic. These extensions can then be used to model uncertain knowledge and to create reasoning processes that are closer to human reasoning.

1.1 Knowledge representation

As knowledge-based systems assist humans in making decisions, often the term 'decision support' is used as synonym for the use of expert systems. For example, users of an expert

[1] A model is usually used as a theoretical construct, for example a simplified representation of facts. Formal models are models which can be described by a formal language. In addition, operational models are models which can be used to execute operations associated with the represented facts (a formal model may be an operational one).

system can be advised to solve problems that might exceed their personal experience, such as to diagnose a rare disease. Daily routine tasks, such as monitoring vital parameters of patients who suffer from chronic diseases, can be executed by an expert system that notifies the physician only in those cases in which his personal experience and his abilities as a human being are required[2]. A knowledge-based system may passively control processes within an institution and provide additional information if the processes are not fully under control.

The terms 'knowledge-based system' and 'expert system' are not consistently defined and used in the literature. A possible definition could be:

Definition 1 (knowledge-based system, expert system):

> *A* **knowledge-based system** *is a computer-based system that includes explicit knowledge represented by a fixed knowledge representation format and an inference machine, which can infer new knowledge or conclusions from the stored knowledge. In particular, the inference machine works independently from the contents of the knowledge base, which need not be limited to a certain domain.*
>
> *The term* **expert system** *is commonly used for knowledge-based systems that show characteristics derived from the behavior attributed to people who are experts in their domain [BC90]. The knowledge of expert systems is on a higher level than the knowledge of a (human) novice [Win92, GRS00]. They should be able to explain the results of their inference process in a way that can be understood by human users [AS00]. Their knowledge bases often consist of facts (formalized statements on properties of objects) and rules (formalized conditional statements) [Rei91, Kan92].*

However, as a consistent distinction between expert systems and knowledge-based systems is not required for this work, the two terms will be used synonymously.

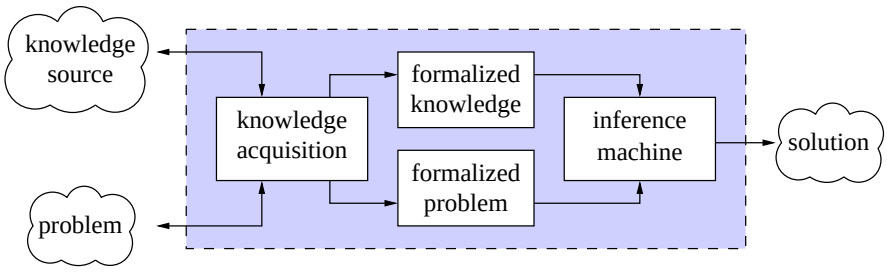

Figure 1.1: Model of the architecture of a knowledge-based system (following [Bib93])

One general structure of a knowledge-based system is shown in figure 1.1. A knowledge-based system acquires external knowledge, computes a conclusion by using the inference machine, and returns the result to external recipients. The common definition of *knowledge*

[2] As remarked earlier, today knowledge-based systems are usually domain specific. Although they can reach a high level of knowledge within their own domain—for instance CADIAG-II (which will be described in detail later) comprises knowledge about more rheumatological diseases than a non-expert physician usually does—such systems are still limited to the domain. Knowledge-based systems usually do not know what they do *not* know. A human user has the advantage of possessing more knowledge on the one hand and the ability to view a problem from a higher abstract level on the other hand.

acquisition is limited to the definition and maintenance of the knowledge base, either by human experts who enter their knowledge explicitly or automatically by the system itself (for instance by statistical analysis of patient data). In addition to the knowledge base that defines how facts relate to each other, the specific problem also has to be formalized. For example, if the knowledge base is composed of rules that use symbols to represent dynamic facts (such as "if the blood glucose level is significantly elevated"), all external facts have to be acquired by formalizing it. This knowledge acquisition step is termed the *data to symbol conversion*.

Definition 2 (data to symbol conversion):

> *The* **data to symbol conversion** *represents facts of the physical world by assigning formal symbols. If the conversion does not yield a one-to-one mapping and a set or range of facts is mapped to a symbol, it might include an abstraction of knowledge.*

Once the required knowledge and the individual problem have been formalized, the inference machine combines both information and computes a result that should solve the problem. As the result may be represented in a formal way it has to be converted to a form that can be understood by external receivers, for instance into natural language messages, which can be easily understood by humans.

Natural language in spoken or written form is the oldest and most common representation format of knowledge[3]. Although natural language is highly expressive, knowledge represented in natural language might not be the optimal choice for computer-based processing of knowledge.

The main problem of using natural language as a computable knowledge representation format is the ambiguity of statements because speech, a special type of language, may have other meanings in addition to its main one. The main meaning can be described independent of a special context (such description is a matter of semantics). Additional meanings may be the result of contexts in speech, where the meaning depends directly on the dialogue, for example when using idioms. In this context it is relevant to distinguish the meaning which was intended by the speaker (or writer) and the meaning which was interpreted by the listener (or reader)[4] [Hel01].

The ambiguity of speech can be rendered unambiguous during a conversation between humans by the context or the implicit background knowledge of the speaker and listener. However, even if the context is defined and background knowledge is available (and the sentence being analyzed is syntactically correct), the overall meaning from the individual semantic meanings of the single terms and combinations could be subject to different interpretations.

Furthermore, one fact can be represented in several ways. Intuitively 'Every philosopher is a human' represents the same thought as 'All philosophers are humans', but formally it requires the definition that 'Every A is B' equals 'All As are Bs'. Another example is the expression 'The Venus is the evening star' that has a different intuitive meaning than 'The Venus is the Venus' (apart from the fact that Venus is a planet and not a star).

A formalism requires that a fact (or thought) is always represented by the same expression of symbols [Łuk51]. By this, the validity of an expression can be proved on the basis of the external form without referring to the semantic meaning of the symbols.

[3] Apart from the mental representation of knowledge.

[4] The analysis of these relationships are a subject of pragmatics or speech act theory [SKB80]

Processing (unstructured) natural language is a separate area of research and is not addressed further in this work. However, natural language is a starting point for other formats of formal knowledge representation in some ways. Presented below are some formats that provide methods to formalize relationships and coherence between facts without necessarily including information about the semantic meaning of facts (and sometime the semantic meaning of the relations). Many of the more specialized knowledge representation formats make use of one or more of these basic methods.

1.1.1 Predicate logic

One way to represent knowledge and to infer conclusions in the way humans would do is to use logical expressions [Rei91]. Anyhow, logic is not the science of human thinking but can be used as a tool to formalize relationships between facts. In Ancient Greece there was a dispute among philosophers as to whether logic is a part of philosophy or just a tool, or both. One answer to this dispute was to treat logic as part of philosophy if syllogisms are defined by concrete terms with an inherent semantic meaning[5]. However, if the syllogisms are defined as pure rules based on symbols (e.g. on letters), then logic is rendered uncertain as an *instrument* of philosophy. The relationship between logic and philosophy is not of great interest for knowledge representation, but the distinction between the semantic meaning of the terms and the meaning of the logical construct is. If the semantic meaning is removed from the logical expression by replacing the terms by symbols, the matter is removed from it and a formal logic remains [Łuk51].

Gottlob Frege developed, in his famous *Begriffsschrift*, a language that was oriented to formal logic, which operates on unambiguous terms and which allows logical consequences to be derived by syntactical rules. He justified his work on the imperfection of natural language to express logical thoughts and to avoid errors in reasoning [Bie97]. Some aspects of this "imperfection" will be the discussed separately in section 1.2 dealing with uncertain knowledge.

Freges formal language is similar to the language of predicate logic. An important characteristic of predicate logic is the independence of the semantic meaning from the syntactical statement. Only the interpretation of the logical expression defines the semantic meaning, regardless of any interpretation given a priori by the labels of the constants and predicates.

As an example, an *atomic expression* such as "Erlangen is a small city" can be represented in predicate logic by

$$\textit{is\_a\_small\_city}(\text{Erlangen})$$

where 'Erlangen' is a constant that represents an object and '*is_a_small_city*' is a predicate that represents a property of an object. Even if the inherent meanings of the terms suggest a certain meaning of the whole expression, its interpretation is limited only in terms of the syntactical difference between predicate and constant. Another legitimate interpretation of the expression might be "Bob has fever" where 'Erlangen' is interpreted as 'Bob' and '*is_a_small_city*' as 'has fever'. To be more independent from the semantic meaning of the terms, constants and predicates are usually represented by neutral labels, such as P as predicate and a as constant—the last example would then be represented as:

$$P(a)$$

[5] Łukasiewicz cites Plato's syllogistic proof of the immortality of the soul as an example [Łuk51]

A predicate with n arguments and n terms that can represent constants or functions but not variables is termed 'atomic sentence'. 'Sentences' are made of atomic sentences that are combined by using expressional logic operators. Predicate logical 'formulas' additionally use variables and qualifiers which may be employed to bind variables [Bib93]. (Other definitions of predicate logic do not use separate definitions for atomic sentences and formulas [Rei91].) For example,

$$\exists a : P(a)$$

could be interpreted as "some cities are small cities". If all variables of a formula are bound, the formula is termed 'closed formula'.

Closed formulas describe objects and represent a truth value, which might be undetermined. As mentioned earlier, an expression $P(a)$ describes the logical relationship between a constant and a predicate; the semantic meaning of predicate logical expressions is the result of a process termed *interpretation*. The interpretation may be formally defined as a function that maps every constant or variable to an object in the *universe of discourse* and every predicate to a truth value 'true' or 'false'. In the last example, the universe of discourse can be defined by the set of all German cities GC. If the predicate P yields 'true' for at least one $a \in GC$, the formula yields 'true'. For the evaluation of the formula, the semantic meaning of the labels must be consistent within the expression. That means that a constant a must have the same meaning, if it appears more than once.

Predicate logic provides, as well as does mathematical logic, the possibility to infer, based on a given set of formulas, new formulas that may represent explicit knowledge that was implicitly represented by the pre-defined formulas. The deduction mechanism of a predicate logic system is defined by a set of axioms and inference rules.

A common inference rule is the 'modus ponens' that allows inference from one formula x to another formula z. If x is valid and another formula y defines the implication of z by x, then the formula z must also be valid. Formally the modus ponens can be defined as:

$$mp(x, y) = \begin{cases} z & \text{if } y \text{ represents the formula } x \Rightarrow z \\ \text{undefined} & \text{else} \end{cases} \tag{1.1}$$

A common notation for such inference rules is the following. The left side shows the formal definition; on the right side the formula y is replaced by the implication.

$$\frac{\begin{array}{l} x \\ y \end{array}}{z} \qquad \frac{\begin{array}{l} x \\ x \Rightarrow z \end{array}}{z}$$

As an example, 'intracellular uric-acid crystals' within the synovia prove gout. The modus ponens can be used for a patient who suffers from intracellular uric-acid crystals to infer the conclusion that he suffers from gout:

$$\frac{\begin{array}{l} \textit{intracellular uric-acid crystals within synovia}(p) \\ \textit{intracellular uric-acid crystals within synovia}(p) \Rightarrow \textit{gout}(p) \end{array}}{\textit{gout}(p)}$$

In formal logic, the inference process is based *only* on the syntactical part of logical expressions, not on the semantic meaning and is therefore independent of the interpretation. A common knowledge representation format that uses predicate logic is Prolog[6].

[6]Prolog is a classical logic programming language. A good starting point online is the 'World Wide Web Virtual Library' at http://www.afm.sbu.ac.uk/logic-prog/ that gives information on Prolog as well as pointers to other logic programming languages.

The most simple way to model knowledge by predicate logic is by a predicate with one argument. Such a predicate can represent knowledge that can be expressed in natural language in adjectival ways. Semantic relationships between constants, formulas or variables can be defined by predicates that use more than one argument.

A two-valued predicate 'is_instance' can be used to represent concept classes and members of concept classes. Concept classes are concepts whose extensions consists of more than one concept. For example, a hummingbird can be a specialized representation of the concept class of birds:

$$\textit{is\_instance}(\text{hummingbird}, \text{birds})$$

Analogously to concept classes, events and actions could be defined by a predicate 'is_fired'. The following sentence yields true if new data is added to the patient record.

$$\textit{is\_entered}(\text{data}, \text{patient\_record})$$

As shown, predicate logic offers a way to formalize natural language and can be used to infer conclusions. However, the representation of one fact, which is described in natural language, using predicate logic, can be defined by more than one expression. There is also no rule as to whether a certain concept or fact should be represented by a constant, by a formula, or by a predicate. This decision has to be made with reference to the proposed use of the knowledge [Bib93].

Classical (predicate) logic formulas yield either 'true' or 'false' and traditionally use the '*closed world assumption*, which assumes that unknown facts can be rated 'false'. Therefore, simple forms of incompleteness such as missing information within the electronic medical record of a patient cannot be represented. Contradictory knowledge cannot be modelled by classical logic, as the inference mechanism cannot derive reliable knowledge from contradictory formulas.

For inference on unknown facts and for the representation of contradictory knowledge, classical two-valued logic has to be extended. For example, epistemic logic can be used to differ facts additionally between "it is known to be true" or "may be believed to be true" [Gel94]. Further, multi-valued logic offers methods to handle vague knowledge by representing, for instance, truth values by numerical values from 0 to 1.

Excursion: many valued logic

As mentioned earlier, expressions based on many valued logics are not restricted to be either 'true' or 'false' but can also represent other truth values. The first many-valued logics were formulated in the early 1920s by Łukasiewicz and Post[7] [Sin70]. Łukasiewicz analyzed modal expressions that represent statements about facts which might be or might not be. He explained his train of thoughts by the following example:

> It may obviously be assumed without any discrepancy that my presence at the city of Warsaw for a given day in future is *possible* but not *necessary*. The modal expression about a future fact "I will be at Warsaw on December 21$^{\text{th}}$" can thus be neither 'true' nor 'false'.

[7] Post formalized his many valued logic independently from Łukasiewicz. Statements can represent truth values from a set of n truth values, where $n \geq 2$. The logical alternative is defined analogous to equation 1.3. The negation is defined as a 'cyclic' negation—the truth value of $\neg\alpha$ is 0 if the truth value of α is n and it is $\alpha + 1$ otherwise.

If the expression would yield 'true', his presence must be, in contradiction to the premise, necessary. If it would yield 'false', his presence must be impossible, which is also in contradiction to the premise. Therefore, the expression can only be "possible", but neither 'true' nor 'false' [Łuk30].

He came to the conclusion that such expressions, formalized as $M\alpha$ = ("it is possible, that α"), cannot be represented by two-valued logic. He proposed the use of a third truth value "neutral" to represent modal statements. Table 1.1 shows three truth tables for the logical operators 'and', 'or', and 'not'. The truth values are represented by 1 for 'true', 0 for 'false', and 0.5 for 'not known'. One consequence of Łukasiewicz' three-valued logic was that neither the law of the excluded middle $\alpha \vee \neg\alpha$ nor the law of non-contradiction $\neg(\alpha \wedge \neg\alpha)$ where a tautology anymore[8]. For $\alpha = 0.5$ both expressions yield 0.5.

Table 1.1: Łukasiewicz first three-valued logical operators truth-tables

and	1	0	0.5
1	1	0	0.5
0	0	0	0
0.5	0.5	0	0.5

or	1	0	0.5
1	1	1	1
0	1	0	0.5
0.5	1	0.5	0.5

not	1	0	0.5
	0	1	0.5

Łukasiewicz remarked that three-valued logic is much closer to human intuition, which is closely connected to the principles of possibility and necessity, than two valued logics could ever be. In the actual context it is not relevant whether his proposals were suitable for defining a modal logic; the interesting fact is that the result was the first explicit formulation of a three valued-logic.

Later he generalized three-valued logic by a four-valued logic and finally by the first logic with infinitesimal truth values [Sin70]. Truth values can be defined by real numbers $[0, 1] \in \mathbb{R}$, where 0 represents false and 1 represents true. The logical operations are defined as:

$$\neg\alpha := 1 - \alpha \tag{1.2}$$

$$\alpha \wedge \beta := \min(\alpha, \beta) \tag{1.3}$$

$$\alpha \vee \beta := \max(\alpha, \beta) \tag{1.4}$$

Even if the representation of knowledge by pure logical expressions may be suitable for mathematicians who have to prove their logical concepts, it might not be the optimal choice for non-mathematicians who are not used to working with large, abstract formulas. For easier understanding, predicate logic expressions can be represented, for instance, by directed acyclic graphs that are also used as representation formalism for semantic nets.

1.1.2 Semantic nets

Semantic nets are historically based on models of human memory in cognition psychologies. Based on studies it is assumed that concepts which have semantic relationships are represented by structures that are connected in a suitable way. These connections are

[8] A tautology is an expression which is true for every α.

termed 'associative relationships' and can be described formally by relations with two arguments. Whenever the representation of a certain concept is activated the activation is transmitted by the associative relationships to related concepts whose level of activation is therefore increased. Whenever the level exceeds a threshold, the concept is activated and thus "recognized" [Rei91].

A semantic net can be visualized as a graphical representation of a data structure. The graph consists of 'concept nodes' which are connected by (mostly unidirectional) links. Unfortunately, even if the basic concept is very intuitive and simple there is no standardized definition of the underlying formalism of semantic nets so far [Bib93].

Usually each link is labeled and its label implies the type or semantic meaning of the relationship. Some types of relationships ('epistemic primitives'), such as 'is-a', 'causes', or 'is-part', are defined domain-independently. Links can be transitive, such as the 'causes' link:

$$\text{'increased level of uric acid'} \xrightarrow{\text{causes}} \text{'intracellular uric-acid crystals'} \xrightarrow{\text{causes}} \text{'goat'}$$
$$\Leftrightarrow \text{'increased level of uric acid'} \xrightarrow{\text{causes}} \text{'goat'}$$

Analogous to predicate logic, identically labeled concept nodes define the same concepts. In a semantic net, all concepts must be unique. Nodes can be used to model properties of a concept by linking another concept node that represents the property and defining the property class by the link.

$$\textit{'solid'} \xleftarrow{\text{form of appearance}} \text{'Aspirin'} \xrightarrow{\text{application method}} \textit{'oral'}$$

To clearly separate concepts and properties in the visual representation, properties are set in italics. Links that connect a concept with a property are termed 'property links', links that connect two concepts are termed 'relationship links'. This syntactic difference is required in humans as well as in machines for a correct interpretation of a semantic net.

Concept classes or events can be modeled analogous to predicate logic. By using the 'is-a' relationship it is possible to define concept classes and individual concepts that inherit properties and relationships of the concept class. Like the 'causes' link, the 'is-a' link is transitive.

Basically, semantic nets are suitable for the representation of predicate logical expressions. However, the representation of the formula by a semantic net would not represent an equivalent fact but only the formula; in such cases the semantic net is used as meta representation language for facts being represented by predicate logic.

The representation of incomplete, contradictory, or uncertain knowledge can be achieved analogous to predicate logic. Additionally it is easy to represent incomplete but restricting knowledge such as 'older than': Two persons of unknown height can have such a relational semantic relationship that defines the order of their ages without explicitly defining the particular age of the persons. Contradictory knowledge can be handled by partitioning a semantic net yielding different representation contexts that individually are not contradictory. The detection of contradictions is limited by the degree of detail and the completeness of the represented knowledge. To detect a contradiction when a person has two properties that define his age, one either needs to *know* that a human can have only one age or it has to be *defined* for such individual concepts of a concept class to be unique.

Uncertain knowledge can be represented by probabilities, or for numeric properties by defining a range of values by two property links 'upper limit' and 'lower limit'. As a

semantic net can be used as meta representation of logical expressions, it would be possible to represent multi-valued logical expressions.

Compared to predicate logic expressions one advantage of semantic nets is the *semantic nearness* of semantically related concepts, which is the result of the graphical representation format. In predicate logic, facts are linked by long series of predicates. Thus, finding related facts in a large knowledge-base may require a long search. This nearness and the use of linguistic labels simplify the understanding of semantic nets by human interpreters. The interpretation can be automated by evaluating properties and following the paths that are defined by transitive links and semantic relationships.

In summary, semantic nets offer the representation and formalization of complex models, which is (because of the graphical representation) eligible for human interpretation. The semantic nearness that concentrates related facts is a concept which is realized even more distinctively by frame representations.

1.1.3 Frames and inheritance systems

Like the semantic nets, frames are based on models of human ability to memorize that have been explored by cognition psychology. In contrast to semantic nets, frames are not directly related to the concept of associative concepts but to conceptual templates or schemes. A frame is a template for stereotype situations. Stereotypes are driven by human expectations as to which properties a typical individual of the class should have.

One example for such a situation is the concept of a bird: a typical bird consists of wings, two legs, and feathers[9]. One characteristic of templates is their incompleteness. Usually a frame includes only such properties that are of significant importance for recognizing an instance of a frame.

For example, the ability to fly seems to be an important property of birds, therefore sparrows seem to be more typical birds than penguins. This characteristic of the human way of categorizing the environment was one topic of a study performed by Rosch [Ros73]. As shown later in section 1.2.3.1, this characteristic is a key concept for the representation of vague knowledge by fuzzy set theory.

The use of schemes as a representation format was first recommended by Minsky [Min75] and Kuipers [Kui75]. Instead of distributing the knowledge that is related to one concept on many knowledge nuggets, for example by predicate logic sentences or semantic nets, all such knowledge is represented in a single structure. This knowledge representation may therefore be termed ‘object oriented’. A frame that represents a scheme consists of a label and a set of slots. Each slot is composed of facets which represent fillers containing at least two of them: the name and the value of the slot.

penguin	
height	24cm
feathers	black/white
can-fly	no

The value of a slot can be another frame; such a slot defines a semantic relationship between one frame and another, the type of the relationship, such as ‘is-a’ or ‘causes’,

[9]The biological definition of birds is much more complex, the example is intentionally kept simple for easy comprehension.

is defined by the name of the slot. As an example, the description of a landscape could include a description of the fauna that could include descriptions of birds.

The concept of specialization can be realized by one frame inheriting from another one. The frame of a hummingbird could be the specialization of a general frame 'bird' which could be a specialization of the concept 'living being'. For the representation of incomplete knowledge it is possible to define empty slots; uncertain knowledge can be defined by slots with an associated degree of uncertainty.

In contrast to predicate logic and semantic nets, frames allow a very compactly structured representation of knowledge. A frame can include all related knowledge in one structure. The concept of defining frames can be realized by using XML[10]. XML is a formal language that is used to define structures of textual documents by tags. Every substructure of a document is defined by a starting tag and an ending tag. The example of the penguin can be represented by XML, for example, as follows. Other representations are also conceivable.

```
<ENTITY name="penguin">
  <ATTRIBUTE label="height">24cm</ATTRIBUTE>
  <ATTRIBUTE label="feathers">black/white</ATTRIBUTE>
  <ATTRIBUTE label="can-fly">no</ATTRIBUTE>
</ENTITY>
```

1.1.4 Production rules

In contrast to the representation formats described earlier, production rules can be interpreted more as an application-oriented method to model human reasoning that integrates logic and algorithms. Often, humans describe their knowledge by statements like "*if* an accident has happened *then* provide first aid". For a given problem, a set of such rules is evaluated to determine a solution.

A rule consists of an antecedent (synonyms are 'premise' or 'left-hand side') and a consequent (synonyms are 'conclusion' or 'right hand side'):

$$\text{if} < \textit{antecedent} > \text{then} < \textit{consequent} >$$

The consequent is applicable if the antecedent is fulfilled. The action that is implied by the consequent can be a modification of the facts but also an interaction or message to the user of the expert system. A system that uses and evaluates a set of production rules can be compared to an interpreting program, whose data is given by the facts and whose program statements are given by the production rules. Therefore, such a system is often used as an inference or problem-solving component of an expert system[11].

Production rules that add knowledge to the fact base can be compared to logical implications. However, a production rule can behave in an entirely different way than a modus ponens implication. One significant difference between a logical implication and a production rule is that the first has a truth value while the later may, but need not necessarily have one (although, the antecedent can be a logical expression which yields a truth value).

The forward (or data driven) inference process of production rule-based systems can be structured in three steps:

1. select all rules whose antecedents are fulfilled

[10] Extensible Markup Language (XML), http://www.w3.org/XML/

[11] One widely known medical expert system that is based on production rules is MYCIN [BS84].

2. select one of these rules and evaluate the consequent
3. repeat until the problem is solved or no more rules are applicable

The first step requires an evaluation of all antecedents of the rule base. If variables are included in antecedents they have to be resolved first. The second step requires selection of one rule whose antecedent is fulfilled. If more than one rule is applicable, this conflict has to be resolved first. Different methods have been defined for resolving conflicts; sometimes more than one strategy has to be applied to reduce the set of applicable rules. Some strategies are:

- if the set of rules is ordered, choose the first (or last) rule in the set
- remove all rules from the set, whose antecedent is less specific compared to other rules; an antecedent a_1 is defined to be less specific than a_2, if a_1 yields 'true' for more parameters than a_2, for example if a_1 depends on the single condition p whether a_2 depends on two conditions $p \wedge q$ (in this example, a_1 is 'true' in one of two cases, a_2 is 'true' in one of four cases)
- select the rule that has not been applied for the longest time (or remove those rules which have recently been applied)
- select an arbitrary rule
- apply all rules in parallel

Another possible inference process is termed backward (or goal directed) inference process. The inference process is not initiated by facts to rules, but by defining a goal (the result or solution of the problem) and selecting those rules whose consequent yield it. In the next step, those rules are selected that yield the antecedent of the last ones as their consequent, and so on. If more than one rule is applicable, the same strategies for solving conflicts as for forward reasoning are applicable.

Backward reasoning has one advantage when the facts are not completely defined, as missing facts can be easily identified and determined, for example by user interactions. Forward reasoning is usable if more than one goal or no specific goal is defined. Forward and backward reasoning can be combined, for example, to identify possible goals and to obtain additional needed facts from the user.

Analogous to predicate logic, semantic nets, and frames, authors of production rule knowledge bases should avoid contradictory knowledge, such as rules that both apply at one time but imply contradictory consequents (directly or indirectly). Uncertain knowledge can be represented by certainty factors. As the antecedent is based on a logical expression, a multi valued logic such as fuzzy logic can be used. Fuzzy control production rules use fuzzy logical expressions to determine a 'degree of applicability' of the consequent (the concept of fuzzy control will be discussed in greater detail later).

1.1.5 Usability aspects of representation formats in clinical context

Natural language is the main representation format for medical knowledge, often combined with pictures. However, because of the earlier mentioned problems, it may be suitable to communicate knowledge from human to human but less suitable to communicate it from human to machine (or even from machine to machine).

If natural language is enhanced by meta-information, for example by "highlighting" parts of the text with XML by clasping parts of the text by XML-tags, it can be made semi-processable (however based on the meta-information—not on the semantic meaning of the natural language). For example, such an approach was made to add information to textual clinical guidelines, which can then be used to display those parts of the guideline to the clinical staff which are relevant in the current clinical context [HBS+01].

Structured formats, such as predicate logic, semantic nets, and frames, are useful for representing declarative information about collections of related concepts. These approaches share most of their basic properties, allowing one to define concepts, properties, concept classes, and to inherit concepts. The meanings of semantic relationships are commonly defined by linguistic labels that, for humans, imply a certain meaning. The difference between semantic relationships of concepts and properties of concepts is explicitly defined by the structures in semantic nets and frames.

The choice of one of these formats may influence the comprehensibility for humans—a physician would hardly appreciate a large knowledge base represented by predicate logic—thus knowledge should be represented at the right level of abstraction. Rule-based systems usually provide relatively simple representations of the underlying facts in the domain but may be more flexible, as they may allow the addition of new knowledge nuggets (rules) without influencing the behavior of the entire knowledge base. While logic is primarily used in a declarative way, saying what's true in the world, rule-based systems (especially forward chaining systems) are more concerned with procedural knowledge - what to do when.

For this work, a hybrid knowledge representation format has been chosen that is based on production rules and includes algorithmic procedures known as 'Arden Syntax for Medical Logic Systems' (a detailed introduction follows in section 1.4) that will be extended in ways to represent uncertain knowledge, or more exactly vague knowledge, later.

1.2 Representing uncertain knowledge

As stated earlier, the representation of uncertain knowledge may be handled by different representation formats in different ways. This section gives an introduction of different *types* of uncertainty and a short historical overview of their representation with the main focus on vague knowledge.

1.2.1 Sources of uncertainty

Four sources and subtypes of the concept that is generally known as uncertainty can be identified, located on two levels of abstraction, the numerical and the symbolic one (figure 1.2).

For the representation and definition of concepts, a representation format may use symbols (for instance labels) instead of defining concepts strictly according to their physical properties. One example is the concept 'red' that may be defined on a range of wavelengths of light, but is commonly used as a vague concept.

Starting from the numerical level, uncertainty can be caused by the (im)precision of measurements, for example as a result of technical limits of the analyzers. Every observation can then only be made by a limited degree of precision.

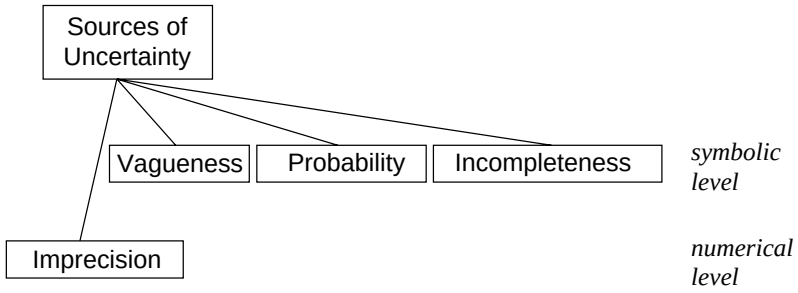

Figure 1.2: Sources of uncertainty

On the symbolic level, one type of uncertainty is the incompleteness of knowledge. Either facts in the knowledge base are unknown or facts of the real world, which are used by knowledge nuggets in the knowledge base, are unknown. Multi-valued logic supports unknown facts by introducing special truth values for 'unknown'. Therefore incompleteness can be fully supported by any representation formalism that at least includes a three-valued logic.

Another type of uncertainty on the symbolic level is based on probability. A classical method to handle such uncertain events or facts is to use the Bayesian theorem. It can be used to compute the most probable diagnosis under the presence of one or more symptoms. The probability is computed on the basis of the a-priori absolute probability of the diagnosis (that can be defined empirically) and on the conditional probabilities of the presence of the symptoms when the diagnosis is present (which can also be defined empirically). The a-priori probabilities should be based on a reliable empirical method, which includes a sufficient number of reliable patient cases, or on estimations given by medical experts.

During the knowledge acquisition for the MYCIN expert system the authors realized that the experts did not use conditional probabilities but could estimate a degree to which a diagnosis is more likely to be present if a certain symptom is present (or not present) [SB75]. These parameters were termed 'measure of belief' and 'measure of disbelief' of believing in a hypothesis in dependence to a given evidence.

However, the result of a statement about the probability of a certain fact implies that the fact is either present or not present. In contrast, the last type of incompleteness known as vagueness or linguistic fuzziness is based on the principle of conceptual schemes or templates, as previously mentioned in section 1.1.3, and can be modeled by fuzzy set theory, yielding a gradual degree of the presence of facts.

1.2.2 Historical roots of the formalization of vagueness in natural language

The vagueness of natural language is not exclusively associated with fuzzy logic. The fuzziness of concept categories was an aspect of philosophical, linguistic, and psychological theories.

1.2.2.1 Paradoxes in the early philosophy

Problems that arise when logical expressions use predicates that are based on vague concepts have been discussed in philosophy since ancient times. The *sorites paradox*[12] form a class of paradoxical arguments (also known as little-by-little arguments) that arise as result of vague definitions within the predicates used by the arguments. An example for such a paradox is the *bald man paradox*:

> It is clear that when a man with full hair looses a single hair, he will still have full hair. Yet, he will get bald if this is repeated often enough.

It appears absurd that a single hair should influence the decision as to whether a man may be termed bald or not. However, based on the assumption that a single hair cannot influence the decision and by using the modus ponens, it is possible to prove that a bald man has full hair and vice versa:

> Obviously a man who has no hair on his head is bald. If a person with n hairs is bald, a person with $n + 1$ hairs is also bald. These rules imply that a man with 1 hair is bald. Thus, a man with 2 hairs is bald as well, and so on.

One could specify a number of hairs that defines a threshold between 'bald' and 'non bald'. However, intuitively such a threshold would be unacceptable, in which a single hair could decide whether a man is bald or not. A predicate like 'is bald' appears to be *tolerant* to sufficiently small changes in the set of facts—in this case the number of hairs (compare [Wri75]). The change of facts that is caused by the step "If a person with n hairs is bald, a person with $n+1$ hairs is also bald" seems to be too small to make any difference to the application of the predicate, whereas significant changes—the accumulation of many small changes—seems to exceed the tolerance.

On the other hand, it must be possible to basically distinguish between the two concepts 'bald' and 'not bald'. One solution to this paradox might be the conclusion that two-valued logic cannot be applied on expressions that include *vaguely* defined concepts. As soritical paradoxes are the result of the inherent vagueness of a language, an ideal language with precise terms could eliminate them[13].

1.2.2.2 The perfect language

In an attempt to realize Leibnitz' ideas of a perfect and ideal language, Frege defined a formal notation for regimenting reasoning by a logic-oriented language that operates on unambiguous symbols (terms). The idea of a perfect language has been propagated in the early 20th century mainly by Russell and Wittgenstein. Russell pointed out that vagueness is not the result of things being, but only existent on the symbolic level. Thus, talking about vagueness and precision is only relevant when talking about representation formats.

> "Vagueness and precision alike are characteristics which can only belong to a representation, of which language is an example. [...] Apart from representation, whether cognitive or mechanical, there can be no such thing as vagueness or precision; things are what they are, and there is an end of it." [Rus23]

[12] The name 'sorites' is derived from the ancient Greek word for 'heap', since the first version of this paradox involved a heap of wheat.

[13] Compare http://plato.stanford.edu/entries/sorites-paradox

Russell tried to prove that all terms are vague, starting with the definition of the color 'red'. Certain shades of the color 'red' may cause problems when humans have to decide whether they still can be termed red. The cause of this uncertainty is not ignorance on the part of the interpreter regarding the meaning of the term 'red', but in the basic vagueness of the concept[14]. On account of this vagueness the principle of the excluded middle cannot be valid for such problems. The law of the excluded middle is true when precise symbols are employed, but not true when symbols are vague [Rus23].

Russell defines 'vagueness' as opposed to 'accuracy'; both terms describe the relationship of a representation to the object being represented. An accurate representation has a one-to-one relation to the objects; a term of a language must have only one meaning and two different words cannot have the same meaning. In contrast, a vague representation has a one-to-many relation; vagueness is therefore a matter of degree. As mentioned in the introduction of predicate logic, to express a concept by natural or formalized language, usually more than one right representation is valid; they have a one-to-many relation to the objects they represent.

However, Russell stated that it would be a mistake to assume that vague knowledge is invalid knowledge. In contrast, he claimed that a vague speculation might be much more likely to be true than a precise one.

1.2.3 Vague categories and their elements

An empirical approach to the definition of vagueness is to use 'consistency profiles'. The vagueness of a symbol is measured by the relation of yes/no answers of persons who had to apply the symbol s on an object o taken from a series of objects O. Each symbol s provides a consistency function c, that defines for each object x the quotient of the occurrence of positive and negative answers $c(o,s) = \frac{h_p(o,s)}{h_n(o,s)}$[15]. The graphical representation of this function is termed the consistency profile for a given symbol with regard to a series of objects and was intended to be used in descriptive form.

1.2.3.1 Cognition psychology

Studies on the categorization of objects by terms were already mentioned in the context of frame representations. Rosch and her staff examined the cognitive efficiency of structures by categories (horizontal) and order and grades of membership within the single categories (vertical). One result was that most categories could not be exactly defined with crisp borders [Ros73, Ros78].

For one experiment the subjects had to specify, on a numeric scale from 1 to 7, how far given objects meet their "idea or picture" of the given category. This value defined the 'degree of prototypicality' of the object in respect to the category. The categories are separated into three classes: basic level categories, superordinate categories and subordinate categories. Objects that belong to the same basic level categories share many properties. One example is the category 'table'. Elements of superordinate categories, for example

[14] Another definition of vagueness by Peirce is based on a non-well-defined usage of language by a speaker: "By intrinsically uncertain we mean not uncertain in consequence of any ignorance of the interpreter, but because the speaker's habits of language were indeterminate..." (C.S. Peirce. Vagueness. *Dictionary of philosophy and psychology*, 2, 1902, cited from [Bie97])

[15] The function is not defined for $h_n(o,s) = 0$. However, as one presumption was that every language (or symbol) is vague, the denominator should never be zero, even if in practice (and as shown later in fuzzy sets) some objects may absolutely define a category and therefore be the reference for an object series.

'furniture', share comparatively few properties, while objects of subordinate categories share properties among different categories, for example 'dining table' and 'kitchen table'.

Other experiments touched the relationship of prototypicality to family resemblance. Family resemblances define a concept not by essential common characteristics that apply for every member of the concept, but by overlapping similarities[16]. Elements of a category that are most prototypical have the highest family resemblance to all elements of the same category and the lowest family resemblance to elements of other categories [RM75]. Thus, a prototype may be termed the "best representative" of a category.

Categories without sharp boundaries provide, on a comparable degree of linguistic abstraction, a higher content of information compared to categories without fuzzily defined boundaries, as elements in such a fuzzily defined category can be classified vertically. The use of such categories is less problematic as they do not require the definition of sharp boundaries whose exact limits are not reasonable because of lack of knowledge or facts. As the membership is not restricted to a 'yes' or 'no' decision, the resulting models are often more robust [Bie97].

Figure 1.3 illustrates the issue of robustness by a model of a therapy recommendation. The recommendation is based on a set of symptoms which are each based on measured or observed data. Such data could be, for example, the measured blood oxygen level. If it is within a pre-defined range, it represents the symptom "decreased blood oxygen level". On a higher level of abstraction, this symptom could be called "hypoxemia" and could necessitate an increase of FiO_2.

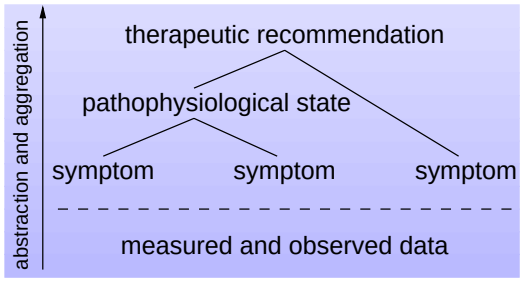

Figure 1.3: Process of abstraction

If the symptom "decreased blood oxygen level" was defined by a crisp set, and the measured blood oxygen level was closely outside the borderlines that define the set, the entire model no longer applies. Whereas, if data to symbol conversion were defined with fuzzy borders, the observation might still represent the symptom "decreased blood oxygen level" to a reduced degree. Therefore the symptom could still be called "hypoxemia" and the entire model would still apply, even if the degree of, say, "application", might be slightly decreased compared to the first example. The overall fuzzy definition can therefore be more robust in terms of small changes of the underlying facts than the crisp one.

If the last example was not only based on one observation but on many facts that all include vagueness, and the vagueness is increased by aggregating less abstract concepts to more abstract concepts, the robustness of the entire model would be progressively reduced when using sharp definitions for vague concepts. Zadeh defined the 'principle of incompatibility'

[16] Wittgenstein, who molded this term, unfortunately did not exactly define the meaning of 'similarities' in this context.

in which he describes the relationship between the complexity of a system, precision, and significance:

> "... as the complexity increases, our ability to make precise and yet significant statements about its behavior diminishes until a threshold is reached beyond which precision and significance (or relevance) become almost mutually exclusive characteristics." [Zad73]

1.3 Fuzziness theories

Cantors definition of sets presumes that the question as to whether an object belongs to a set or not can be answered non-ambiguously by 'yes' or 'no'. For the definition of mathematical concepts this crisp definition of sets that have sharp and precise boundaries works well. In practice, precise thresholds and criteria are often required and can at least be defined as long as measurable parameters exist. On the other hand, a definition of linguistically defined concepts by exact thresholds is often arbitrary and subsequently leads to problems as mentioned in the last section. The benefit of getting a computable definition by precise thresholds frequently involves a loss of information in borderline cases.

An alternative formalization of such linguistic classes may be the use of a *characteristic function* that defines a gradual degree of membership of objects to a class instead of a bivalent one. This concept of *fuzzy sets* is an extension of classical set theory and was introduced by Lotfi Zadeh in 1965 [Zad65] (all formal definitions are made in section 1.3.1). He called this gradual type of characteristic function *membership function*; in this work *compatibility function* is used[17]. Zadeh's concept of fuzzy sets were his starting point to evaluate concepts for representing linguistic concepts and using them for reasoning. So far evolution has reached the concept of 'computing with words'.

Approaches to represent fuzzy semantics quantitatively by fuzzy subsets (for instance characterizing concepts such as "middle aged"), led to the definition of the concept of a linguistic variable, whose values are labels of fuzzy sets [Zad71, Zad73]. The values of a linguistic variable can be modified by 'linguistic hedges' such as 'very', 'slightly', or 'much' and combined by using the connectives 'and', 'or', and 'not'. For example, if a linguistic variable "height" represents the height of a person, possible values could be 'small', 'average', or 'tall' which could be modified to 'very small', 'small or average', or 'slightly tall'.

Zadeh states that the linguistic approach, even if it abandons numbers, does not break with the mathematical way of dealing with problems. Instead it links quantitative classifications with qualitative ones by using words whenever precise numerical characterizations are not appropriate to increase the precision of the meaning of words [Zad76b].

The linguistic approach also affects the representation of truth values. In contrast to bivalent systems where truth can be represented only by two values, 'true' and 'false', the linguistic approach can be used to adopt multi-valued logics by labels such as 'true', 'very true', 'not quite true', etc. [Zad89]. Today, in common language the term 'fuzzy logic' which originally stands for a multi-valued logic based on fuzzy subsets representing truth values, stands for all concepts related to fuzziness theories. In addition to the representation of truth values by fuzzy sets, the degree of compatibility of an element

[17] The term is chosen as the function will mainly define by which degree a measured value is compatible with the definition of a medical concept.

of the universe of discourse to the fuzzy set can be interpreted as a truth value of the predicate [Zad90]. For example, the degree of compatibility of Bob to the class "persons who have glaucoma" can be interpreted and equated to the truth value of the predicate *has_glaucoma*(Bob) (except that in classic predicate logic this formula would yield either 'true' or 'false' while the degree of compatibility can additionally represent any value in between).

Simple relations between linguistic variables can be represented by fuzzy conditional statements, more complex relations by fuzzy algorithms including linguistic statements or fuzzy if-then rules. Fuzzy logic and fuzzy if-then rules are used by 'fuzzy control' that became a popular concept of fuzziness theories in industrial applications. Fuzzy control uses linguistic variables as input and output values of fuzzy production rules. The first important application of fuzzy control rules was achieved in 1973 by Mamdani and Assilian, who defined a system to control a small steam engine. The remarkable result was not only *that* the control system, which has been set up within a weekend, worked, but *how fast and easy* a rather complex system could be described by a set of rather simple if-then rules. Fuzzy control was the break through of "fuzzy logic" in industry, especially in Japan where in the late 80s many consumer products started to use these technologies.

Finally, as mentioned before, the current methodology that is propagated by Zadeh is termed *computing with words* and is a consequent derivative of the linguistic approach [Zad96, Zad99]. However, this work only employs certain some rather basic concepts of all these theories and methods, such as fuzzy sets (especially the *degree of compatibility*), fuzzy logic, and linguistic variables which will be defined in the next section.

1.3.1 Fuzzy set theory

A crisp set is defined as a collection of objects known as elements. In the following crisp sets are termed *sets* and represented by capitals, such as A, whereas lower-case letters such as a represent single elements. The statement "a is element of A" is notated symbolically by $a \in A$.

A set can be defined in many ways. In the first method the elements of the set are explicitly listed:

$$A = \{2, 3, 4, 5\} \tag{1.5}$$

Another method is to define a criterion that the elements of the set have to fulfill:

$$A = \{x | x \geq 2 \text{ and } x \leq 5\}, x \in X \tag{1.6}$$

If the elements $x_0, \ldots, x_n$ in equation 1.6 are elements of a universe of discourse X, another way to define the set is to use a *characteristic function*:

$$\mu_A(x) = \begin{cases} 1 & x = 2, 3, 4, 5 \\ 0 & \text{otherwise} \end{cases} \tag{1.7}$$

For every $x \in X$, the characteristic function defines the membership $x_i \in A$ by returning 1 for 'membership' and 0 for 'no membership'. Additionally, the characteristic function can be represented graphically as shown in example 1.

Example 1: (crisp set)

The set A of all real numbers $x \in \mathbb{R}$ which are greater than 18 is defined by the following characteristic function (figure 1.4 shows a plot of the characteristic function).

$$\mu_A(x) = \begin{cases} 1 & x \geq 18 \\ 0 & \text{otherwise} \end{cases}$$

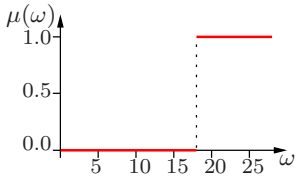

Figure 1.4: Characteristic function of the set of real numbers which are greater than 18

Example 1 could be used to define the set of "adults" as a set of individuals who are at least 18 years old. Whereas such a crisp definition might be required for legal reasons, it is not suitable for use in other cases, such as for the administration of drugs, or to term a patient on the verge of turning eighteen a child. To avoid such significant changes of categorization, the set could be defined as a 'fuzzy set'.

Definition 3 (fuzzy set, degree of compatibility, label):

A **fuzzy set**

$$A = \{(x, \mu_A(x)), x \in X\}$$

is characterized by a compatibility function

$$\mu_A : X \rightarrow [0, 1]$$

that defines for each $x \in X$ the **degree of compatibility** *or, in other words, the degree of membership of x in fuzzy set A as a real number in the interval $[0, 1]$. A is the* **label** *of the fuzzy set. The degree of compatibility of 1 is defined as 'full compatibility' or 'full membership', whereas the degree of 0 is defined as 'no compatibility' or 'no membership'.*

Neither the universe of discourse nor the compatibility function is limited to formal definition. The function may be defined by an equation, a graph, or by the explicit definition of the degree of compatibility of each element. One way to formally define a fuzzy set on $X = \mathbb{R}$ with parameterizable functions is to use s-, z-, or π-type functions as originally defined by Zadeh [Zad76a].

$$s(x, \alpha, m) = \begin{cases} 0 & ; x \leq \alpha \\ 2(\frac{x-\alpha}{m-\alpha})^2 & ; \alpha < x \leq \frac{\alpha+m}{2} \\ 1 - 2(\frac{x-m}{m-\alpha})^2 & ; x < \frac{\alpha+m}{2} \leq m \\ 1 & ; \text{otherwise} \end{cases} \tag{1.8}$$

$$z(x, \beta, m) = 1 - s(x, \beta, m) \tag{1.9}$$

$$s/z(x,\alpha,m_1,m_2,\beta) = \begin{cases} s(x,\alpha,m_1) & ; x \leq m_1 \\ z(x,\beta,m_2) & ; \text{otherwise} \end{cases} \tag{1.10}$$

The π-type function can be viewed as a special case of a combined s/z function with $m_1 = m_2$. The terms s- and z-type describe the form of the function graph (compare figure 1.5).

The first parameter α of an s-function defines the lower threshold up to which the function returns 0 and therefore defines the range of values which are absolutely incompatible with the term defined by the fuzzy set. The second parameter m defines the upper threshold from which the function returns, 1 defining the range of values which are fully compatible with the term. Zadeh defined the ordinate of the inflection point as an additional parameter, which is simply replaced here by the expression $\frac{\alpha+m}{2}$.

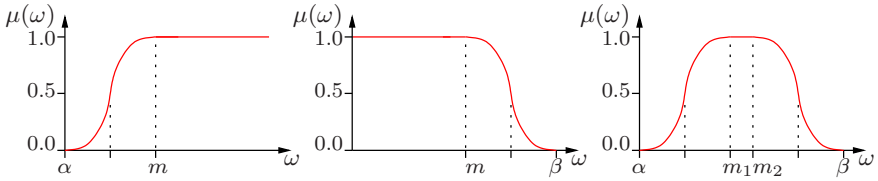

Figure 1.5: Parameterizable s-/z-type compatibility function

These types of functions provide very smooth transitions around the thresholds and seem intuitively to be very suitable for defining the fuzzy set. However, practical experience showed that such accuracy is not relevant; simpler linear functions as shown in figure 1.6 are sufficient.

The linear equivalent to the s-type function is defined as:

$$\mu_{\text{right}}(x,x_0,\Delta x_0) = \begin{cases} 0 & x \leq x_0 - \Delta x_0 \\ \frac{x-(x_0-\Delta x_0)}{\Delta x_0} & (x_0 - \Delta x_0) < x < x_0 \\ 1 & x_0 \leq x \end{cases} \tag{1.11}$$

Analogously, the linear z-type function is defined as:

$$\mu_{\text{left}}(x,x_0,\Delta x_0) = \begin{cases} 1 & x \leq x_0 \\ 1 - \frac{x-x_0}{\Delta x_0} & x_0 < x < x_0 + \Delta x_0 \\ 0 & x \geq x_0 + \Delta x_0 \end{cases} \tag{1.12}$$

Finally the combined function is defined as:

$$\mu_{\text{trapez}}(x,x_0,x_1,\Delta x_0,\Delta x_1) = \begin{cases} 0 & (x \leq x_0 - \Delta x_0) \vee (x \geq x_1 + \Delta x_1) \\ \frac{x-(x_0-\Delta x_0)}{\Delta x_0} & (x_0 - \Delta x_1) < x < x_0 \\ 1 & x_0 \leq x \leq x_1 \\ 1 - \frac{x-x_1}{\Delta x_1} & x_1 < x < (x_1 + \Delta x_1) \end{cases} \tag{1.13}$$

The absolute parameters α and β have been replaced by the relative interval Δx_0 and Δx_1. The following examples show the general use of the linear compatibility functions.

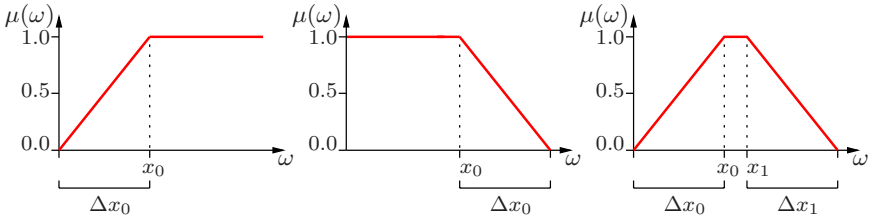

Figure 1.6: Parameterizable linear compatibility function

Example 2: (fuzzy sets: age definition)

For classifying children and adults, two fuzzy sets "child" and "adult" are defined on the universe of discourse $X = \mathbb{R}^+$. The fuzzy set child is defined by a linear s-function with $x_0 = 17$ and $\Delta x_0 = 2$ and the fuzzy set adult by a linear z-function with $x_0 = 19$ and $\Delta x_0 = 2$ (figure 1.7).

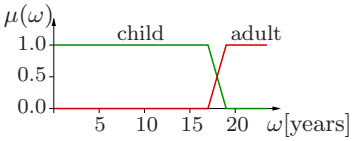

Figure 1.7: Fuzzy sets for "child" and "adult"

The definition of the compatibility function mainly depends on the experts who define the fuzzy set and on the context in which the fuzzy set has to be used. The definition used by the last examples might be suitable for computations where the dose of a drug depends on whether the patient is a child or an adult. It might be less suitable for deciding whether a person is 'adult' enough to drive a car or to watch a movie. Therefore fuzzy sets often are subjective and context-sensitive.

The use of parameterizable linear functions as compatibility functions has two advantages. First, the effort to define the functions is reduced to the definition of single parameters that define the interval from 'full compatibility' to 'no compatibility'. Further, the mathematics for operations between such fuzzy sets can be implemented efficiently.

However, not every concept can be modeled by using these three functions, for example if more than one range of values has to be defined as fully compatible. For this task piecewise linear functions could be used. However, for extending Arden Syntax, functions 1.11 to 1.13 are quite sufficient.

Basic operations on fuzzy sets

As in crisp set theory, a single fuzzy set can be inverted by a complement operator and two or more fuzzy sets can be combined by intersection or union operations. In contrast to the corresponding operations on crisp sets these operations on fuzzy sets are not unique. Each operation is represented by a class of operators, the choice of a particular operator has to be determined by the purpose for which it is used.

The most common operators for fuzzy union, intersection, and complement are defined as

$$(A \cup B)(x) = \max[A(x), B(x)] \tag{1.14}$$

$$(A \cap B)(x) = \min[A(x), B(x)] \tag{1.15}$$

$$\overline{A}(x) = 1 - A(x) \tag{1.16}$$

Figure 1.8 illustrates the use of these definitions. The left side shows the fuzzy sets that are used to compute the union, the intersection, and the complement or complement, the right side shows the result (indicated by the bold line).

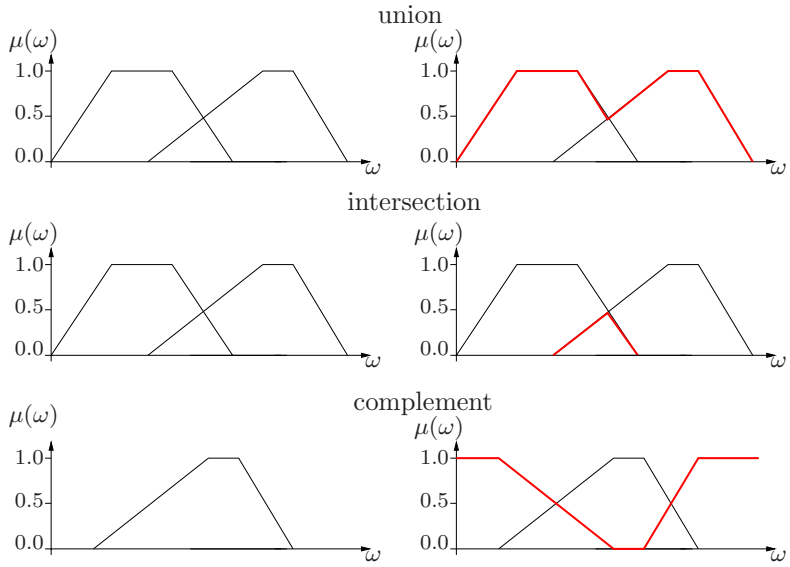

Figure 1.8: Fuzzy union, intersection, and complement by using equations 1.14 to 1.16

1.3.2 Linguistic variables

The main function of mathematical variables is to represent unknown values which usually are elements of a predefined and restricted set by a symbolic label. In addition to such numerical variables, Zadeh defined *linguistic variables* whose values are linguistic terms that are semantically related labels of fuzzy sets [Zad73, Zad87].

Definition 4 (Linguistic variable):

Mathematically, a **linguistic variable** *can be defined by a quintuple*

$$V_L = \{X, T, \Omega, G, B\} \tag{1.17}$$

where X is the name of the variable, T a set of terms representing the values of X, Ω the universe of discourse, G a set of syntactical rules which generate T, and finally B a set of semantic rules that define the linguistic discretization of Ω.

Linguistic variables are can be used to adequately represent expressions in natural language rules, such as "if *intraocular pressure* is *increased* then…". The concept and use of a such a linguistic variable is illustrated by the next example.

Example 3: (Linguistic variable: intraocular pressure)

The normal eye maintains an internal pressure from 12 to 22 mm of mercury. A linguistic variable 'intraocular pressure' can be defined as $X = \text{IOP}$, $T = \{\text{normal}, \text{increased}\}$, and $\Omega = [0, 70]$[18]. The linguistic discretization B is

[18] In this example, G is not defined separately.

defined by two fuzzy sets, which define the relationship of $t \in T$ to Ω (figure 1.9). A fuzzy set is characterized by a compatibility function which defines the degree of compatibility of $\omega \in \Omega$ to a term: a degree of 0.0 indicates no compatibility whereas a degree of 1.0 indicates full compatibility.

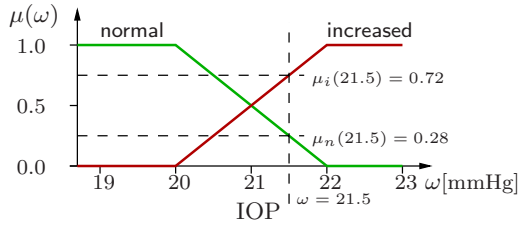

Figure 1.9: Linguistic variable 'IOP' (intraocular pressure)

Based on an numerical input value of $\omega = 21.5$ the linguistic variable 'IOP' has the value 'normal' by a degree of 0.28 and the value 'increased' by a degree of 0.72.

Usually a classic variable represents at each moment only one value (that however might be indeterminate). As shown in the last example where the variable represents 'normal' and 'increased' simultaneously, a linguistic variable can represent one or more values that even might be contradictory, each by a certain degree.

If the variable represents different characteristics of a diagnosis, such as 'normal', 'pathological', 'suspicious', or 'glaucomatous', its single values can be easily communicated in textual form. For example, such a message could be: "The patients eye status is rated 'glaucomatous' by a degree of 0.8" or "The patients eye status is highly rated 'glaucomatous'." Such variables usually represent the results of fuzzy control systems.

1.3.3 Fuzzy control

Fuzzy control systems are based on a set of fuzzy production rules that use linguistic variables as input and output values (figure 1.10). The input values are used as antecedents by the single production rules and define the degree of validity of the rules' conclusion. The output is an aggregation of the single results of all applicable rules.

Before a numeric value can be used as input for a production rule it has to be represented by a linguistic variable. Usually, for every term $t \in T$ the degree of compatibility of the given numerical value is determined by applying the value to the corresponding fuzzy set. This step is termed the *fuzzification* of the input value.

As the next step the input variable is used by the set of production rules as condition on the left side. Every rule consists of an antecedent (the condition) and a consequent (the conclusion) and is formed as follows:

```
if <condition> then <conclusion>.
```

The conditional expression on the left side can be composed of one or more comparisons of a linguistic variable and a linguistic value.

```
<input_variable> has <value>
```

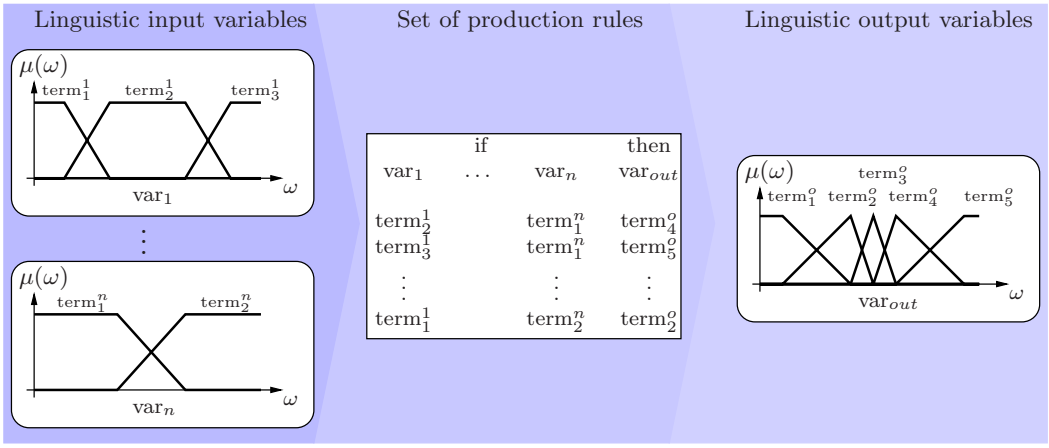

Figure 1.10: Structure of fuzzy control rule sets

where `value` is one of the linguistic values T. If the condition consists of more than one comparison they can be combined using logical operators like 'and', 'or', and 'not'.

The conclusion usually consists of one or more assignments of a value to a linguistic variable such as

```
set <variable> to <value>.
```

The degree of truth of the condition determines the degree of validity of the conclusion. If the conditional expression is fulfilled only partially then the output variable is also set to the value partially. More than one value—even conflicting ones—can be assigned to the same output variable. If the same value is assigned to the variable more than once, the maximum degree of validity is chosen. For example, figure 1.11 shows two linguistic variables: the left one is used as input variable, the right one as output variable. This variable represents four linguistic values 'none', 'low', 'medium', and 'high', based on a numerical universe of discourse. The variable has been set to 'low' by a degree of 0.4 and to 'medium' by a degree of 0.6 and can be interpreted as "more medium than low".

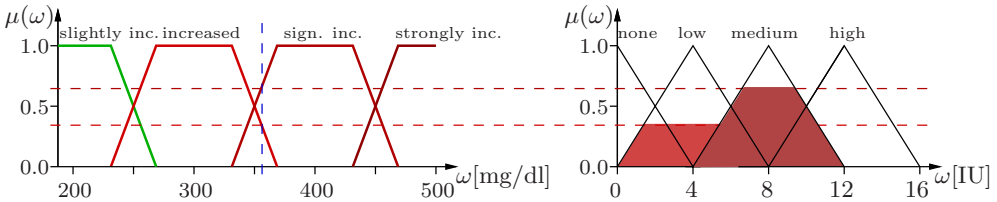

Figure 1.11: Reaction of production rules for a given numerical input value

As mentioned earlier, linguistic variables that represent concepts that can be directly communicated to a person may be included, for example, in a textual message. Whereas, when such a variable represents a numerical parameter as shown in figure 1.11 and a concrete value $\omega \in \Omega$ is needed, for instance for dosing a medication, the linguistic variable must additionally be *defuzzified*.

Definition 5 (defuzzification):

> *The value of linguistic variables that are defined on a numerical universe of discourse can be converted from its linguistic representation to a numerical one*

by the **defuzzification** *of the linguistic variable. As a variety of methods to defuzzify linguistic variables are available, some common methods are defined exemplarily.*

The **Mean-of-Maximum (MoM)** *method selects the value y where where $\mu(y)$ yields the maximal compatibility. If the local maximum is a plateau the mean is usually computed by equation 1.18 [Bie97].*

$$y_0 := \frac{y_{\min} + y_{\max}}{2};\ y_{\min} = \min M,\ y_{\max} = \max M \tag{1.18}$$

The **Center-of-Maxima (CoM)** *method computes a weighted average that is computed of the average of all local maxima.*

$$y_0 = \frac{\sum_i x_i \mu(x_i)}{\sum_i \mu(x_i)};\ x : \text{average of plateaus} \tag{1.19}$$

The **Center-of-Gravity (CoG)** *methods computes a the center of gravity of the area that is defined by the union of all fuzzy sets.*

$$y = \frac{\int_{y \in Y} y\mu(x_1, \ldots x_n)(y)dy}{\int_{y \in Y} \mu(x_1, \ldots x_n)(y)dy} \tag{1.20}$$

Applying the MoM method on a compatibility function with separated maxima, the result might be a value whose degree of compatibility is less than the computed maximum. Further, this method may yield a rather discontinuous result as the result is only driven by the fuzzy set of the linguistic value with the highest degree of assignment. Therefore, at the borderlines where this main influencing value changes, the result could change significantly. Thus this method is not recommended for practical use [NKK96]. Fur the fuzzy extensions to Arden Syntax, the CoM method will be mainly used.

1.3.4 Fuzzy algorithms

Fuzzy algorithms are used in everyday life, such as for cooking recipes, instructions for finding a path, or instructions for treating a disease. In contrast to conventional algorithms, fuzzy algorithms contain fuzzy statements, or, as stated by Zadeh, "the class of nontrivial problems for which one can find precise algorithmic solutions is quite limited" [Zad68]. He illustrates the concept of fuzzy algorithms by fuzzy statements such as

- "Set y approximately equal to 10 if x is approximately equal to 5"
- "If x is large, increase y by several units"
- "Move several steps forward".

Assigning a precise meaning to a fuzzy instruction can be done by using compatibility functions. For example, the expression "several steps" can be defined by a discrete fuzzy set[19] A:

$$A = \{0/0, 1/0, 2/0, 3/0, 4/0.8, 5/1, 6/1, 7/1, 8/0.7, 9, 0, 10/0, \ldots\}$$

[19]The notation used by this example defines the set by pairs of the elements and their corresponding degree of compatibility.

Five, six, or seven steps are defined to be absolutely compatible with the expression "several steps". Less than four steps and more than eight steps may definitely not be termed "several steps"; four or eight steps are borderline cases.

However, the ambiguity of the execution of the fuzzy instruction is not resolved by the precise definition of a fuzzy instruction. How many steps would a human take, if this fuzzy set would represent his interpretation of "several steps"[20]?

Two methods to determine a concrete instruction are defined in [Zad68]. The 'probabilistic execution' defines for each possible outcome of a fuzzy instruction a probability that is proportional to the degree of compatibility. In the example, 4 steps would be chosen with probability $\frac{0.8}{4.5}$, whereas 8 steps would be chosen with probability $\frac{0.7}{4.5}$, and 5, 6, and 7 steps with probability $\frac{1}{4.5}$. The other outcomes could not be chosen. Optionally, the method could be extended by defining a threshold that removes all outcomes whose degree of compatibility is less than the threshold.

The 'nondeterministic execution with threshold' selects any of the outcomes whose actual degree of compatibility is greater than a defined threshold. As a variant of this method, the element with the highest degree if chosen.

Obviously, in cases of a fuzzy choice of possibilities to act, finally one decision has to be made. On the other hand a decision between two possible decisions that differ in their degree of compatibility only marginally implies similar problems as the decision why an element should be included into a crisp set and another element that is closely outside the definitions should not; a small difference of the input factors could result in a significant change of the output.

An alternative method to represent typical "*if ... then ... else*" instructions is defined by the *compositional rule of inference* which can be used with fuzzy sets that represent numerical values [Zad73]. This rule considers both alternatives in cases where usually a binary decision, such as described earlier, is required. Zadeh illustrates the concept with an example

if *small* then *large* else *not very large*

with *very small* as input value. The result of the instruction is composed both by the concept *large* and *not very large*, each of it gaining as much influence on the result as defined by condition and input value.

The compositional rule of inference is defined by

$$y = x \circ R \tag{1.21}$$

where R is a fuzzy relation that represents the fuzzy instruction, x is a fuzzy subset that represents the conditional instance, and y the fuzzy subset which is induced by the composition $\circ$. The result is therefore a fuzzy set and not one concrete value. The use of this method is not explained in detail, as only the concept of taking two alternatives into consideration that both have a certain influence on the result is of importance for the extensions of Arden Syntax. More detailed information about the compositional rule of inference can be found in [Zad73].

Common constructs of algorithms that are widely used in classical programming languages are yes/no criteria such as "If x is small then stop else goto 7". Zadeh assumes that an

[20] In his paper Zadeh excluded any external factors that could influence the decision, such as the expenditure of the energy involved, etc.

individual would chose the alternative with the higher degree of truth and defines this rule as the 'rule of the preponderant alternative' [Zad73]. However, this approach is similar to the 'execution with threshold' that implies a crisp yes/no decision. In contrast, the compositional rule of inference implies that the execution has to be carried out in parallel, as both alternatives gain influence according to the degree of truth of the definition. This problem is one to be resolved when extending the Arden Syntax and defining fuzzy conditional statements.

Other formal definitions of fuzzy algorithms are based on fuzzy Turing machines [Zad68, San70].

1.4 The Arden Syntax for Medical Logic Systems

The Arden Syntax is a hybrid knowledge representation format that consists of a frame-like document structure and a procedural programming language. It was born in 1989 during a workshop at the Columbia University's Arden Homestead Conference Center. The workshop was driven by the idea of creating a knowledge representation format that would facilitate the definition of knowledge bases and the utilization of decision support systems. At the time the group identified the rather difficult creation of knowledge bases as one possible reason for the poor acceptance of decision-support systems [Hri94a]. Most decision-support systems remained small to medium in size and were bound to the institution as the launch of new systems usually required to define the knowledge base from the scratch [Gao93]. It is assumed that no individual institution would develop knowledge bases that cover all aspects of medicine and it would therefore be desirable to be able to share parts of knowledge bases among institutions [GSA+92]. Advances in medical research imply changes in the medical knowledge base that has to be maintained over time. Therefore, the main aims for the Arden Syntax were reusability, shareability, and understandability.

It was proposed that these aims would be achieved by standardizing the syntax, by taking care of data base queries, and by using a representation format that should be easy to understand even for non-programmers. The syntax has been largely derived from the formats used by HELP (LDS Hospital) and CARE systems (Regenstrief Medical Record System) [Gao93, Pry94]. The concept of Arden Syntax allows one to define 'rule-like' knowledge bases where an action is the (conditional) result of an event. A knowledge base represented by Arden Syntax is modular and consists of individual rules known as *Medical Logic Modules* (MLMs).

Usually, an Arden Syntax-based system operates as follows: The system ("rules engine") listens to the occurring events and chooses those MLMs out of its knowledge base, which define the occurred event as an evocation condition. These MLMs are evoked as defined by the condition, either instantly or (if defined) delayed, and execute the included data base queries. Then the decision logic that is represented by a rather simple programming language concludes either 'true' or 'false'. In the case of a false conclusion the process stops here. Otherwise the rules engine executes an action that is separately defined by the MLM.

Most elements of the decision logic use keywords that are close to natural language and can be used to create easily readable expressions such as 'if age is less than 18 then' or '5 minutes after the time of event 1'. A three-valued logic provides the representation of incompleteness as a basic form of uncertainty. However, to represent uncertainty in terms of vaguely defined concepts, the syntax needs to be extended by concepts of fuzzy

set theory and fuzzy logic. This extension of the Arden Syntax is the main topic of this thesis and is described in the next chapter. Before going into greater detail, the rationale of Arden Syntax and some information about its most recent developments are given in the last part of this section.

A tutorial for writing MLMs of Arden Syntax version 1 has been published in 1994 and can be used to gain an impression of the syntax [Hri94b]. The specification of actual versions 2 and 2.1, which will be published at the end of 2003, can be obtained from HL7; an updated implementation guide is on its way [Hls02a]. As further introductions to the Arden Syntax [HLP+94, HPW95] are recommended.

1.4.1 General aspects

An important aspect of the Arden Syntax is the proposed target audience and the users of MLMs. As mentioned in the introduction, many knowledge representation formats can be used to solve a given problem. The choice of a specific format is mainly driven by the question *what* has to be solved and who has to define the knowledge.

Classically an expert who has the domain specific knowledge is supported by a knowledge engineer who is skilled in the knowledge representation format and who helps the expert to formalize its knowledge. However, it is important to motivate the *users* of the system to contribute to the knowledge base and to keep it up to date, to create a "sense of ownership" for the system [GM99]. Tools for maintenance have to meet *their* needs and the representation format is one of them. Therefore an easily readable, understandable, and writable syntax could help the experts to work on it independently and to improve its use as well as enhance the interest of the clinical staff in clinical decision support.

For this reason especially the algorithmic part of the Arden Syntax is kept simple, to help those who are not familiar with programming languages or logic representation formats to understand the modules. Its simplicity is one of its major differences compared to other representation formats that are more complex and might even be more powerful in terms of creating complex algorithms. Using the Arden Syntax in different projects seemed to validate this requirement [JMB94].

1.4.1.1 Modularity

Arden Syntax was intended to be a format for knowledge bases that can be represented as a set of discrete modules. An MLM is structured like a frame by three categories whose entries are slots that are composed of a name and a value. Each MLM should include sufficient knowledge for making one specific decision, such as generating an alarm message or suggesting diagnostic hypotheses. The independence of the MLMs implies that is should be generally possible to understand and use every knowledge module without having references to other MLMs [Gao93]. Further, MLMs can be easily added or removed without affecting the overall performance of the system or other MLMs[21].

An MLM contains in its 'maintenance' and 'library' category knowledge about maintainability, such as author and status. It explains the represented knowledge by free text or provides links to further sources of knowledge, such as literature. This information should help to increase the user' acceptance and the credibility of the rules [JMB94]. They may be

[21] Except, of course, if an MLM makes use of the possibility to reference and call other MLMs directly. However, it is generally possible to remove and add parts of the knowledge base without affecting other parts of it.

used to explain the decisions made by the module. The last category 'knowledge' includes the decision logic and data base queries.

1.4.1.2 Programming language

The programming elements used for implementing the decision logic provide statements for program flow control (such as assignments, if-then statements, and loops), operators for building expressions that combine and manipulate data (such as logical, comparative, or algorithmic operators), and variables of different data types.

One important issue is the handling of variables. In contrast to many programming languages, Arden Syntax variables are not bound to data types, but can dynamically represent data of different types during the execution of an MLM. One advantage of this dynamic data typing is, for example, that one variable can be used to process data which might be stored in the data base by different data types [HCJC92]. For example, a query for serum potassium may return either a numerical value or a code like "hemolyzed". However, it is up to the author of an MLM to ensure that the logic will handle every type of data that can be represented by the variable during runtime correctly, as many operators are defined only to work on a limited subset of the Arden Syntax data types. In the last example textual codes may have to be handled in a different way than numerical ones. Owing to the missing type declaration of variables, the use of wrong data types as arguments of operators cannot be detected during the compilation of the MLMs[22].

Even if the syntax is kept simple, Gao et al. attempted to enhance the readability by creating editors to support the authoring of MLMs. The presented editors were in the range from computer-based forms with pick-lists and default values for some input fields to Microsoft Word-based solutions. The conclusion was that a good knowledge editor can help physicians to turn their knowledge into MLMs, as not only the syntactical representation of knowledge is a challenge for the deployment of knowledge bases, but also the ability to turn one's knowledge and practice into algorithms [GSA+92]. Another form-based editor has been created using Protégé-II[23] [BE97].

1.4.1.3 Separation from data mapping and logic

Data base queries have been separated from the decision logic by two separate slots in the 'knowledge' category to increase the shareability of MLMs. In theory, whenever knowledge has to be transferred from one institution to another, only the data base mappings would have be modified instead of rewriting the entire rule. However, studies showed that the shareability of MLMs is still problematic due to different data base formats.

One limitation is the availability of the data that are used by the rules and must therefore be present in both data bases. However, once the data is available, further problems may arise due to the absence of a standardized medical vocabulary or encoding system. This lack of standardization may be manifested in major differences between data base models (conceptual organization of the data) that are used by the institutions and therefore in differences of the data types that are returned by data base queries. Even if the variables

[22] Even if some structures of the Arden Syntax are similar to common programming languages, MLMs usually have to be compiled (translated) into a format that can be executed within the decision support system. In recent projects MLMs were converted into program code, such as C++, into a 'pseudo-code' that is used by an interpreter, or into a knowledge representation format that is used by an expert system shell.

[23] http://protege.stanford.edu/index.html

are not bound to a pre-defined data type and can therefore represent the data, the logic may have to be altered as mentioned earlier. Further problems may be caused by different measurement units for numerical data base entries [HPW95].

Columbia Presbyterian Medical Center to the LDS Hospital [PH94]. On account of differences of the data structures in the data bases, the receiving institution had to redefine not only the data to symbol conversion by updating the data base queries, but even elements of the decision logic. Overall, half of the modifications affected the logic of the MLMs. These problems might not only arise when knowledge is to be shared, but also affect the maintenance of the knowledge base when data base structures within one institution do change [JHHC98].

This problem of non-standardized data base queries is actually known as *curly braces problem* and is one topic in the responsible working groups of HL7. Some implementations of Arden Syntax-based systems made use of different terminology systems [ASG+94,Lud94, Dup94]. However, a standardized component of the Arden Syntax that can be used to define data base queries is planned not before version 3 of the Arden Syntax in the near future.

1.4.2 Arden in use

Since the early 90s, Arden Syntax is being used at several institutions that published papers about the evolution of their systems.

Since the early 70s the LDS Hospital uses a decision support system known as HELP. As Arden was derived from the underlying syntax, the LDS Hospital partially adapted their system rather early to the new Arden Syntax. 3M corporation implemented an Arden Syntax compiler that translated Arden Syntax MLMs into an intermediate language, which then was compiled into the HELP system. The system was used for data driven alerts and remainders, such as monitoring laboratory values, and for clinical protocols. Messages were delivered either by terminal messages, as printed reports, or by a nursing electronic beeper system [Pry94]. Today the Arden Syntax is not the main representation format used in the LDS Hospital.

The Columbia Presbyterian Medical Center implemented an Arden Syntax-based system that has been linked to the institution's patient data base and was able to route and display alert messages. The system was based on a compiler which translated Arden Syntax MLMs to pseudo-codes that were then executed (or, more exactly, interpreted by an interpreter). One conclusion of the implementation was that data base queries take up most of the execution time. Thus the time loss due to the interpretation of the p-codes has not been significant [HCJC92]. Another conclusion was to rearrange the order of operations during the process of compilation: If decisions depend on certain data and the result of the decision would imply reading more data, only data that are relevant for the decision should be queried from the data base rather than reading the complete data first. However, this partial annihilation of the separation of data and logic is made during runtime and the MLMs still define the queries separately from the decision logic.

An Arden Syntax-based expert shell at the University of Giessen translated MLMs into PL-SQL code that was directly used within an Oracle data base [Taf99]. Additionally, information required for the storage and evocation of MLMs are stored in a set of data base tables. Special focus was given to the detection of "unjustified" messages that, for example, may be the result of the continuous execution of rules whereas their resulting messages are

read asynchronously. The system identified and removed outdated or duplicated messages automatically [TAW+99].

At the University of Linköping MLMs have been translated into the C++ programming language and used within an expert system for tasks in the laboratory, such as real-time validation of laboratory test results [Gao93, Joh97]. A different approach was adopted in the Massachusetts General Hospital: a commercial expert shell was used and Arden Syntax MLMs were manually converted into the needed format [JMB94]. They used the Arden Syntax as formalized and uniformed format for a textual library from which knowledge can be extracted and converted if needed.

1.4.3 Current state

The Arden Syntax became an ASTM standard in 1992 [Ast92] and is maintained by the Health Level Seven, Inc. since 1998 [Hls99]. Within HL7 the Arden Syntax Special Interest Group (Arden Syntax SIG) that is hosted by the Clinical Decision Support Technical Committee (CDS TC) is responsible for further development of Arden Syntax. The group meets regularly three times a year and has recently defined the latest version 2.1 of Arden, which includes minor changes in the syntax, such as some new operators and a first approach for an XML representation of Arden Syntax MLMs.

At the end of 2002 current discussions regarding Arden Syntax mainly concern the handling of complex data types and solving the curly braces problem. In the last year the former HL7 Arden Syntax TC was renamed the Clinical Decision Support TC; since this time it is hosting two 'Special Interest Groups' (SIGs), the Arden Syntax SIG and the Clinical Guidelines SIG. The latter consists of members of the old Arden Syntax TC and new members who contribute their knowledge and experience about guideline formats to a common guideline standard.

The main topics of the Arden Syntax SIG in the last two years, and therefore changes to the Arden Syntax Version 2.0, were the following:

Structured reports (v2.1):
Before Motorola quit their efforts in HL7 they proposed a *structured write* statement that could be used for a unified specification of messages. The structure was based on an XML document definition and included, in addition to the message, one or more receivers, some timeouts for the delivery and acknowledgement of the message, and information about follow-up actions. This extension has been included in the latest Arden Version 2.1 which is currently opened for balloting.

New operators (v2.1):
A slight extension of the syntax has been proposed by McKesson; it provides some new operators but does not change basic principles of Arden Syntax. These extensions mainly improved the handling of string data [Hls02b][24].

Representation of uncertain knowledge (v2.x/v3):
The ability to represent uncertain knowledge was desired even before the current work on Fuzzy Arden was started. In an early harmonization state between Arden Syntax and

[24]The need for an operator to extract substrings has also been identified when implementing the Arden Syntax system at the University of Giessen [Taf99].

the guideline format GLIF[25], some new operators for Arden Syntax were proposed; one of them was termed 'is unknown'. At that point the committee decided not to include it as it "would introduce a four-state logic and would significantly alter the behavior of the current Arden logical operators" [Hls00a].

Obviously the extensions proposed in the following chapter also alter the behavior of the syntax; the changes are not restricted to the logical operators. However, as shown later, the "fuzzy concepts" can be perfectly integrated without violating any principles of the Arden Syntax. All proposals included in this thesis have been presented to the Arden Syntax SIG at the past four HL7 working group meetings and were discussed with great interest (compare [Hls01a] to [Hls02a]). Currently the SIG expects an extended specification document that incorporates the extension of discussion as a working item for final decisions.

Harmonization with Guidelines SIG/Curly braces problem (v3):
Another topic within HL7 is harmonization with the future guideline format of HL7. The Clinical Guidelines SIG was created indirectly at a workshop in March 2000 in Boston by the InterMed[26] group. This workshop on clinical guidelines had the purpose of bringing together people from universities, industry, and other healthcare-related organizations who are interested in a unified and standardized electronic guidelines format. In the course of 2000 the GLIF group decided to join HL7 and it was decided to create a guideline-related SIG which was attended by several members of other electronic guideline groups as well [Hls00a, Hls00b].

The first meeting of the new GLIF SIG was held in January 2001. Owing to the attendance and contributions of members of other guideline groups, such as Prodigy[27] or GEM[28], the SIG has been renamed the Clinical Guidelines SIG [Hls01f]. As the SIG decided not just to adopt GLIF but to aggregate the concepts and ideas of all participating groups into a single standard, the work has been separated into individual topics, such as an XML architecture for electronic guideline documents, a flowchart format for the graphical representation, and terminological aspects.

Initially the Arden Syntax was intended to be used as an expression language for the representation of conditional statements and data base queries. However, soon it emphasized that one main feature of the Arden Syntax—the simplicity—avoided the use of a guideline format as it would have to be able to operate on complex data structures (objects). The group presented alternative approaches such as the expression language GEL (Guideline expression language) or an object-oriented query language [Hls01e]. These approaches led to discussions within the TC and the Arden Syntax SIG as, on the one hand a common expression language within HL7 was desired but, on the other hand the Arden Syntax group did not want to significantly alter its syntax. The main concerns about such significant changes came from the industrial attendees as they feared that a very important feature—the simplicity—of Arden Syntax would be lost.

The work on the expression language continued independent of the Arden Syntax. As a result the expression language GELLO defines an object-oriented format for queries (which can be compared to the curly braces constructs in Arden Syntax) and for referencing results of queries (thus for defining conditional statements) [Hls01d]. Comparisons of

[25] Guideline Interchange Format, http://www.glif.org
[26] http://www-camis.stanford.edu/projects/intermed-web/
[27] http://www.prodigy.nhs.uk
[28] http://ycmi.med.yale.edu/GEM/

queries and conditional statements implemented by GELLO and implemented by Arden Syntax showed that such an extended syntax is required for processing data objects. As a consensus it was decided, that GELLO should be able to use HL7 data types (and associated methods) but should not include control structures such as if-then or assign statements [Hls02c].

So far GELLO is under development as a separate product in addition to the Arden Syntax. The most recent discussions showed that the Arden Syntax SIG and the involved vendors mainly have concerns regarding the use of object-oriented methods in Arden. However, GELLO can be used to define database queries in the curly braces.

In parallel the Arden Syntax will be extended stepwise to be able to handle objects. Initially the data types are to be extended by a dot notation that allows one to define objects as "record structures". A variable could then represent more than one value, each identified by a label and accessible by concatenating the label of the variable and the label of the "sub-value" by a dot. For example, the diastolic value of a blood pressure observation could be accessed by '`BPObservation.diastolic`'. With such an extension, Arden Syntax could operate on complex objects on the one hand, but would not lose its simplicity on the other [Hls02d].

Chapter 2

Fuzzy Arden Syntax

Since the Arden Syntax does not provide elements to represent vague knowledge, it has been cautiously extended by concepts of fuzzy set theory, fuzzy logic, and fuzzy control. Basically fuzziness might have to be formalized in those algorithmic parts of an Arden Syntax rule that include vaguely defined concepts, which are represented by expressions that use comparison operators. For example, the concept 'fever' can be formalized by using an 'is within' comparison operator that has fuzzified upper and lower limits for defining the valid range of temperature values. Generally, this type of representing fuzziness concerns most expressions that include selection criteria, such as data base queries or conditional expressions. Furthermore, the resulting 'fuzziness' (the degree by which the numerical value matches a fuzzified condition) must be processed and taken into account during the execution of the algorithm.

The first section provides a brief introduction of the elements of the Arden Syntax of version 2 that are needed to define MLMs. The description is brief; more detailed information is available in the specification document from HL7 [Hls99] or in the publications mentioned in the introduction.

The second section describes the conceptual models for the extension of the Arden Syntax, which are then applied in section 2.3 to elements of the syntax. The final section defines further extensions that can be used to realize linguistic variables and fuzzy-control-like knowledge bases by Arden Syntax.

2.1 Elements of the Arden Syntax

A knowledge base that is represented by Arden Syntax is composed of a set of independent rules known as Medical Logic Modules (MLMs). Each MLM should include sufficient knowledge for making one specific decision, such as generating an alarm message or providing diagnostic theories.

Usually an MLM is stored in an ASCII text file. Figure 2.1 shows one sample MLM that is published in the Arden Syntax specification.

Each MLM is structured by three categories: maintenance, library, and knowledge. Each category includes entries (slots), which are composed of a name and a corresponding value, either as free text, coded by keywords, or structured by statements and operators. The maintenance category contains slots with general information, such as title, source, version, and status. The library category contains slots with explanatory information, key words, and citations of other sources of information.

```
maintenance:
    title: Check for penicillin allergy;;
    mlmname: pen_allergy;;
    arden: ASTM-E1460-1995;;
    version: 1.00;;
    institution: Columbia-Presbyterian Medical Center;;
    author: George Hripcsak, M.D.;;
    specialist: ;;
    date: 1991-03-18;;
    validation: testing;;
library:
    purpose:
        When a penicillin is prescribed, check for an allergy.
        (This MLM demonstrates checking for contraindications.);;
    explanation:
        This MLM is evoked when a penicillin medication is ordered.
        An alert is generated because the patient has an allergy
        to penicillin recorded.;;
    keywords: penicillin; allergy;;
    citations: ;;
knowledge:
    type: data-driven;;
    data:
        /* an order for a penicillin evokes this MLM */
        penicillin_order := event {medication_order where
                                    class = penicillin};
        /* find allergies */
        penicillin_allergy := read last {allergy where
                                        agent_class = penicillin};
        ;;
    evoke:
        penicillin_order;;
    logic:
        if exist(penicillin_allergy)then
            conclude true;
        endif;
        ;;
    action:
        write "Caution, the patient has the following allergy to"
           || " penicillin documented: " || penicillin_allergy;;
    urgency: 50;;
end:
```

Figure 2.1: Sample MLM: contraindication alert (taken from [Hls99])

The knowledge category provides all information that is needed for an expert system to evaluate and execute the rule. “Fuzzy extensions” concern most of the knowledge slots:

Data slot: The data slot maps institution-specific data to local Arden Syntax variables by defining data base queries, events, message destinations, and references to subrules (other MLMs) or external programs.

All institution-specific definitions are formalized within curly braces and are not part of the standard specification. To minimize non-standardized parts of an MLM, conditional parts of data base queries, such as temporal restrictions, can (and should) be defined “outside” the curly braces by using Arden Syntax operators.

Logic slot: This slot defines the decision criterion of the rule that returns either ‘true’ or ‘false’. The decision is usually based on the data stored in variables that have been defined in the data slot. Arden Syntax provides a simple and easily readable

programming language that can be used to alter and evaluate this data; therefore the representation of the decision criterion is basically algorithmic in nature.

Action slot: The action slot is executed if the decision logic arrived at a positive conclusion. Usually an MLM generates a message that is sent to a destination defined in the data slot. However, the action is not limited to sending messages; it can also be used to return data to an MLM that called the current one or to call other MLMs. However, the option for user interactions is not part of the syntax but might be implemented as a special destination for a message[29].

Evoke slot: The evoke slot defines when the MLM has to be executed. The execution of an MLM can be triggered by an event that has to be defined in the data slot (the event slot uses the local variable that was used in the data slot). Optionally the execution can be started delayed and/or in a cyclic manner after the event occurred. The slot is empty if the MLM is intended to be called directly, for instance by another MLM.

2.1.1 Data types

The basic function of an MLM—decision-making—is based on query, manipulation, and evaluation of data. The Arden Syntax comprises different data types for representing numerical values, character strings, truth values, temporal values, and lists. Unknown data or data that result from invalid operations are represented by 'null'.

Boolean Valid values of this data type are 'true' and 'false'. However, logical operators are defined by a trivalent logic which is explained in the next section.

Number Arden Syntax has one data type for numerical data and does not distinguish between integer and real numbers; all arithmetic is done internally in floating point numbers. Number constants can be defined by one or more digits (0 to 9) and an optional decimal point, and may end with an exponent, represented by an E or e. For example:

2.4E-12 or 42

Time dates are represented in an extended ISO 8601-1988 format. Dates are defined as yyyy-mm-dd, times by yyyy-mm-ddThh:mm:ss with 'T' or 't' as separator[30]. Optionally, fractional seconds and a time zone can be defined. For example:

2002-07-07T13:43:00

Arden Syntax provides keywords that define time constants. 'Now' defines the time of execution of an Arden Syntax rule, 'eventtime' defines the date of the event that evoked the rule, and 'triggertime', the time of a delayed rule evocation.

Duration Intervals in time without an anchor to any particular point in time can be represented by duration values. For example:

[29] Message destinations are institution specific and can therefore be defined arbitrarily within curly braces.

[30] Except for very few elements, Arden Syntax is not case sensitive.

3 years or 0.1 months

The print output of durations is specific to the duration that has to be printed, for example the duration 26 hours could be printed as '1 day, 2 hours'. Internally all durations are represented either as months or seconds. Conversions from one representation to another are regulated by the specification.

String This data type represents streams of characters of variable length. Strings are delimited by double quotes. For example:

"HL7" or "Arden Syntax"

Term Terms are constants strings for special use, such as for defining MLM names in MLM references. Terms are delimited by single quotes. For example:

'pen_allergy'

Lists Lists are ordered sets of data values of any data type except lists itself and may be heterogeneous. For example:

(42, 1975-07-05, "HL7", null)

Each data value is associated with an additional attribute 'primary time'. The primary time defines the most relevant time stamp for the data value, for example the time of an examination. Each data value that is the result of a data query should have a primary time. If data are defined in the logic slot, for example by assigning constants to variables, or if data are manipulated by operators that lose the primary time, their primary time may be 'null'.

Arden Syntax does not provide data types to handle complex data such as objects. In object-oriented programming languages, data collections can be "hidden" behind object definitions, which provide one data type for the entire collection. For example, an object-oriented data type for blood pressure observation could include information about the systolic and diastolic pressure, the observation method, and a reference to a patient. Further, objects provide methods for accessing and manipulation data in them.

2.1.2 Logic

The Boolean truth values can be either 'true' or 'false'. A trivalent logic is used for logical operations. The classical operators 'and', 'or', and 'not' are defined to operate on the Boolean truth values and additionally on 'null'. Table 2.1 shows the truth tables for Arden Syntax logic operators. Arguments that are not Boolean values are handled as 'null'.

According to the specification, the third value 'null' expresses uncertainty. The uncertainty may arise from missing data in the database, errors during the execution of operators (e.g., division by 0), or due to an explicit assignment of 'null'. This type of uncertainty rather signifies incompleteness than vagueness (compare section 1.2). Therefore, the interpretation of 'null' as uncertainty is not inconsistent with fuzzy logical extensions presented in this work.

Table 2.1: Arden Syntax logical operators truth-tables

and	true	false	null
true	true	false	null
false	false	false	false
null	null	false	null

or	true	false	null
true	true	true	true
false	true	false	null
null	true	null	null

not	true	false	other
	false	true	null

2.1.3 Programming language

Arden Syntax provides a simple programming language to define the algorithms in the structured slots (data, logic, and action), which is similar to Pascal or Basic. Variables are used to store and manage data, operators are used to build expressions that manipulate data, and statements are used to define the algorithm by combining expressions and to control its execution.

2.1.3.1 Variables

A variable is a temporary place holder for a data value and is defined by an identifier (*variable name*). In contrast to other programming languages, such as Pascal, C, or Java, variables need not be explicitly defined before they can be used. Further, a variable is not bound to an explicit data type. During the execution of the algorithm it may represent data of different data types.

As Arden Syntax only provides primitive data types and as lists cannot be nested, objects with more than one value that are the result of data base queries have to be split into their single data values and stored in different variables. For example, a blood pressure observation object that includes the diastolic and systolic pressure has to be separated into two individual variables.

2.1.3.2 Operators

Operators are used to manipulate data. Arden Syntax classifies unary, binary, and ternary operators, which accept one, two, or three arguments. Most operators are defined to work on a subset of the data types. Their formal definitions declare the valid types of valid arguments. As variables cannot be defined explicitly, expressions that contain variables cannot be verified by syntactical rules in means of data types[31].

The Arden Syntax specification classifies the operators additionally by their functionality:

List operators for the creation, manipulation, and for sorting lists.

Logical operators define ‘and’, ‘or’, and ‘not’ for logical operations.

Simple comparison operators provide the comparisons ‘greater than’, ‘less than’, ‘equal’ and combinations of them. The common symbolic operator

[31] With more effort it might be possible to verify parts of the algorithms where constants are assigned to variables but all operations that depend on data from the data base cannot be verified.

names, such as >, <, and =, can be synonymously used in addition to the linguistic operator names.

Is comparison operators provide additional comparisons for interval comparisons. Further, this class of operators includes data type comparisons.

Occur comparison operators can be used for primary time comparisons.

String operators provide string manipulation and formatting.

Arithmetic operators provide arithmetical operations for numbers, times, and durations.

Temporal operators can be used to compute relative dates (for example, '2 hours after time of event1' or '2 days ago').

Duration operators provide methods for the creation of durations and the extraction of single information of date, such as the day or month of a given date.

(Query) aggregation operators are mostly defined for manipulating or aggregating lists[32].

(Query) transformation operators also work on lists and provide the selection of elements based on different orders or the computation of relations between the elements, such as the difference from element to element.

Numeric function operators provide mathematical operations that are defined on numbers and one additional operator that generates numbers from strings or Boolean data. This 'as number' operator generates the numerical value 1 from a 'true' and the numerical value 0 from a 'false'.

Time function operator returns the primary time of a data value.

2.1.3.3 Expressions and Statements

Strictly speaking, expressions consist of operators, constants, and variables, and are used within statements. Statements define the algorithm of the decision logic. More generally, the term 'expression' also refers to the entire expression made of statements and expressions.

The most basic statement is the 'assignment' of data to a variable. The data itself is represented by an expression. For assignments, both the symbolic representation ':=' and the linguistic representation 'let ... be' are valid. The following example shows three assignments of a string value to a variable 'error_msg'.

Example 4: (Use of assignment statements)

```
error_msg := "no valid value for 'weight'";
let error_msg be "no valid value for 'weight'";
error_msg := "no valid value for " || error_cause;
```

The last assignment additionally includes a string concatenation operator that appends the value of the variable 'error_cause' to the string constant.

[32]The selection of a single element out of a list is defined also in this class of operators, even if it would be reasonable to classify it as a list operator.

An 'if-then' statement defines one or two code blocks whose individual execution depends on a conditional expression. If the expression yields 'true' the first code block is executed. If it yields 'false' (or more exactly, if it yields any other value) and a second code block is defined by the keyword 'else', the second one is executed. Finally, the program flow of the algorithm continues with the statement that follows on the 'if-then' statement. 'If-then' statements can be nested, as shown in the next example.

Example 5: (Nested if-then statements)

```
0: msg :="Body Mass Index = " || bmi;
1: if bmi >= 27.8 and gender = "M"  then
2:    msg := msg || " which is above the 85th percentile for men.";
3: elseif bmi >= 27.3 and gender = "F" then
4:    msg := msg || " which is above the 85th percentile for women.";
5: endif;
```

First, the constant part of the message is prepared (line 0). If the body mass index (BMI) of the current patient, which is represented by the variable 'bmi', is at least 27.8 and the gender of the patient is male (line 1), then the message is extended by a warning that the patient's BMI exceeds the recommended range (line 2). If the patient's BMI is either below the first condition or if the patient is not male, the second condition is evaluated (line 3). If it yields 'true', then the message is extended analogously (line 4). Lines 3 and 4 represent an 'if-then' statement that is nested within another one.

Loops can be realized by the 'while... do' loop and 'for... in... do' loop statements. The first one cyclically executes a code block as as long the condition yields 'true'. The second one uses a variable and a list as arguments and executes a code block as many times as the list has elements. In every iteration the variable represents the corresponding element of the list.

Example 6: (Loop statements)

```
 0: num := 1;
 1: msg := "The following patients are eligible for the study: ";
 2: while num <= (count patients) do
 3:   a_patient := patients[num];
 4:   is_eligible := call test_study with a_patient;
 5:   if (is_eligible is equal true) then
 6:      msg := msg || " a_patient, ";
 7:   endif;
 8:   num:= num + 1 ;
 9: enddo;
10:
11: for a_patient in patients do
12:   is_eligible := call test_study with a_patient;
13:   if (is_eligible is equal true) then
14:      msg := msg || " a_patient, ";
15:   endif;
16: enddo;
```

In this example both loops are used to browse through a list of patient ids or patient names. The first one uses a separate counter variable (line 0) to access the list element (line 3). The counter is increased in every iteration of the 'while loop' (line 8) until it exceeds the number of elements of the list (line 2).

The second one automatically accesses the list element by using the 'for loop' statement (line 11).

Both algorithms call another MLM to prove whether the patient is eligible for a study and append its identifier to a textual message (lines 4 to 7 and lines 12 to 15).

2.2 Conceptual models of the extensions

One reason for the fuzziness of vague concepts is the process of abstraction from real-world observations and measurements to terms. Often, terms and single characteristics of abstract concepts are defined within a given range of real world data, as shown exemplarily by the concept of 'fever'.

Example 7: (Definition of fever)

The concept "temperature of human body" ('body temperature') may be defined in the range of 15 to 45 degrees Celsius. In means of fuzzy set theory, this range can be termed the universe of discourse.

The characteristic "fever" can be defined for a sub-range of the universe of discourse, say from $37.0°C$ to $38.0°C$.

An action that depends on the decision whether a patient's temperature could be termed fever or not can be represented by an Arden Syntax comparison as shown in the following example.

Example 8: (Definition of fever with Arden Syntax)

Presuming that the variable 'temperature' represents a valid numerical value, a decision based on the condition whether the patient has fever can be implemented as follows:

```
if temperature is within 37 to 38 then ... endif;
```

If the temperature represents any value from 37 to 38 (including the upper and lower limit), the code indicated by the three dots is executed.

This definition of a conditional expression leads to known problems concerning crisp definitions, as pointed out in the introduction. Values which are infinitesimally smaller than the lower threshold or infinitesimally greater than the upper threshold would unintuitively result in a completely different decision than values that are equal to the thresholds. Conditional expressions similar to this example are widely used in any parts of an MLM that include selection criteria, for example when selecting data from a database or for branch conditions in the logic slot (such as 'if-then' or 'while' statements).

Therefore, to consider the fuzziness of medical concepts, those elements of the Arden Syntax have to be extended that are used to express conditional expressions as shown in the last example. Ostensibly, the extensions will mainly affect comparison operators, such as 'is greater than' or 'is within ... to'. Further, the extensions will affect the logical data type as well as operators or conditional program statements, such as the 'if-then' statement, which have to be able to interpret fuzzy truth values that are returned by fuzzily defined expressions.

2.2.1 Fuzzy operators: fuzzy sets and fuzzy truth values as model for fuzzy comparisons

A traditional Arden Syntax comparison operator compares data values and returns a crisp truth value. Arden Syntax provides comparison operators for

- simple comparisons, such as 'is greater', 'is less', or 'is equal'
- more complex comparisons, such as the comparison to a range of values
- comparisons of string values, such as the 'matches pattern' operator

The concept of fuzzy sets is applied to *fuzzified comparison operators* that compare numerical values and return fuzzy truth values instead of crisp ones. (The comparison of string values cannot be fuzzified by applying fuzzy sets, as the universe of discourse is not numerical and the comparison of strings may include aspects like the semantic meaning of strings or even their pronouncement.)

Definition 6 (fuzzy behaving operator, fuzzified by):

> *A* **fuzzy behaving operator** *or* **fuzzy operator** *is a comparison operator that uses numerical arguments which can represent a crisp threshold or a fuzzily defined threshold.*
>
> *If an argument has to represent a fuzzily defined threshold, the gradual transition from 'true' to 'false' is defined by an additional parameter specified by the keyword* **fuzzified by**.
>
> *The result of a fuzzy operator is always a fuzzy truth value that is equal to the degree of compatibility of the argument to the condition.*

As the result of a fuzzy operator could be a truth value that is neither 'true' nor 'false' but a value in between, the former Boolean truth data type is extended to support truth values and is termed *fuzzy truth value*. First, the range of Boolean $\{\text{false}, \text{true}\}$ has been interpreted as $\{0, 1\}$ where 0 corresponds to 'false' and 1 to 'true'. Then, the two truth values were extended to a range of truth values $[0, 1] \in \mathbb{R}$ to specify a degree of truth.

Definition 7 (fuzzy truth value):

> *A* **fuzzy truth value** *represents gradual truth values from 'true' to 'false', represented by a number:*
>
> $$[0, 1] \in \mathbb{R}$$
>
> *The former values 'false' and 'true' now correspond to the boundary values* 0.0 *and* 1.0 *and can be used synonymously. Just like any variable, a variable that represents a truth value can additionally take on the value 'null' whereby logical operations have now to be defined on*
>
> $$[0, 1] \in \mathbb{R} \cup \{null\}$$
>
> *Fuzzy truth constants are notated by a number with a leading F, such as $F1.0$, $F0.4$, or $F0.0$, where 'true' can be used synonymously with $F1.0$ and 'false' with $F0.0$.*

By using these extensions the condition whether a patient has fever as defined in example 8 can be represented as shown in the following.

Example 9: (Fuzzy definition of fever)

A fuzzy definition of the characteristic "fever" may be defined by fuzzifying each threshold by 0.5 and can be implemented by Fuzzy Arden as:

if temperature is within 37 fuzzified by 0.5 to 38 fuzzified by 0.5 then ... endif;

Compared to the former crisp definition, this conditional expression extends the range of valid arguments on both sides[33]. The dimension of the extension is controlled by the 'fuzzified by' keywords (compare figure 2.2).

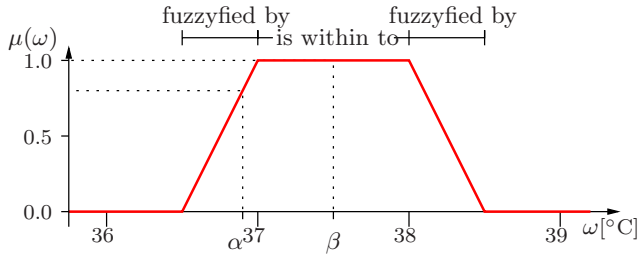

Figure 2.2: Membership function of a fuzzified 'is within to' operator

The body temperature value $\alpha = 37.9$ has a degree of compatibility of 0.8 to the fuzzy set, thus the result of the comparison would be a fuzzy truth value of $F0.8$. The body temperature $\beta = 38.5$ has a degree of compatibility of 1.0 to the fuzzy set, thus the result of the comparison would be 'true'.

2.2.2 Fuzzy data: effects of conditional contexts on algorithmic elements

So far fuzzy operators can be used to model vaguely defined medical concepts. The result of a fuzzy comparison is a fuzzy truth value that describes the degree of compatibility of the argument to the condition. Now fuzzy truth values can influence conditional expressions, such as the selection of data values or branches in program flow.

Example 10: (Conditional expression: selection of data)

```
res := () where time of it is within past 6 hours fuzzified by 30 minutes;
```

This expression uses the 'where' operator that selects from the list used as left argument those elements where the corresponding element of the list on the right side is 'true'. The expression 'is is within...' yields a list of truth values that represent the results of the comparison operator applied to each element of the original list (the right argument of the 'where' operator represented by the keyword 'it').

The selection in example 10 is based on the condition whether each element is not much older than six hours. The individual fuzzy truth value is defined by the degree of compatibility of each element to the condition. While the classical behavior of the 'where'

[33] It has been mentioned that the definition of fuzzy sets might significantly depend on the author of a rule, and that such definitions may be subjective.

operator is defined unambiguously (the corresponding element is only chosen if the current element is 'true'), fuzzy truth values as condition require a different approach as to how values should be selected or not selected. If, for example, the current truth value is $F0.7$ — should the corresponding element be chosen or not? A similar situation occurs whenever fuzzy truth values are used by 'if-then' statements.

Example 11: (Conditional expression: program flow branch)

```
if age is greater than 18 years fuzzified by 6 months then
    [..]
else
    [..]
endif;
```

The conditional expression in example 11 controls the program flow. Like a traditional crisp truth value the fuzzy truth value defines whether the dependent code is applicable or not–if it is 'false', the code has not to be executed, if it is 'true' it has to. Thus, a fuzzy truth value defines the *degree of applicability* of the two code blocks. As a fuzzy truth value can be neither 'true' nor 'false', the degree of applicability is defined gradually too.

Definition 8 (degree of applicability):

> *Program statements that require a decision between the execution of two code blocks include a conditional expression that yields a fuzzy truth value. The degree of truth defines the individual* **degrees of applicability** *of both code blocks.*
>
> *The degree of applicability of the code block that should be executed if the condition yields 'true' is equal to the fuzzy truth value. The degree of the other code block determined by the negation of the truth value.*

The problem of choosing the right code block to be executed is still to be solved. If the degree of applicability is only partially 'true', the code should be executed—eventually. This problem of binary decisions that depend on fuzzy truth values has been already presented in the introduction of fuzzy algorithms (1.3.4).

Whenever the condition returns a truth value that is neither clearly 'true' nor 'false', a binary decision can no longer be made. The methods to handle such fuzzy algorithms presented in the introduction had been applied in their original publications on the selection of one element of a fuzzy set, for instance the number of steps to be made according to the instruction "make several steps", or realized as fuzzy relation between two fuzzy subsets. Applying these concepts to the decision as to which element of a set with two elements (the alternative code blocks) should be chosen, allows following solutions:

1. Probabilistic execution: The degree of applicability defines the probability of execution of the code blocks. Only one of the code blocks is executed.
2. Execution with threshold: A crisp threshold greater than 0.5 is defined: That code block whose degree of applicability is greater than or equal to the threshold is executed, the other one is not.
3. Parallel execution: Both code blocks are executed in parallel in every case where the condition is not clearly 'true' or 'false'. As defined by the compositional rule of inference, both alternatives gain as much influence on the result as the condition defines.

The first and second method behave rather crisply while the third can consider the fuzziness of the condition by assigning due importance to each alternative. Therefore, Fuzzy Arden mainly uses this method to handle fuzzy decisions whereas the 'threshold' approach is used for special situations. (The formal definitions will be made in the next section when defining the conditional statements and the 'where' operator.)

According to the compositional rule of inference, all expressions within the code blocks are influenced by the degree of applicability of the corresponding block. This degree is termed the 'conditional context' of the expressions.

Definition 9 (conditional context):

> *Statements and operations whose execution depends on a fuzzy conditional expression have to consider the resulting uncertainty as their* **conditional context**. *The conditional context indicates by a degree from $F0.0$ to $F1.0$ whether the execution of the affected statements or expressions is "justifiable" or not.*
>
> *Regarding conditional statements (if-then statement, while loop), the conditional context is defined by the degree of applicability of the code blocks. Regarding the 'where' operator, the conditional context is defined by the degree of compatibility of its conditional expression*[34].
>
> *If the conditional context is less than $F1.0$ it is termed* **reduced**. *The root conditional context is always set to $F1.0$, as it only depends on the occurrence of events (which start the execution of the MLM)*[35].

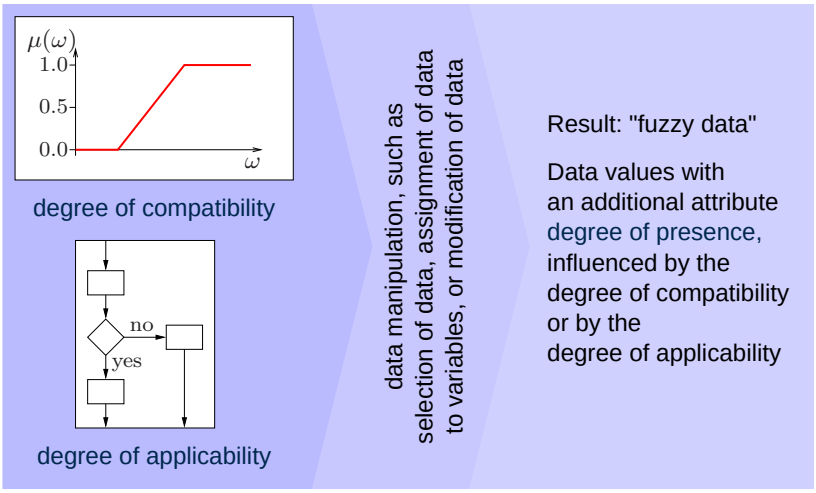

Figure 2.3: Conditional context

Figure 2.3 illustrates the two situations that cause a conditional context. Basically the source of the context is the degree of compatibility of a value to a fuzzy condition (for example, when using the 'where' operator). Another source is the degree of applicability

[34]The 'where' operator is a special case, as it does not include any other statements or expressions. However, its functionality of selecting or not selecting elements from a list is influenced by the conditional context.

[35]If Arden Syntax would include the concept of fuzzy events, which could be used as "fuzzy eligibility criteria" for the execution of an MLM, then the root conditional context would also be reduced.

of an entire code block (which, finally, is the result of the degree of compatibility of a value to the conditional expression used by the conditional statement).

If a data value is assigned to a variable within a conditional context of an 'if-then' statement, the execution of the assignment is influenced by the conditional context. If the condition yielded a crisp 'true', the assignment would have been definitely executed, if it was 'false', it definitely would not have been executed. Therefore the data represented by the variable would be either present or not.

If the condition yielded a fuzzy truth value the conditional context would neither be 'true' nor 'false'; its influence on the assignment can be interpreted as an indicator as to whether the value is present or not, in other words, as indication of the *presence* of the data. The closer the conditional context gets to 'true', the more present are the corresponding statements, operators, and finally the data. The more the context is reduced, the more do the elements "fade away" until the context is 'false' and they would no longer be executed. Analogously, the selection of data based on a partially fulfilled conditional expression of a 'where' operator can be interpreted as to how "present" the selected elements are. This indicator is termed the *degree of presence* of data.

Definition 10 (degree of presence):

> *Arden Syntax data is extended by an additional attribute* **degree of presence** *that is defined by the conditional context in which a data value has been selected, assigned, or modified. The degree of presence can take on values from* 0.0 *to* 1.0*.*
>
> *By default, the degree of presence of data is* 1.0*. It is only reduced if the selection or manipulation depends on a condition that is partially true and cannot be directly assigned. In this document the following notation is used for the visualization of the degree of presence:*
>
> {<data value>, <degree of presence>}
>
> *The visualization is not part of the syntax and therefore not usable in MLMs to assign the degree of presence manually.*

The degree of presence of a data value describes the context of its creation or modification. To differ data that has been created or modified in a "classical" crisp way from data with a reduced degree of presence, these two "types" are termed 'crisp data' and 'fuzzy data'. These classes of data are only defined for better comprehension and do not technically define new data types, such as numbers or strings. Strictly speaking, both types of data are crisp, as a data value is nothing but itself[36].

Definition 11 (fuzzy data, crisp data):

> *Data that have a degree of presence less than* 1.0 *and greater than* 0.0 *are termed* **fuzzy data***. Data that have a degree of presence of* 1.0 *are termed* **crisp data***. Data with a degree of presence of F*0.0 *are equal to the 'null' value.*

[36] In special, there is no relationship of "fuzzy numerical data" to the concept of 'fuzzy numbers'. Fuzzy numbers are special fuzzy sets that not only represent themselves, but may partially represent also other values simultaneously. Fuzzy data of data type 'number' represent crisp numbers that have a reduced degree of presence.

Lists are handled separately. They represent heterogeneous collections of data values and can contain both fuzzy and crisp data. Their degree of presence is always 1.0.

The following example illustrates the concept of fuzzy data by a selection of data that is controlled by a temporal condition.

Example 12: (Compute average of test results based on fuzzy selection criterion)

The average of test results of the past three days is to be computed. In order to additionally consider those test results which were collected shortly before this period, the range of selected data was stretched by 12 hours using a fuzzily defined selection. This selection is illustrated in Figure 2.4. t_0 defines the actual time of execution, t_1 to t_5 are time stamps of test results collected within the preceding 84 hours.

Two of these five selected test results, collected at t_4 and t_5, are within the stretched range and would not have been considered in a crisp selection. However, by making a fuzzy selection, their respective values influence the result of this operation as well.

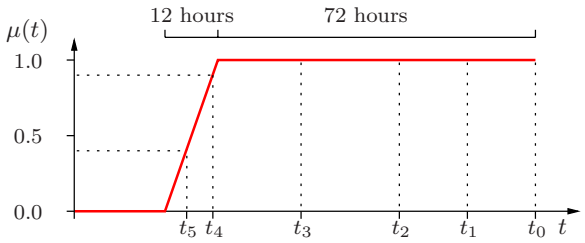

Figure 2.4: Selection of test results from a previous period

In Fuzzy Arden, this task can be formulated in the following way:

read average {} where it occurred within past 3 days fuzzified by 12 hours

First, an institution-specific instruction within the curly braces[37] returns a list of crisp data elements from the database:

{} ⇒ (42, 40, 38, 41, 42)

The next operation contains a fuzzy operator and returns a list of fuzzy truth values:

it occurred within past 3 days fuzzified by 12 hours ⇒ (F1.0, $F1.0$, $F1.0$, $F0.9$, $F0.4$)

These values represent the degree of compatibility of the collection time of the single test results with the expression 'within past 3 days', which is defined as a fuzzy set with Δx set to twelve hours.

Next, the 'where' operator selects elements from its left argument based on the truth value of its right argument:

[37] The space within the curly braces was intentionally left blank in the example.

(42, 40, 38, 41, 42) where (F1.0, F1.0, F1.0, F0.9, F0.4) ⇒
({42, 1.0}, {40, 1.0}, {38, 1.0}, {41, 0.9}, {42, 0.4})

In this step, crisp data and fuzzy truth values are combined to fuzzy data.

The last operator computes the average of the values and must consider the fuzziness of individual data elements. The average operator is defined in a later section. Finally, the operation results in {40.4, 1.0}, whereas the crisp operation would have computed only the average of the first three values returning 40.

2.2.2.1 Use of crisp operators within a reduced conditional context

Whenever an operation is executed within a conditional context, the degree of presence of its result cannot be higher than the conditional context. For example, if two crisp strings are concatenated within a reduced conditional context, the degree of presence of the resulting string is equal to the conditional context. If the strings were already fuzzy data whose degree of presence was smaller than the conditional context, it would not have further influence on the result of the operation.

Definition 12 (Influence of the conditional context on the degree of presence):

The degree of presence of the result of an operator is always less than or equal to the conditional context that embeds the operator.

The following example illustrates this rule.

Example 13: (Influence of the conditional context on the degree of presence)

The following code fragment uses an expression 'condition' that yields a fuzzy truth value $F0.4$ and a variable 'var1' that represents a fuzzy numerical value {21, F0.8}.

```
0: if /* condition */ then
1:   restext1 := "The value " || var1 || " has a degree of presence of "
2:               || (var1 is present);
3:
4:   var2 := var1 * 2;
5:
6:   restext2 := "The value " || var2 || " has a degree of presence of "
7:               || (var2 is present);
8: endif;
```

The conditional statement defines the reduced conditional context of 0.4 (line 0). The result of the first concatenation is a string that is stored in 'restext1' (lines 1,2). According to definition 12, it has a degree of presence of 0.4 as defined by the conditional context.

restext1 ⇒ {"The value 21 has a degree of presence of 0.8", 0.4}

The degree of presence of the variable 'var1' that is used as part of the expression is still 0.8, as it has been defined outside the conditional context and the degree of presence of existing data values is not affected by the conditional context.

Analogously, 'var2' represents a fuzzy number with a degree of presence of 0.4 that is the result of the multiplication of 'var1' with the constant 2 (line 4):

var2 ⇒ {42, 0.4}

The result of the last concatenation is represented by 'restext2' (lines 6,7):

restext2 ⇒ {"The value 42 has a degree of presence of 0.4", 0.4}

Another example shows the list handling within a conditional context. List are ordered sets of elements and can be heterogenous (although lists cannot be nested). They can contain both fuzzy and crisp data.

Example 14: (list creation within reduced conditional context)

```
0: if /* condition */ then
1:   reslist := "hl7", "arden", "syntax";
2:   var2 := reslist[2];
3: endif;
```

The expression 'condition' yields a fuzzy truth value $F0.3$. Variable 'reslist' represents a newly created list (line 1) with three crisp data values.

reslist ⇒ {({"hl7", 0.3}, {"arden", 0.3}, {"syntax", 0.3}), 1.0}

By using the 'element' operator, the second element of the list is stored in 'var2' (line 2). The result of the operation has a degree of presence of 0.3.

var2 ⇒ {"arden", 0.3}

2.2.2.2 The 'null' value

Arden Syntax defines 'null' as representation of uncertainty. However, 'null' does not define uncertainty in the sense of vagueness but in the sense of incompleteness. Within conditional expressions or logical operations, 'null' has still to be handled separately.

Fuzzy data with a degree of presence of 0.0 is considered to be handled as 'null', as such a degree of presence indicates that the conditional context during the creation of the data value has been not fulfilled at all (thus such a data value should not exist).

2.2.3 Default 'degree of presence' handling

As seen in the last example, the extensions of the Arden Syntax affect not only logical expressions but also other operators, as their arguments might consist of fuzzy data. Therefore, the functionality of every single operator must be extended to handle such fuzzy data or, more precisely, to handle the degree of presence of data values.

Definition 13 (Default degree of presence handling):

The result of an unary operator has the same degree of presence as the degree of presence of the operand.

For other operators the concept of the 'min' operator of fuzzy logical 'and' operations is applied to the degrees of presence. The result of a binary or ternary operator has a degree of presence that is at most as high as the smallest one of any argument.

These rules apply for all operators that do not redefine the handling of fuzzy data separately.

The following example illustrates the default handling of the degree of presence by two arithmetic operators and one comparison operator.

Example 15: (Default handling of the degree of presence)

The addition of number constant to a fuzzy data numerical value uses the default handling for the degree of presence of the arguments.

3 + {4, 0.8} ⇒ {7, 0.8}

The negation of one fuzzy data numerical value keeps the degree of presence of the argument.

−{5, 0.9} ⇒ {−5, 0.9}

A fuzzy comparison of fuzzy data with a constant returns the same result as the comparison of a crisp one would do, but the degree of presence of the truth value is handled as defined by the default handling.

{5 days, 0.4} is less than or equal 1 week fuzzified by 2 days ⇒ {$F0.5$, 0.4}

2.3 Definition of the extensions

This section defines the single extensions of Fuzzy Arden that are based on the conceptual models as explained in the last section. These extensions can be viewed as a first step towards extending the Arden Syntax by concepts of fuzzy set theory and fuzzy logic. They can be used to model vagueness with a compromise of readability and backward compatibility.

2.3.1 Fuzzy data model

The **truth data type** is extended and termed 'fuzzy truth' as defined on page 43. Fuzzy truth constants are notated by the keyword 'fuzzy' followed by a number and can represent any degree of truth from $F1.0$ ('true') to $F0.0$ ('false'). 'True' and 'false' are still usable as keywords.

Table 2.2: Fuzzy and crisp Arden Syntax data values

type/classification	**sample values**
fuzzy truth	true, $F0.9$, fuzzy 0.50, $F0.3$, false
crisp	"hl7"
	42
	2002-05-01T12:00:00
	{"hl7", 1.0}
fuzzy	{"hl7", 0.5}
	{42,0.5}
	{2002-05-01T12:00:00, 0.78}
	{"hl7", 0.1}
null	{"hl7", 0.0}
	null

Arden Syntax data has two attributes, viz. the primary time attribute and the degree of presence attribute as defined on page 47. If the degree of presence is below 1.00 the data value is classified as fuzzy data; if the degree of presence is 1.00 it is defined as crisp data. A degree of presence of 0.00 is equivalent to 'null'. Table 2.2 shows some sample data values.

2.3.2 Fuzzy operators

Fuzzy operators are comparison operators whose thresholds are fuzzified. In summary, all 'simple comparison' operators and subsets of the 'is comparison' and 'occur' operators were redefined as fuzzy operators. Comparisons between strings and the 'matches pattern' operator, which compares string values to given patters, could basically be fuzzy operators too. However, in this stage of the extension they are handled as crisp comparisons, as linguistic comparisons can be very complex.

The fuzzy operators are defined according the following notations[38]. Data type constraints are defined by:

<num:type> := <num:type> **op** <num:type> **op** <num:type>

If available, the linguistic operator names are used to define the fuzzy operators (**'op'**). The Arden Syntax specification defines the following keywords that can be used as type definition:

Boolean	Boolean data type
number	number data type
time	time data type
duration	duration data type
string	string data type
item	used only in 'call' statements
any-type	null, Boolean, number, time, duration, or string
non-null	Boolean, number, time, duration, or string
ordered	number, time, duration, or string

For the extended operators, two new keywords 'numerical' and 'fuzzy' are used additionally to the predefined type definitions:

numerical	fuzzy truth, number, time, or duration
fuzzy	fuzzy truth

A comparison operator defines one value or a range of values where it yields 'true'. By fuzzification, this range of valid arguments is either extended or reduced. It is *extended* for those operators where the thresholds originally were included in the range of valid arguments, such as the 'greater than or equal' operator (a value that is slightly smaller than the threshold may still be classified as "equal" somehow). It is *reduced* for those operators that did not include the thresholds in the range of valid values, such as the 'greater than' operator (a value that is slightly greater than the threshold may be classified as not sufficiently greater).

[38] Parts of the (re-)definitions of operators are cited from the specification without being marked as citations. This affects only some passages that are not influenced by the fuzzy extensions.

Each fuzzy comparison operator definition illustrates the reduction or extension of this range by a graph that sketches the compatibility function. The differences of the range that is extended or reduced is indicated by a hatched area, where a hatched area above the compatibility function indicates a reduced range and a hatched area below the function indicates an extended range of valid values.

2.3.2.1 is equal

The = operator has two synonyms: 'eq' and 'is equal' and checks for equality returning a fuzzy truth value. If the arguments are of different types, 'false' is returned. If an argument is 'null', then 'null' is always returned. When used with a numerical argument, an optional 'fuzzified by' argument can be defined that modifies the compatibility function as shown in figure 2.5.

<n:fuzzy> := <n:numerical> **is equal** <n:numerical> [**fuzzified by** <n:numerical>]

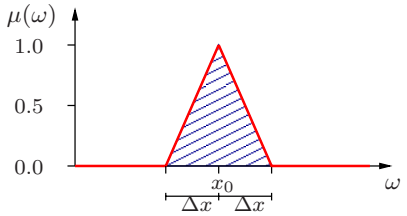

Figure 2.5: Fuzzy operator: compatibility function 'is equal'

Its use is:
```
F0.0  := 110 is equal 100 fuzzified by 10;
F0.5  := 110 is equal 100 fuzzified by 20;
F0.66 := 110 is equal 100 fuzzified by 30;
false := 42 is equal "42" fuzzified by 1;
null  := 3 days is equal 72 hours fuzzified by null;
```

2.3.2.2 is not equal

The <> operator has two synonyms: 'ne' and 'is not equal'. It checks for inequality, returning a fuzzy truth value. If the arguments are of different types, 'true' is returned. If an argument is 'null', then 'null' is returned. When used with a numerical argument, an optional 'fuzzified by' argument can be defined that modifies the compatibility function as shown in figure 2.6.

<n:fuzzy> := <n:numerical> **is not equal** <n:numerical> [**fuzzified by** <n:numerical>]

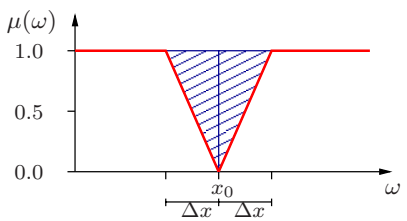

Figure 2.6: Fuzzy operator: compatibility function 'is not equal'

Its use is:
```
F1.0  := 110 is not equal 100 fuzzified by 10;
F0.5  := 110 is not equal 100 fuzzified by 20;
F0.33 := 110 is not equal 100 fuzzified by 30;
true  := 42 is not equal "42" fuzzified by 1;
null  := 3 days is not equal 72 hours fuzzified by null;
```

2.3.2.3 is less than

The < operator has three synonyms: 'lt', 'is less than', and 'is not greater than or equal'. It is used on ordered types; if the types do not match, 'null' is returned. When used with a numerical argument, an optional 'fuzzified by' argument can be defined that modifies the compatibility function as shown in figure 2.7.

<n:fuzzy> := <n:numerical> **is less than** <n:numerical> [**fuzzified by** <n:numerical>]

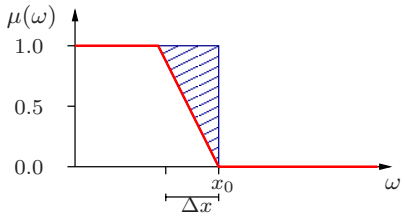

Figure 2.7: Fuzzy operator: compatibility function 'is less than'

Its use is:
```
true  := 30-06-2002 is less than 01-07-2002;
F0.5  := 30-06-2002 is less than 01-07-2002 fuzzified by 2 days;
F0.02 := 99 is less than 100 fuzzified by 50;
null  := 3 days is less than "72 hours";
```

2.3.2.4 is less than or equal

The <= operator has three synonyms: 'le', 'is less than or equal', and 'is not greater than'. It is used on ordered types; if the types do not match, 'null' is returned. When used with a numerical argument, an optional 'fuzzified by' argument can be defined that modifies the compatibility function as shown in figure 2.8.

<n:fuzzy> := <n:numerical> **is less than or equal** <n:numerical> [**fuzzified by** <n:numerical>]

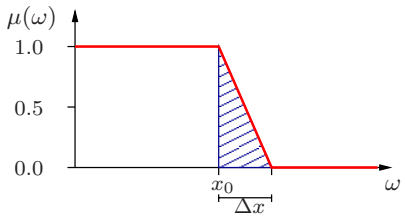

Figure 2.8: Fuzzy operator: compatibility function 'is less than or equal'

Its use is: false := `05-07-2002 is less than or equal 01-07-2002;`
$F0.71$:= `05-07-2002 is less or equal than 01-07-2002 fuzzified by 1 week;`
$F0.4$:= `38.2 is less than or equal 38.0 fuzzified by 0.5;`
null := `3 days is less than or equal "72 hours";`

2.3.2.5 is greater than

The > operator has three synonyms: 'gt', 'is greater than', and 'is not less than or equal'. It is used on ordered types; if the types do not match, 'null' is returned. When used with a numerical argument, an optional 'fuzzified by' argument can be defined that modifies the compatibility function as shown in figure 2.9.

<n:fuzzy> := <n:numerical> **is greater than** <n:numerical> [**fuzzified by** <n:numerical>]

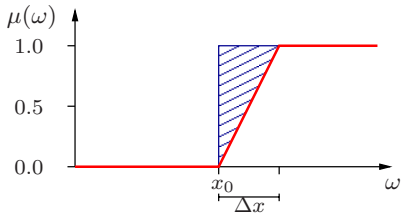

Figure 2.9: Fuzzy operator: compatibility function 'is greater than'

Its use is: true := `02-07-2002 is greater than 01-07-2002;`
$F0.5$:= `02-07-2002 is greater than 01-07-2002 fuzzified by 2 days;`
$F0.02$:= `101 is greater than 100 fuzzified by 50;`
null := `3 days is greater than "72 hours";`

2.3.2.6 is greater than or equal

The >= operator has three synonyms: 'ge', 'is greater than or equal', and 'is not less than'. It is used on ordered types; if the types do not match, 'null' is returned. When used with a numerical argument, an optional 'fuzzified by' argument can be defined that modifies the compatibility function as shown in figure 2.10.

<n:fuzzy> := <n:numerical> **is greater than or equal** <n:numerical> [**fuzzified by** <n:numerical>]

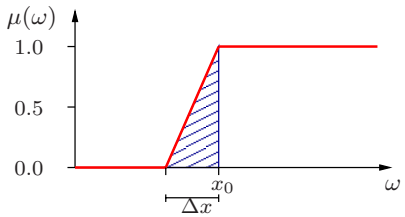

Figure 2.10: Fuzzy operator: compatibility function 'is greater than or equal'

Its use is: false := 30-06-2002 is greater than or equal 01-07-2002;
$F0.14$:= 30-06-2002 is greater or equal than 01-07-2002 fuzzified by 1 week;
$F0.5$:= 1 day is greater than or equal 25 hours fuzzified by 2 hours;
null := 3 days is greater than or equal "72 hours";

2.3.2.7 is within to

The 'is within ... to' operator checks whether the first argument is within the range specified by the second and third arguments; the range is inclusive. It is used on ordered types; if the types do not match, 'null' is returned. Optional 'fuzzified by' arguments can be defined that modify the compatibility function as shown in figure 2.11.

<n:fuzzy> := <n:numerical> **is within** <n:numerical> [**fuzzified by** <n:numerical>] **to** <n:numerical> [**fuzzified by** <n:numerical>]

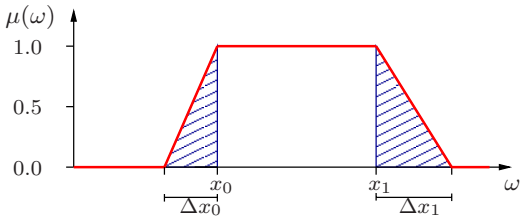

Figure 2.11: Fuzzy operator: compatibility function 'is within to'

Its use is: true := 110 is within 100 fuzzified by 20 to 120 fuzzified by 20;
$F1.0$:= 100 is within 100 fuzzified by 20 to 120 fuzzified by 20;
$F0.25$:= 85 is within 100 fuzzified by 20 to 120 fuzzified by 20;
false := 150 is within 100 fuzzified by 20 to 120 fuzzified by 20;

2.3.2.8 is within preceding

The 'is within ... preceding' operator checks whether the left argument is within the inclusive time period defined by the second two arguments (from the third argument minus the second to the third). Optional 'fuzzified by' arguments can be defined that modify the compatibility function as shown in figure 2.12.

<n:fuzzy> := <n:time> **is within** <n:duration> [**fuzzified by** <n:duration>] **preceding** <n:time> [**fuzzified by** <n:duration>]

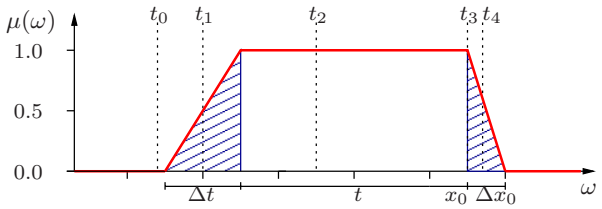

Figure 2.12: Fuzzy operator: compatibility function 'is within preceding'

To keep the examples short for this and some of the following operators, some time stamps are sketched exemplarily in the corresponding compatibility function graphs. Assuming that 't0' represents 2002-09-08T09:36:00, 't1' represents 2002-09-09T00:00:00, 't2' represents 2002-09-10T12:00:00, 't3' represents 2002-09-12T12:00:00, and finally 't4' represents 2002-09-12T16:48:00, the use of the 'is within preceding' operator is:

```
false := t0 is within 3 days fuzzfied by 1 day preceding t3 fuzzified by 12 hours;
F0.5  := t1 is within 3 days fuzzfied by 1 day preceding t3 fuzzified by 12 hours;
true  := t2 is within 3 days fuzzfied by 1 day preceding t3 fuzzified by 12 hours;
F0.6  := t4 is within 3 days fuzzfied by 1 day preceding t3 fuzzified by 12 hours;
```

2.3.2.9 is within following

The 'is within ... following' operator checks whether the left argument is within the inclusive time period defined by the second two arguments (from the third argument to the third plus the second). Optional 'fuzzified by' arguments can be defined that modify the compatibility function as shown in figure 2.13.

<n:fuzzy> := <n:time> **is within** <n:duration> [**fuzzified by** <n:duration>] **following** <n:time> [**fuzzified by** <n:duration>]

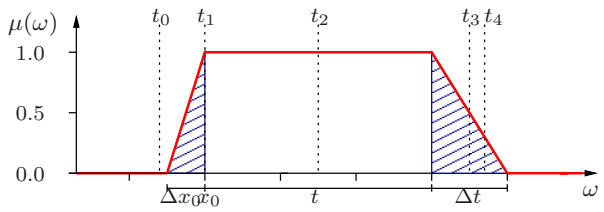

Figure 2.13: Fuzzy operator: compatibility function 'is within following'

Its use is (assuming that the variables represent the same time stamps as in the last example):

```
false := t0 is within 3 days fuzzfied by 1 day following t1 fuzzified by 12 hours;
true  := t2 is within 3 days fuzzfied by 1 day following t1 fuzzified by 12 hours;
F0.5  := t3 is within 3 days fuzzfied by 1 day following t1 fuzzified by 12 hours;
F0.3  := t4 is within 3 days fuzzfied by 1 day following t1 fuzzified by 12 hours;
```

2.3.2.10 is within surrounding

The 'is within ... surrounding' operator checks whether the left argument is within the inclusive time period defined by the second two arguments (from the third argument minus the second to the third plus the second). Optional 'fuzzified by' arguments can be defined that modify the compatibility function as shown in figure 2.14.

<n:fuzzy> := <n:time> **is within** <n:duration> [**fuzzified by** <n:duration>] **surrounding** <n:time>

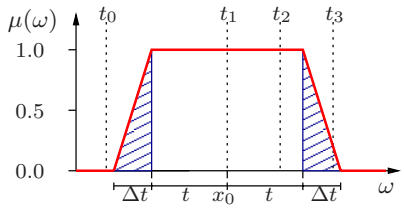

Figure 2.14: Fuzzy operator: compatibility function 'is within surrounding'

Its use is (assuming that 't0' represents 2002-09-06T21:36:00, 't1' represents 2002-09-08T12:00:00, 't2' represents 2002-09-09T04:48:00, and 't3' represents 2002-09-09T21:36:00):

```
false  := t0 is within 1 day fuzzified by 12 hours surrounding t1;
true   := t2 is within 1 day fuzzified by 12 hours surrounding t1;
F0.2   := t3 is within 1 day fuzzified by 12 hours surrounding t1;
```

2.3.2.11 is within past

The 'is within past' operator checks whether the left argument is within the time period defined by the right argument ('now' minus the right argument to 'now', where 'now' represent the time when the execution of the MLM started). The time period can be extended by an optional 'fuzzified by' argument that modifies the compatibility function as shown in figure 2.15.

<n:fuzzy> := <n:time> **is within past** <n:duration> [**fuzzified by** <n:duration>]

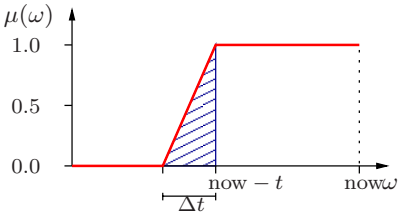

Figure 2.15: Fuzzy operator: compatibility function 'is within past'

Its use is (assuming that 't0' represents 2002-09-06T21:36:00, 't1' represents 2002-09-08T12:00:00, 't2' represents 2002-09-09T04:48:00, and 't3' represents 2002-09-09T21:36:00):

Its use is:
```
false  := (now - 5 hours) is within past 2 hours fuzzified by 2 hours;
F0.5   := (now - 3 hours) is within past 2 hours fuzzified by 2 hours;
true   := (now - 2 hour) is within past 2 hours fuzzified by 2 hours;
```

2.3.2.12 is within same day as

The 'is within same day as' operator checks whether the left argument is on the same day as the second argument. The time period can be extended by an optional 'fuzzified by' argument that modifies the compatibility function as shown in figure 2.16.

<n:fuzzy> := <n:time> **is within same day as** <n:date> [**fuzzified by** <n:duration>]

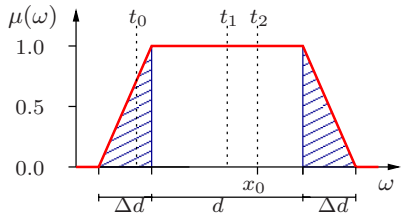

Figure 2.16: Fuzzy operator: compatibility function 'is within same day as'

Its use is (assuming that 't0' represents 2002-09-08T21:36:00, 't1' represents 2002-09-09T12:00:00, and 't2' represents 2002-09-09T16:48:00):

Its use is: $F0.6$ `:= t0 is within same day as t2 fuzzified by 12 hours;`
true `:= t1 is within same day as t2 fuzzified by 12 hours;`

2.3.2.13 is before

The 'is before' operator checks whether the left argument is before the second argument. The time period can be reduced by an optional 'fuzzified by' argument that modifies the compatibility function as shown in figure 2.17.

<n:fuzzy> := <n:time> **is before** <n:time> [**fuzzified by** <n:duration>]

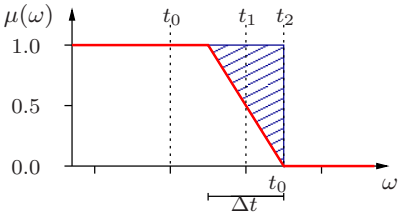

Figure 2.17: Fuzzy operator: compatibility function 'is before'

Its use is (assuming that 't0' represents 2002-09-08T12:00:00, 't1' represents 2002-09-09T00:00:00, and 't2' represents 2002-09-09T12:00:00):

true `:= t0 is before t2 fuzzified by 1 day;`
$F0.5$ `:= t1 is before t2 fuzzified by 1 day;`

2.3.2.14 is after

The 'is after' operator checks whether the left argument is after the second argument. The time period can be reduced by an optional 'fuzzified by' argument that modifies the compatibility function as defined in figure 2.18.

<n:fuzzy> := <n:time> **is before** <n:time> [**fuzzified by** <n:duration>]

Its use is (assuming that 't0' represents 2002-09-08T12:00:00, 't1' represents 2002-09-09T00:00:00, and 't2' represents 2002-09-10T00:00:00):

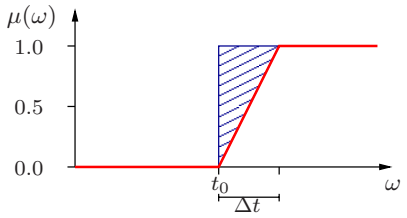

Figure 2.18: Fuzzy operator: compatibility function 'is after'

$F0.5$ `:= t1 is after t0 fuzzified by 1 day;`
false `:= t2 is after t0 fuzzified by 1 day;`

2.3.3 Crisp operators

This section defines the functionality of crisp operators for processing fuzzy data. Working with fuzzy data requires correct handling of the degree of presence that has, analogous to the primary time attribute, to be computed for the result of an operation and might additionally influence the result in its value.

As defined by definition 13 on page 50, a unary crisp operator keeps the degree of presence of its only argument. If a crisp operator has more than one argument, the presence attribute of the result equals the minimum of the individual presence attributes of its arguments.

Next, only those operators which are influenced by the degree of presence of their arguments in other ways than defined by the default handling are defined separately. The order of the operator categories in this section follows their order in the specification.

2.3.3.1 List operators

Basically the list operators follow the default handling of fuzzy data. However, as lists are just collections of data and cannot have a reduced degree of presence, the lists operators are described briefly.

List concatenation

The list concatenation operator ',' creates a list from one data value or by appending two or more data values which can also be lists. The result is always one list containing the data in their order as arguments. If one argument is a list, its elements are copied instead of storing the list as one element of the new one, as a list cannot be nested. The degree of presence of the resulting list is always 1.0.

Merge

The 'merge' creates a list by appending two lists and additionally sorts the result by the primary time of the list elements.

Sort

The sort operators 'sort data' and 'sort time' reorder a list either by the value of the elements or by their primary time. The individual degree of presence of the elements is retained. Additionally, a new sort operator 'sort presence' that sorts the elements of a list in ascending order by their degree of presence is defined.

The 'sort' operator reorders a list based on element keys, which are either the element values (keyword data), the primary times (keyword 'time'), or the degrees of presence (keyword 'presence'). An optional modifier may be used with the sort operator. If used, the corresponding modifier must be placed immediately after the 'sort' keyword. The following keywords can be placed after the sort keyword: 'data', 'time', or 'presence', which are mutually exclusive. If no modifier is used, the 'sort' operator defaults to a 'data sort'. The data are always sorted in ascending order. For sorting in descending order, 'reverse' can be used.

<n:any-type> := **sort [data** | **time** | **presence]** <n:any-type>

The sort options are considered to be part of the 'sort' operator for precedence purposes. This resolves the potential conflict with the 'time [of]' and 'is present' operators. Thus the expression 'sort time x' should be parsed as "sort the list x by time" rather than as "extract the primary times from the list x and sort the list of times". Analogously, the expression 'sort presence x' should be parsed as "sort the list x by the degree of presence".

Its use is (assuming that 'data1' has a data value of {30, 0.1}, {10, 1.0}, {20, 0.5}:

```
{10, 1.0}, {20, 0.5}, {30, 0.1} := sort data data1;
{20, 0.5}, {10, 1.0}, {30, 0.1} := sort presence data1;
```

Element

The Arden Syntax specification defines one particular list operator, the 'element' operator, in the 'aggregation operators' section. In this work, it is redefined in the current section instead of following the order of the specification document, to complete the set of list-specific operators.

The binary 'element' operator can be used to extract one or more elements of a list. The result is either a single value or a list; the degree of presence of the value or of the list elements is retained.

2.3.3.2 Where operator

The 'where' operator plays an important role when using Fuzzy Arden, as it can be used to create fuzzy data by selecting crisp data based on fuzzy truth values.

Where (binary, non-associative)
The 'where' operator performs the equivalent of a relational 'select ... where ...' on its left argument. In general, the left argument is a list, often the result of a query to the database. The right argument is usually of the fuzzy truth type (although this is not required), and must be the same length as the left argument.

<n:any-type> := <m:any-type> **where** <m:any-type>

If called with crisp truth values, the result is a list that contains only those elements of the left argument where the corresponding element in the right argument is 'true'. If the right argument is anything else, including 'false', 'null', or any other type, then the element in the left argument is dropped.

Example 16: (Where operator on crisp truth values)

```
(10, 30) := (10, 20, 30, 40) where (true, false, true, 3)
```

If the right argument contains fuzzy truth values, it drops only those elements of the left argument where the corresponding element of the left element is 'false', 'null', or any other data type. The resulting list contains all those elements whose corresponding truth value is greater than $F0.0$. If the truth value is higher than 'false' but not 'true' the resulting element is a fuzzy data type with its degree of presence equal to the fuzzy truth value.

Example 17: (Where operator on fuzzy truth values)

```
(10, {30, 0.5}) := (10, 20, 30, 40) where (true, false, F0.5, 3)
```

If the left argument contains fuzzy data, the degree of presence of the elements in the result list is set to the minimum of the corresponding degrees of presence:

Example 18: (Where operator on crisp truth values: further examples)

```
({"a", 1.0}, {2, 0.3}) := ("a", [2, 0.3], "c") where (true, F0.5, false)
({"a",1.0}, {"a",0.5}) := "a" where (true, F0.5, false)
("a", {2, 0.3}, "c") := ("a", {2, 0.3}, "c") where true
({"a", 0.2}, {2, 0.2}, {"c", 0.2}) := ("a", {2, 0.3}, "c") where F0.2
```

2.3.3.3 Logical operators

The 'and' operator performs the logical conjunction and the 'or' operator the logical disjunction of their two arguments. The 'not' operator performs the logical negation of its argument.

<n:Boolean> := <n:any-type> **and** <n:any-type>

<n:Boolean> := <n:any-type> **or** <n:any-type>

<n:Boolean> := **not** <n:any-type>

The three-valued logic is extended by fuzzy logical operators. The truth tables of the three operators 'and', 'or', and 'not' are shown in table 2.3. The degree of presence of the fuzzy truth value is handled by the default handling.

The 'fuzzy and' and 'fuzzy or' operators are mathematically defined by the 'min' and 'max' operators:

$$(x_1 \text{ and } x_2) := \min(x_1, x_2) \tag{2.1}$$

$$(x_1 \text{ or } x_2) := \max(x_1, x_2) \tag{2.2}$$

The option of choosing alternative operator sets could be included in the future development of the syntax.

Table 2.3: Fuzzy logical operators truth-tables

x_1 **and** x_2	true	$F0.x$	false	other
true	true	x_2	false	null
$F0.x$	x_1	fuzzy and	false	null
false	false	false	false	false
other	null	null	false	null

x_1 **or** x_2	true	$F0.x$	false	other
true	true	true	true	true
$F0.x$	true	fuzzy or	x_1	x_1
false	true	x_2	false	null
other	true	x_2	null	null

not x_1	true	$F0.x$	false	other
	false	$1 - x_1$	true	null

2.3.3.4 Is operators

The following operators are defined in the 'is operators' class in addition to the fuzzily defined ones in sections 2.3.2.7 to 2.3.2.14.

Is present: The 'is present' operator has one synonym: 'is not null'. (Similarly, 'is not present' has one synonym: 'is null'.) If the argument is not 'null' then the operator returns its degree of presence as fuzzy truth value; otherwise it returns 'false'. The degree of presence of the result is 1.0.

<n:fuzzy> := <n:any-type> **is present**

Is in: The 'is in' operator checks for membership of the left argument in the right argument, which is usually a list.

<n:fuzzy> := <n:any-type> **is in** <m:any-type>

The criterion whether a data value is an element of the list is restricted to the value and does not include the need for equality of the attributes 'primary time' and 'degree of presence'. If an element is found to be contained in both arguments, the result is a fuzzy truth value that equals the lower degree of presence of both elements. The degree of presence of the single results is 1.0.

Example 19: (Crisp is-operators)

```
(F0.8, true, false, false)  :=  ({1, 0.8}, {2, 1.0}, {3, 0.0}, null) is present;
    (F0.7, F0.8, F0.3)      :=  (1, {2, 0.8}, {3, 0.4}) is in ({1, 0.7}, 2, {3, 0.3});
     (true, false, false)   :=  (1, 2, {3, 0.8}) is in (1);
           (true, true)     :=  (null, {1, 0.0}) is in ("a", "b", "c");
```

2.3.3.5 Aggregation operators

Most aggregation operators handle the degree of presence of their arguments in the default way. The following operators handle the degree of presence of their arguments alternatively. The degree of presence of their result is 1.0.

exist: The 'exist' operator returns 'true' if the argument contains at least one crisp non-'null' item in the list. Otherwise it returns a fuzzy truth value that equals the highest degree of presence attribute of any item of the argument.

<1:truth> := **exists** <n:any-type>

average: The 'average' operator calculates the average of a numerical list.

<1:numerical> := **average** <n:numerical>

If the list contains fuzzy data, a weighted average is computed by applying:

$$\text{avg}([x_1 \dots x_n]) := \frac{\sum_n x_n \mu(x_n)}{\sum_n \mu(x_n)} \tag{2.3}$$

median: The 'median' operator calculates the median value of a numerical list. The list is first sorted. If there is an odd number of items, it selects the middle value and its degree of presence is kept. If the size of the list is even, the average of the two middle items is computed, as defined by the 'average' operator.

<1:numerical> := **median** <n:numerical>

any: Originally, the 'any' operator returns 'true' if at least one of the items in a list is 'true'. If used with fuzzy truth values, it returns the highest fuzzy truth value (comparable to a fuzzy logical 'or' operation by applying the 'max' operator). It returns 'null', if any non-fuzzy truth value[39] is an element of the list.

<1:truth> := **any** <n:any-type>

all: Originally, the 'all' operator returns 'true' if all of the items in a list are 'true'. If used with fuzzy truth values, it returns the lowest fuzzy truth value (comparable to a fuzzy logical 'and' operation by applying the 'min' operator). It returns 'null' if any non-fuzzy truth value is an element of the list.

<1:truth> := **all** <n:any-type>

no: The result of the 'no' operator is equal to the operation 'not any'.

<1:truth> := **no** <n:any-type>

2.3.4 Statements for program flow control

Conditional statements that control the execution of an algorithm ('program flow') are partly responsible for the creation or modification of data in an uncertain environment termed 'conditional context' (this concept has been defined earlier in section 2.2.2.). This section defines the 'if-then', 'while', and 'conclude' statements that can create new conditional contexts and might also be influenced by their current conditional context.

[39] Any string, number, date, or duration, regardless whether fuzzy or crisp data.

2.3.4.1 If-then statement

An Arden Syntax 'if-then' statement controls whether a block of statements (referenced as *code block* in the following) has to be executed or not. It is only executed if the condition is met; otherwise an optional 'else' block is executed. If the condition has no crisp result, both blocks are executed in parallel; the degree of truth of the condition defines the degree of applicability of the first code block, the complement defines the degree of applicability of the else code block.

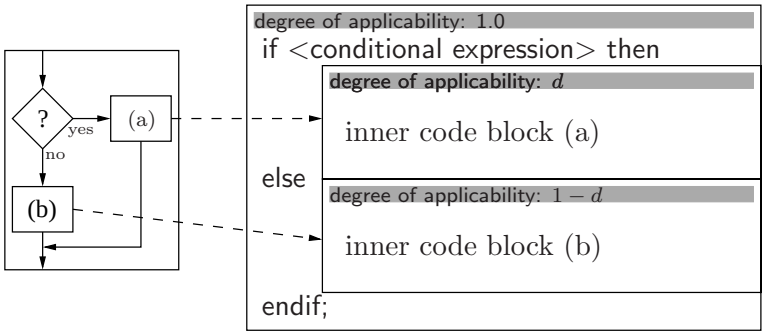

Figure 2.19: Scheme for a fuzzy 'if-then' statement

Figure 2.19 shows a scheme for a fuzzy 'if-then' statement; the gray boxes stand for the degree of applicability of the individual code block and therefore for the corresponding conditional context of all included statements and operators. The place holder d represents the degree of truth of the conditional expression. The degree of applicability of the code block that encloses the entire statement is 1.0 as long its execution does not depend on any conditions.

Example 20: (Calculation of drug dose)

Assuming that the dose of a drug is dependent on the patient's age, the correct dose is calculated by two equations: one is suitable for children, the other for adults. As the equations return significantly different results, both equations are embedded in a fuzzified 'if-then-else' construct that uses a fuzzily defined condition.

The fuzzy sets "child" and "adult" were defined in example 2 on page 22. If the variable `birthdate` would represent the patient's date of birth, one possible implementation in Arden Syntax would be:

```
0: age := (now - birthdate) / 1 year;
1: if age is less than or equal 17 fuzzified by 2 then
2:   dose := call equation1; /* equation 1 */;
3: else
4:   dose := call equation2; /* equation 2 */;
5: endif;
```

As long as the patient is younger than 17 years or older than 19 years, the statement will behave crisply and return either the result of equation 1 or equation 2. If his age is in the borderline range, both code blocks will be executed and the variable 'dose' represents in two different conditional contexts two different fuzzy values (lines 2,4).

Assuming that the patient's age is 17.5 years, the result of the conditional expression is $F0.75$; thus the degree of applicability for the first equation is 0.75 and for the second it is 0.25.

Let us assume that r_1 is the numerical result of equation 1 and $r_2 \neq r_1$ the result of equation 2. The value represented by the variable 'dose' in the first code block would be the fuzzy value $\{r_1, 0.75\}$. Analogously, the value of the same variable in the second code block would be $\{r_2, 0.25\}$.

As shown in this example, an 'if-then' statement[40] that uses a fuzzily defined condition can have an ambiguous result; in this case, the same variable represents two different values, each by a different degree of presence. To get an overall result, both values have to be processed and the "fuzziness" of their parallel existence has to be "defuzzified".

Following the compositional rule of inference, each sub-result should gain as much influence on the overall result as the degree of applicability of its conditional environment implies. This works well for numerical values, but other combinations of data or of conditional contexts require different methods to "defuzzify" the results and transfer them back to the higher conditional context that embedded the current one.

Definition 14 ("Defuzzification" of data in parallel conditional contexts):

Case 1: *One numerical variable (number, date, time, fuzzy truth) represents in two parallel, inverse conditional contexts two different values x_1^n and x_2^n of the same data type. Its value x^{n-1} in the superior conditional context is computed as the weighted average of the two sub-results:*

$$avg(x_1, x_2) := \frac{x_1\mu(x_1) + x_2\mu(x_2)}{\mu(x_1) + \mu(x_2)} \tag{2.4}$$

The degree of presence of the result is set to the superior conditional context (that might be 1.0 *in case that the superior context is the root context).*

$$\mu(x^{n-1}) := \text{conditional context}_{n-1} \tag{2.5}$$

Case 2: *One variable represents in two parallel, inverse conditional contexts two different values x_1^n and x_2^n of the same non-numerical type (string, list, . . .) or of two different data types. Its value x^{n-1} in the superior conditional context is defined by selecting the value with the highest degree of presence (if both values have the same degree of presence, the first is chosen).*

The degree of presence of the result is computed by

$$\mu(x^{n-1}) := \text{conditional context}_{n-1} \cdot \mu(x^n) \tag{2.6}$$

If the result is a list, the individual degree of presence of every list element is computed analogously.

Case 3: *One variable represents in one conditional context one value x^n of any type. Its value is kept, its degree of presence in the superior conditional context is computed analogous to case 2 by*

$$\mu(x^{n-1}) := \text{conditional context}_{n-1} \cdot \mu(x^n) \tag{2.7}$$

[40] The example uses an 'if-then' and an 'else' statement. As the latter cannot be used without the first one, both are further referenced as 'if-then' statements.

By these methods, two approaches of fuzzy algorithms are used for the execution of conditional statements and the evaluation of their results. The inherent parallelism of the compositional rule of inference is applied to the execution of two alternative code blocks in case 1. If both alternatives compute two results of one variable and the two results can be averaged, each result gains influence according to the concept of the compositional rule of inference as defined by the conditional expression. If the values cannot be averaged (case 2 and 3), the concept of 'execution with threshold' is applied by choosing the result with the higher degree of presence (compare section 1.3.4).

The degree of presence is computed in two different ways. In case 1, where both alternatives gain influence according to the conditional contexts, the result has the same degree of presence as defined by the conditional context that embeds the alternatives. In the other cases, where only *one* alternative gains influence and the other is lost, the degree of presence of the result must be reduced to point out the difference to case 1. This is done by multiplying the degree of presence of the result in the embedded conditional context by the surrounding conditional context. The next examples illustrate this concept.

Example 21: ("Defuzzification", case 1: Calculation of drug dose, cont.)

By applying the defuzzification method in example 20 the result of the computation would be

$$\begin{aligned} r_0 &= \frac{x_1\mu(x_1) + x_2\mu(x_2)}{\mu(x_1) + \mu(x_2)} \\ &= \{\ \tfrac{x_1\mu(x_1)+x_2\mu(x_2)}{\mu(x_1)+\mu(x_2)},\ 1.00\} \end{aligned}$$

A more generic example shows how to "defuzzify" data by in case 2 and case 3.

Example 22: ("Defuzzification", case 2 and 3)

The following code snippet is given:

```
/**
 * 3 blood pressure values: F1.0 := bp1 is present
 *                          F1.0 := bp2 is present
 *                          F0.7 := bp3 is present
 **/
if risk is greater than 80 fuzzified by 10 then
    msg := "Please contact your doctor immediately!";
    res_list := bp1, bp2, bp3;
else
    msg := "No action required.";
endif;
```

Three variables, 'bp1', 'bp2', and 'bp3', represent three blood pressure values. The last was selected on the basis of a not absolutely fulfilled condition and has therefore a reduced degree of presence. Table 2.4 shows the value of the message variable and the degrees of presence of both result variables, depending on the value of the variable 'risk'.

If the risk factor is not greater than 80, then the degree of applicability of the first code block is 0.0 and that of the second code block is 1. The final value of 'msg' is unambiguously the one defined in the second block, the variable 'res_list' is 'null'.

Table 2.4: Results of example 22

'risk'	'msg'		μ('res_list')
	value	μ('msg')	
80	`"No act..."`	1.0	null
85	`"Please..."`	0.5	0.5, 0.5, 0.35
90	`"Please..."`	1.0	1.0, 1.0, 0.7

If the risk factor is 85 then the condition is partly fulfilled by a degree of $F0.5$ and both blocks are executed. As the values of the variables cannot be averaged, the one with the higher degree of presence is chosen. As both have the same degree of presence within their code blocks, the values out of the 'true' code block are taken as the results.

So far, the influence of the conditional context on the degree of presence of operator results and the handling of variables has been described. A further important aspect is the influence of the conditional context on other conditional contexts. Such contexts, which can be defined for example by nested 'if-then' statements, depend not only on the underlying conditional expression, but also on the current conditional context.

Definition 15 (Degree of applicability in nested conditional contexts):

If a conditional statement is used within a reduced conditional context, the resulting degree of applicability of the depending code blocks is computed by multiplying the result of the conditional expression with the current conditional context. This affects all included 'if-then' statements and 'while' loops.

The sum of two parallel, inverse conditional contexts is always equal to the value of the conditional context that embeds them. The following example illustrates the concept of nested conditional contexts.

Example 23: (Nested 'if-then' statements)

This abstract example combines two 'if-then' statements to a nested 'if-then-else' construct as shown in figure 2.20. The code blocks are referenced as (1), (1.1), (1.2), et cetera. The degree of applicability of the code blocks is referenced as d_1 and d_2, the overall degree of applicability of the entire construct is set to $d_0 = 1.0$.

This example uses two variables, 'res1' and 'res2', that represent different values, depending on two conditional expressions.

1. The first conditional expression yields a fuzzy truth value $F0.8$. Therefore code block (1.1) has a degree of applicability 0.8 while code block (1.2) has a degree of applicability 0.2.
2. The second conditional expression yields a fuzzy truth value $F0.3$. The degree of applicability of the depending code blocks (1.1.1) and (1.1.2) is determined by multiplying the degree of truth 0.3 with the current conditional context 0.8. Code block (1.1.1) has a degree of applicability 0.24, code block (1.1.2) has a degree of applicability 0.56.

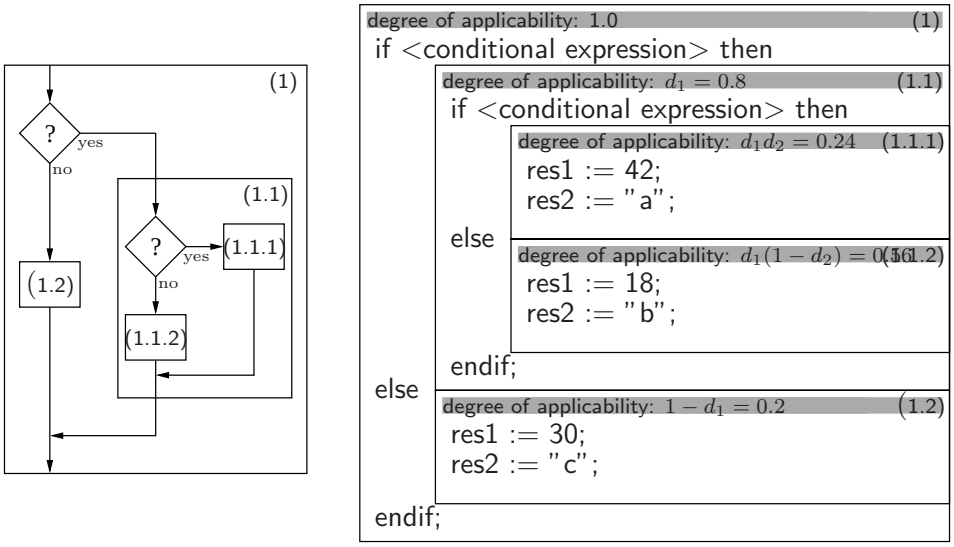

Figure 2.20: Scheme for a nested fuzzy if-then statement

3. During the execution of the algorithm, variable `res1` represents following fuzzy values:

$$\begin{array}{lll} (1.1.1) : \{42, 0.24\} & & \\ (1.1.2) : \{18, 0.56\} & \Rightarrow & (1.1) : \{25.2, 0.8\} \\ & & (1.2) : \{30, 0.2\} \quad \Rightarrow \quad (1) : \{26.16, 1.0\} \end{array}$$

4. Variable `res2` represents in the process of execution following fuzzy values:

$$\begin{array}{lll} (1.1.1) : \{\text{"a"}, 0.24\} & & \\ (1.1.2) : \{\text{"b"}, 0.56\} & \Rightarrow & (1.1) : \{\text{"b"}, 0.45\} \\ & & (1.2) : \{\text{"c"}, 0.2\} \quad \Rightarrow \quad (1) : \{\text{"b"}, 0.45\} \end{array}$$

2.3.4.2 'While' loop

A 'while' loop has to be handled in a similar way as nested 'if-then' statements. Figure 2.21 illustrates the relationship from 'while' statements to nested 'if-then' statements.

The 'while' loop executes a code block (marked with (1.1)) as many times as the condition is 'true'. If the condition is 'false', the algorithm continues with the statement that follows the 'while' loop.

If the condition is a fuzzy truth value which is neither 'true' nor 'false', both code blocks have to be executed. Up to this point the situation is comparable to the use of 'if-then' statements. The main difference is that the second code block is not *embedded* into the program flow, but defines the "usual way" the algorithm would have to continue after the execution of the 'while' loop terminated. Thus, in case of an uncertain truth value it might get executed *before* the 'while' loop has terminated.

In such a case, this parallel execution might be repeated in the next iteration of the loop. If the condition is neither 'true' nor 'false', both code blocks are executed in parallel again. Whereas this behavior is usually necessary for the inner code block of the loop, code block (1.2) would not get executed a second time if a crisp 'while' loop were used.

The iterations continue as long as the condition is not absolutely 'false'. If the inner code block (1.1) is executed n times, the rest of the algorithm is also executed n times.

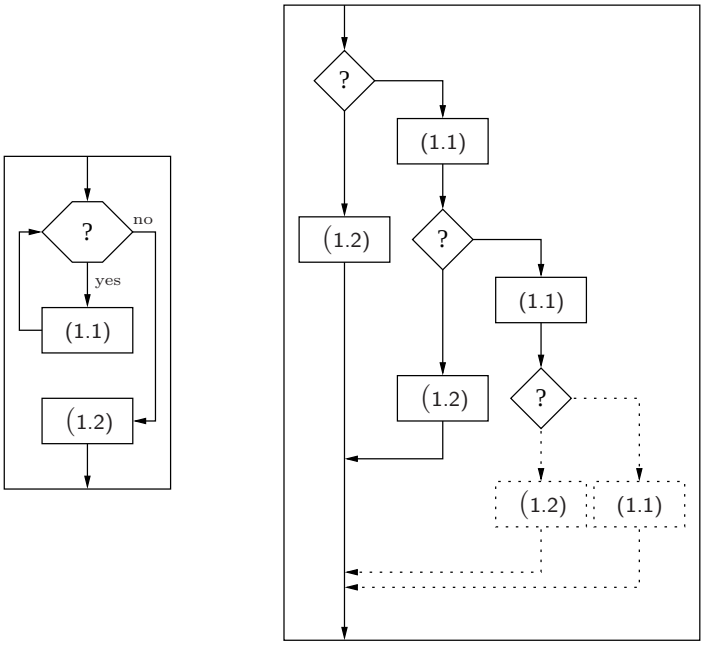

Figure 2.21: Schema of a fuzzy 'while' loop

Example 24: (Execution of a fuzzy 'while' loop)

A small pseudo-code algorithm is used to illustrate this point. Assuming that the 'write' command displays a text message on the screen:

```
i := 1;
while i = 0 fuzzified by 5 do
  write "1: My degree of applicability is " || (true is present);
  i := i+1;
enddo;
write "2: My degree of applicability is " || (true is present);
```

The output of this short algorithm would be:

```
1: My degree of applicability is 0.8
2: My degree of applicability is 0.2
1: My degree of applicability is 0.48
2: My degree of applicability is 0.32
1: My degree of applicability is 0.19
2: My degree of applicability is 0.29
1: My degree of applicability is 0.04
2: My degree of applicability is 0.15
```

Therefore, fuzzy conditional expressions as argument of 'while' loops should be used with caution.

2.3.4.3 Concluding procedure

The result of an MLM usually is a message that is generated in the action slot. This slot is executed when a 'conclude' statement whose condition yields 'true' is achieved.

Using fuzzy truth values, the decision logic of an MLM might conclude neither 'false' nor 'true'. Thus, 'true' or 'false' conclusions are only borderline cases of an extended range of conclusions.

Furthermore, it is possible to execute more than one conclude statement with different fuzzy truth values, for instance, within 'if-then' statements or loops. A single 'conclude' statement does only end the execution of the *current* program branch. If other code blocks are still running, their execution is not ended (this functionality is realized by the new 'terminate' statement that is defined later).

Definition 16 (Conclude statement):

The **conclude statement** *terminates the execution of the code block where the conclude statement is used. Its form is:*

conclude <expr>;

When a conclude statement is executed, the current sequence of statements is terminated.

The expression <expr> defines the degree of truth by which the actual code block makes a conclusion. It does not directly define whether the action slot has to be executed, as more than one conclude statements may be executed. The degree of presence of the truth value is equal to the conditional context.

If more than one conclude statement is reached during the execution of the logic slot, every single one is executed, contributing to one overall conclude value.

The overall execution of the logic slot terminates only if

- *all program branches are terminated by a conclude statement.*
- *the last statement of the logic slot is reached.*
- *the 'terminate' statement is used.*

If the conclude statement is used without an argument or the argument is not a fuzzy truth value, the default is 'conclude $F0.5$'. If no conclude statement is executed, the **default** *conclusion is conclude $F0.5$.*

As more than one conclude statement can be executed, an *overall concluding truth value* has to be computed before executing the action slot.

Definition 17 (Concluding (truth) value):

The **concluding truth value** *is a measure for the degree of truth of the decision made in the logic slot. It can be any fuzzy truth value from 'false' to 'true' and is available in the action slot by using the keyword* **concluding**.

The concluding value can be the result of one or more conclude statements in the action slot. The overall value is the average of the single conclude values weighted by their degree of presence. With C the set of all individual conclude values of an logic slot; the result is computed by:

$$c_{avg} := \frac{\sum_{c \in C} c\mu(c)}{\sum_{c \in C} \mu(c)}$$

Example 25: (Concluding truth value)

These examples illustrate the use of single or multiple conclude statements. The value of one single conclude statement that does not depend on any condition is retained:

conclude statement	cond. context	truth value	weighted addend
`conclude true;`	1.0	$F1.0$	1.0
		overall concluding value:	$F1.0$

Analogously, a single 'false' conclude statement is also retained:

conclude statement	cond. context	truth value	weighted addend
`conclude false;`	1.0	$F0.0$	0.0
		overall concluding value:	$F0.0$

Multiple conclude statements are used as shown in the next examples.

conclude statement	cond. context	truth value	weighted addend
`if` $F0.5$ `then`			
`conclude true;`	0.5	$F1.0$	0.5
`else`			
`conclude false;`	0.5	$F0.0$	0.0
`endif;`			
		overall concluding value:	$F0.5$

If one 'true' and one 'false' conclude statement is executed and both are weighted identically, the overall conclude value is neither 'true' nor 'false' ($F0.5$).

conclude statement	cond. context	truth value	weighted addend
`if` $F0.8$ `then`			
`if` $F0.3$ `then`			
`conclude true;`	0.24	$F1.0$	0.24
`else`			
`conclude;`	0.56	$F0.5$	0.28
`endif;`			
`else`			
`conclude false;`	0.2	$F0.0$	0.0
`endif;`			
		overall concluding value:	$F0.52$

The influence of the first condition on the "conclude true" is balanced by the second one, which is more 'false' than 'true'; therefore the 'conclude true' and the 'conclude false' are almost annulled. In the next example, the neutrally used conclude statement has been replaced by a 'conclude false' statement and a final 'conclude false' has been added.

conclude statement	**cond. context**	**truth value**	**weighted addend**
`if` $F0.8$ `then`			
`  if` $F0.3$ `then`			
`  conclude true;`	0.24	$F1.0$	0.24
`  else`			
`  conclude false;`	0.56	$F0.0$	0.0
`  endif;`			
`else`			
`  conclude false;`	0.2	$F0.0$	0.0
`endif;`			
`...`			
`conclude false;`	1.0	$F0.0$	0.0
		overall concluding value:	$F0.12$

As the program flow continues after the execution of the if-then statement, additional conclude statements may be executed as in the last example. In this case, the additional 'false' reduces the overall conclusion by half (without, an overall value of $F0.24$ would be achieved). If the first conclude statement is also replaced by a 'false' conclude statement, the rule concludes absolutely false:

conclude statement	**cond. context**	**truth value**	**weighted addend**
`if` $F0.8$ `then`			
`  if` $F0.3$ `then`			
`  conclude false;`	0.24	$F0.0$	0.0
`  else`			
`  conclude false;`	0.56	$F0.0$	0.0
`  endif;`			
`else`			
`  conclude false;`	0.2	$F0.0$	0.0
`endif;`			
		overall concluding value:	$F0.0$

Because of the redefinitions, the execution of the logic slot has no longer to be terminated immediately by a 'conclude' statement (if used within an 'if-then' statement, for instance). Therefore, a new statement 'terminate' is defined to instantly terminate the decision logic.

Definition 18 (Terminate statement):

The terminate statement ends the execution in the logic slot. Its form is:

terminate <expr>;

The expression (<expr>) in the terminate statement defines the overall concluding value that is not modified by any conclude statements, which may have been executed earlier. No further execution in the logic slot occurs, regardless of the expression. There may be more than one terminate statement in the logic slot, but only one will be executed in a single run of the MLM. If the expression does not return a fuzzy truth value, the default is 'terminate $F0.5$'. The action slot is executed independent of the terminate value.

Mainly, the newly defined 'terminate' statement equals in its functionality the old 'conclude' statement (except for executing the action slot even in the case of a 'false' conclusion).

Example 26: (Instant termination of the logic slot)

```
logic:
  ...
  /* check termination criterion */
  ...
  if blood_glucose_level is less than or equal 180 fuzzified by 30 then
    terminate false;
  endif;
  ...
  /* continue with MLM */
  ...

action:
  if concluding is false then
    write "The patient does not meet the requirements";
  else
    ...
  endif;
end:
```

One consequence of the extensions is the default execution of the action slot. Therefore the slot must additionally check the overall concluding value and decide whether the action slot should be executed or not. To achieve the usual behavior of the action slot, a simple expression such as

```
if concluding = true then ... endif;
```

that embraces all other statements of the slot can be implemented. However, as the ability to derive conclusions of different grades of truth is a sophisticated feature of the extensions, it may be included in the message as in the next example.

Example 27: (Integrating the concluding value into the message)

If needed the concluding value can be evaluated in the action slot, for example by including it in the message of the MLM:

```
write "The current ophthalmic data of patient " || patientId ||
      " is rated as suspicious by a degree of " || concluding || "."
      " We suggest immediate intervention.";
```

For instance, the message could be like this[41]:

> "The current ophthalmic data of patient 12345 is rated as suspicious by a degree of 0.88. We suggest immediate intervention."

The inclusion of the fuzzy truth value as number into the message has the advantage that the "defuzzification" is left to the physician who might have a wider overview of the current situation. Another way to include the degree of truth of the conclusion would be to select one message from a set of different messages:

```
msg := "The current ophthalmic data of patient " || patient_id ||
      " is rated ";
if concluding is less than F0.5 then
  msg := msg || " slightly suspicious. We suggest..."
elseif concluding is within F0.5 to F0.8 then
  msg := msg || " suspicious. We suggest..."
elseif concluding is greater than F0.8 then
  msg := msg || " very suspicious. We suggest..."
endif;
```

In this case one message could look like this:

> "The current ophthalmic data of patient 12345 is rated very suspicious. We suggest immediate intervention."

The extensions defined in this section can be used to model vagueness by conditional expressions and to process the resulting fuzzy truth values and data with a reduced degree of presence by all Arden Syntax operators and statements.

2.4 Linguistic variables

As the second step of extending the syntax this section defines, in addition to the known data-driven MLMs, a further new type of linguistic variable MLMs. The concept of linguistic variables has been introduced in section 1.3.2. As linguistic variable MLMs define, in contrast to data driven MLMs, a data structure rather than an algorithm, such MLMs

[41] It should be noted that the Arden Syntax does not specify how patient ids are referenced within an MLM. The rules engine used for this work provides a constant labeled 'patientId' that represents the id of the patient who is associated with the current event. In every case in which an event was defined entirely independent of patients, it is either 'null' or may represent another important id. The extension of the Arden Syntax by such an identifier is highly recommended and could be part of the curly braces fix.

do not need a logic or action slot. Instead, it is necessary to define all elements that specify a linguistic variable as shown in figure 2.22. To simplify the use of linguistic variables in Arden Syntax, it is required that all terms in T (the set of valid terms) be generated by the author; thus the set of syntactical rules G, which would create T, is not needed separately. Instead, T is explicitly defined.

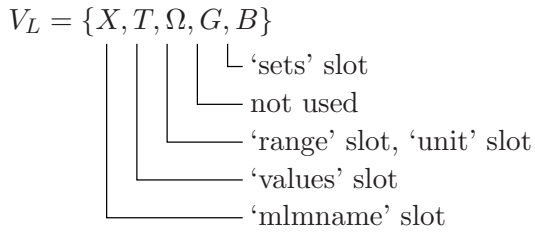

Figure 2.22: Representation of linguistic variable elements by an MLM

In contrast to the last extensions which largely concerned the functionality of the Arden Syntax elements, these extensions affect the structure of the knowledge slot of an MLM. The first two categories of an MLM are not modified. The name X of the variable is identified by the name of the MLM. The knowledge category is redefined by the following slots.

Type (coded, required) The type slot defines which slots are contained in the knowledge category. A linguistic variable knowledge category is defined by the keyword 'linguistic variable', which implies that the following slots exist: 'values', 'input', 'defuzzification', 'range', 'unit', 'sets'. Some of these slots are optional.

Values (coded, required) A linguistic variable is defined by a set of linguistic terms. The values slot defines these values as a list of Arden Syntax terms. For example,

```
values: 'decreased', 'normal', 'increased';;
```

Input (structured, optional) The 'input' slot defines the data source for the linguistic variable. Valid sources are data queries to the data base or references to other MLMs that return a linguistic variable as a result of their action slot.

Defuzzification (coded, optional) This slot defines a defuzzification method that can be used to compute a numerical value of the linguistic variables. So far, only two methods—CoM and CoG—are supported.

Range (coded, required) The range slot defines the universe of discourse Ω as a range of numerical values by Arden Syntax numbers as lower and upper delimiter. For example,

```
range: 0, 100;;
```

Unit (coded, required) This slot defines the unit of the universe of discourse as an Arden Syntax term. This information is only informal[42].

Sets (coded, required) Each value of the linguistic variable is defined by a fuzzy set. The sets slot contains all fuzzy set definitions.

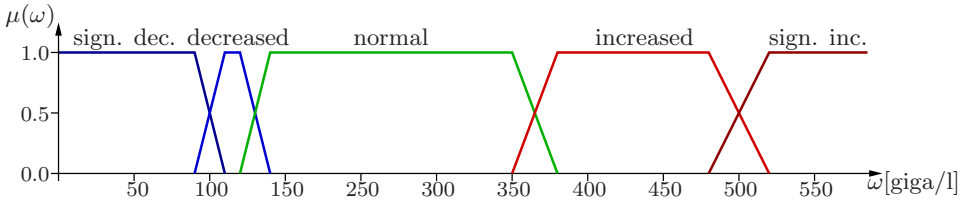

Figure 2.23: Linguistic variable 'blood count, platelets'

As a first example, the linguistic variable 'blood count, platelets' that is defined in figure 2.23 is represented by an MLM in example 28.
The example comprises five values from 'significantly decreased' to 'significantly increased'. Their fuzzy sets are defined on a numerical universe of discourse in 'giga/l', the numerical input value can be directly read from the data base by the read statement defined in the 'input' slot. A defuzzification slot has not been defined, as this variable represents a laboratory examination and is therefore used only as an input variable.

Example 28: (Linguistic variable 'blood count, platelets')

```
maintenance:
    title: Linguistic variable 'blood count, platelets';;
    mlmname: blood_count_platelets;;
    arden: 1.2flv;;
    version: 1.00;;
    institution: Siemens Medical Solutions, University of Vienna
                 Medical School;;
    author: Sven Tiffe;;
    specialist: ;;
    date: 2002-07-04;;
    validation: testing;;
library:
    purpose:
        ;;
    explanation:
        The fuzzy sets used in this MLM are based on the
        definition of the concept 'blood count, platelets'
        as used in the medical expert system CADIAG-II;;
    keywords: ;;
    citations: ;;
knowledge:
    type:   linguistic variable;;
    values: 'significantly decreased', 'decreased', 'normal', 'increased',
            'significantly increased';;
    input:  read last { $BCPObservation, $Value
                        where (( $Dkz = "B0829,N04") and
                               ($patientId = %patientId)) };;
    range:  0, 700;;
    unit:   'giga/l';;
    sets:
      'significantly decreased' := linear((90,1.0), (110,0.0));
      'decreased' := linear((90,0.0), (110,1.0), (120, 1.0), (140, 0.0));
      'normal'    := linear((120,0.0), (140,1.0), (350, 1.0), (380, 0.0));
      'increased' := linear((350,0.0), (380,1.0), (480, 1.0), (520, 0.0));
      'significantly increased' := linear((480, 0.0), (520, 1.0));;
end:
```

[42] In combination with an intelligent unit management system, this information could prevent incorrect computations due to varying units within the data model.

2.4.1 Initialization of linguistic variable MLMs

A linguistic variable MLM can be used in other MLMs as a template ("data type") for individual *instances* of the linguistic variable. The template defines, for example, which linguistic values can be assigned to the instance of the variable. The instance then represents individual values during runtime. This means that a linguistic variable MLM is not directly executed, but merely provides the necessary information for using it.

2.4.1.1 Definition and initialization

As a linguistic variable is defined as an independent MLM, it first has to be referenced before it can be used within the decision logic of another MLM. A linguistic variable is usable as an *input* variable that has to be initialized by a numerical input value, or as an *output* variable that does not have to be initialized as every value is separately assigned to it.

Definition 19 (Linguistic variable statement, fuzzification):

> *The* **linguistic variable statement** *defines a reference to an MLM that has to be of the 'linguistic variable' type. The reference can be assigned to a local variable that represents an instance of the linguistic variable.*
>
> <var> := **linguistic variable** <term>
> *or*
> <var> := **linguistic variable** <term> **from institution** <string>
>
> *Optionally, the* **fuzzification** *of the variable can be done by initializing it either manually, by defining a numerical input value using the keyword 'with', or automatically, by evaluating the input slot of the linguistic variable MLM (if defined). If the variable is not initialized, every value of the variable is set to a degree of 0.0.*
>
> <var> := **init linguistic variable** <term>
> *or*
> <var> := **init linguistic variable** <term> **with** <numerical>
> *or*
> <var> := **init linguistic variable** <term> **from institution** <string>
> *or*
> <var> := **init linguistic variable** <term> **from institution** <string>
> **with** <numerical>

2.4.1.2 Definition of data sources

The 'input' slot can either contain one read statement, as used in the data slot of common MLMs, or one or more references to other MLMs. The read statement is defined without using a local variable, as in example 28.

When other MLMs are referenced as data source it is necessary that they return, as the result, a linguistic variable of the same type as the current one. The local values are then

initialized by using the values of the returned linguistic variables. If one linguistic value is present in more than one input variable, its highest degree is chosen.

Example 29: (Use of MLMs as data source for a linguistic variable)

A linguistic variable MLM 'TEST_LV' defines in its input slot two references to other MLMs.

```
input: /* the input values are read from every referenced MLM and
          automatically aggregated to one input value */

         mlm 'TOSCA_RULE_BLOCK_8';
         mlm 'TOSCA_RULE_BLOCK_7';;
```

The referenced MLMs have to use the current linguistic variable as output variable. Therefore, they have to reference the MLM that defines the variable in their data slot without initializing it. In the logic slot, values are assigned to the variable, which is then returned as result in the action slot.

```
data:   test_lv := linguistic variable 'TEST_LV';
        ...
logic:  ...
        let test_lv be 'increased';
        ...
action: return test_lv;;
```

2.4.1.3 Definition of the fuzzy sets

Each value of a linguistic variable is defined by a fuzzy set or, more exactly, by a compatibility function. The definition consists of a function type and a list of points (x, y) that define the curve of the compatibility function in the form of:

```
'term' := <type>((x1,y1), (x2,y2), ... , (xn, yn));
```

So far, only 'linear' is a valid keyword for `<type>` and defines a parameterizable linear function on $(x_1, y_1), \ldots, (x_n, y_n)$. The degree of compatibility of a value $\omega \in \Omega$ is determined after detecting the relevant segment $x_i \leq \omega \leq x_{i+1}$ by applying equation 2.8.

$$\mu(\omega) = y_{i+1} - \frac{(x_{i+1} - \omega)(y_{i+1} - y_i)}{x_{i+1} - x_i} \tag{2.8}$$

The point with the smallest x-coordinate x_l defines the degree of compatibility $\mu(x_l) = y_l$ for all $\omega \leq x_l$, the one with the greatest x-coordinate x_r defines the degree of compatibility $\mu(x_r) = y_r$ for all $\omega \geq x_r$ (figure 2.24 shows an example).

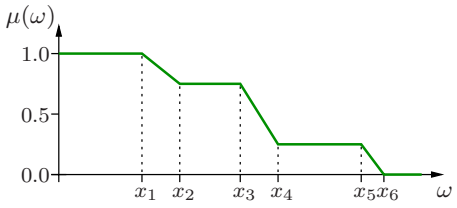

Figure 2.24: Piecewise linear definition of a compatibility function

The main difference between this function type and the one used for fuzzified comparisons (equations 1.11 to 1.13) is that the transition from 'false' to 'true' or vice versa can be

defined piecewise linear. The common s-, z-, or sz-functions can be also defined by using the notation. The next example uses such functions to define three (abstract) values 'decreased', 'normal', and 'increased'.

Example 30: (Definition of linguistic terms based on common compatibility functions)

A typical s-function is defined as

```
'decreased' := linear((x1,1), (x2,0));
```

or

```
'decreased' := linear((x_left,1), (x1,1), (x2,0), (x_right,0));
```

where 'x_left' and 'x_right' are the lower and upper delimiters of the universe of discourse. Both definitions are equivalent; the additional specification of the borders might result in more figurative definitions of the fuzzy set that might be easier to comprehend.

Analogously, a typical z-function can be defined as:

```
'increased' := linear((x1,0), (x2,1));
```

A typical sz-function can be defined as:

```
'normal' := linear((x1,0), (x2,1), (x3,1), (x4,0));
```

2.4.2 Use of linguistic variables by common MLMs

Currently, linguistic variables can be used to be compared with linguistic values, to assign such values to them, or be defuzzified. They cannot be directly used with arithmetic operators or comparison operators. The common Arden Syntax operators behave as individually defined for the use of invalid arguments.

2.4.2.1 Comparison of linguistic values

If a decision depends on the value of a linguistic variable, it has to be compared to a constant term. Such a comparison can be done by the *linguistic variable is comparison operator* that returns a fuzzy truth value.

Definition 20 (linguistic variable is comparison operator):

To compare an instance of a linguistic variable with a value $t \in T$ of the variable definition, the comparison operator 'is' is used:

<1:fuzzy> := <1:linguistic variable> **is** <1:term>

The operator returns the 'degree of assignment' of the term that is converted to a fuzzy truth value.

This operator can be further used to extract one single value, for example for including it into textual messages.

Example 31: (Linguistic variables: use of comparisons)

This example uses the linguistic variable 'blood count, platelets', which is defined in example 28.

```
data:  platelets := linguistic variable 'blood_count_platelets';
       ...
logic: sig_decreased := platelets is 'significantly decreased';
       decreased     := platelets is 'decreased';
       normal        := platelets is 'normal';
       increased     := platelets is 'increased';
       sig_increased := platelets is 'significantly increased';

       msg := "The platelets blood count value of the current patient is"
       || "\n'significantly decreased' by a degree of " || sig_increased
       || "\n'decreased' by a degree of " || decreased
       || "\n'normal' by a degree of " || normal
       || "\n'increased' by a degree of " || increased
       || "\n'significantly decreased' by a degree of " || sig_increased;
```

If the variable 'platelets' has not been initialized, the message would look like this:

"The platelets blood count value of the current patient is
'significantly decreased' by a degree of 0.0
'decreased' by a degree of 0.0
'normal' by a degree of 0.0
'increased' by a degree of 0.0
'significantly decreased' by a degree of 0.0"

If the variable is initialized with a numerical value, for instance by

```
data:  platelets := initialize linguistic variable 'blood_count_platelets' with 250;
```

then the message would look like this:

"The platelets blood count value of the current patient is
'significantly decreased' by a degree of 0.0
'decreased' by a degree of 0.0
'normal' by a degree of 1.0
'increased' by a degree of 0.0
'significantly decreased' by a degree of 0.0"

In the next example, the input value for the variable is automatically retrieved from the data base by executing the input slot, as no argument is passed along with the initialization option:

```
data:  platelets := initialize linguistic variable 'blood_count_platelets';
```

If the query returned 360, the variable would be between normal and increased:

"The platelets blood count value of the current patient is
'significantly decreased' by a degree of 0.0
'decreased' by a degree of 0.0
'normal' by a degree of 0.67
'increased' by a degree of 0.33
'significantly decreased' by a degree of 0.0"

2.4.2.2 Assignments to linguistic variables

Like the assignment of values to common variables, it is possible to assign linguistic values to linguistic variables. To emphasize the linguistic character of this variable type, the linguistic representation of the assignment operator is used exclusively.

Definition 21 (Linguistic variable assignment statement, degree of assignment):

Like the common assignment statement, the **linguistic variable assignment statement** *places a value defined by a term into a local variable that has to be a linguistic variable. If the variable has not been defined as a linguistic variable in the data slot, then the local variable is set to null.*

let <identifier> **be** <term>

Each value can be assigned by a degree from 0.0 *to* 1.0*. A* **degree of assignment** *of* 0.0 *means that the variable absolutely does not represent the value; a degree of* 1.0 *means that the variable absolutely does. This degree is defined by the conditional context in which the assignment is executed. If the same value is assigned to one linguistic variable more than once, the highest degree of assignment is chosen.*

The degree of assignment can additionally be modified by an optional numerical argument from 0.0 *to* 1.0*.*

let <identifier> **be** <term> with <number>

If the modifier is used, then the degree of assignment is the result of the multiplication of the current conditional context with the modifier.

By using the comparison operator and the assignment statement, fuzzy control rules can be easily implemented by Fuzzy Arden.

Example 32: (Linguistic variables: use of assignments)

The dose for regular insulin medication depends on the blood glucose level of the patient. Four rules are defined for the calculation of the dosage, provided that 4 international units regular insulin reduce the blood glucose level approximately by 100 mg/dl [H^+89]. To implement them, two linguistic variables 'blood_glucose_level' and 'insulin dose' are defined and used as follows:

```
if if blood_glucose_level is 'slightly increased' then
  let insulin_dose be 'none';                          /*  0 IU */
elseif blood_glucose_level is 'increased' then
  let insulin_dose be 'low';                           /*  4 IU */
if blood_glucose_level is 'significantly increased' then
  let insulin_dose be 'medium';                        /*  8 IU */
else
  let insulin_dose be 'high';                          /* 12 IU */
endif;
```

The terms of the variable 'blood_glucose_level' could be defined as follows:

```
'slightly increased'      := linear((225,1.0), (275,0.0));
'increased'               := linear((225,0.0), (275,1.0), (325,1.0), (375,0.0));
'significantly increased' := linear((325,0.0), (375,1.0), (425,1.0), (475,0.0));
'absolutely increased'    := linear((425,0.0), (475,1.0));
```

The terms of the result variable could be defined by simple triangles:

```
'none'   := linear((0,1.0), (4,0.0));
'low'    := linear((0,0.0), (4,1.0), (0,0.0));
'medium' := linear((4,0.0), (8,1.0), (12,0.0));
'high'   := linear((8,0.0), (12,1.0), (16,0.0));
```

If the blood glucose level would be 250 mg/dl, the first condition would yield $F0.5$, as would the second one too. Thus, the linguistic variable 'insulin_dose' would be set to '0IU' by a degree of 0.5 and by the same degree to '4IU'[43].

2.4.2.3 Textual representation and defuzzification of linguistic variables

If linguistic variables are used to compute the result of rules, it is necessary to output the result in an appropriate way.

As mentioned earlier, it is possible to extract individual values of the variable by using the 'is' comparison operator. The operator returns the degree by which the value was assigned, which can then be included, for instance, in a textual message.

When linguistic variables are used by the string concatenation operator (or, analogously, by the 'to string' operator) it is converted into a textual representation that provides the name of the variable and each value with the corresponding degree of assignment. Like the representation of 'duration' data types, the textual representation is institution specific and may vary.

In cases where a numerical value is required, for example to dose a medication, the variable can be defuzzified using the 'defuzzify' operator.

Definition 22 (Defuzzify operator):

The **defuzzify operator** *returns a numerical value based on the defuzzification of the current linguistic values of a linguistic variable. Its form is:*

<numerical> := **defuzzify** <identifier>;

The mathematical defuzzification is defined by the respective slot of the linguistic variable MLM.

$$CoM := y = \frac{\sum_i x_i \mu(x_i)}{\sum_i \mu(x_i)};\ x : \text{average of plateaus}$$

$$CoG := y = \frac{\int_{y \in Y} y\mu(x_1, \ldots x_n)(y)dy}{\int_{y \in Y} \mu(x_1, \ldots x_n)(y)dy}$$

[43] This example is kept simple for easy comprehension; it shows a linear correlation between the condition and the result. In practice, more complex systems behave in a non-linear fashion and therefore benefit from the underlying fuzzy sets more than the presented one (and, in practice, high blood glucose would probably be treated differently).

Chapter 3

Application of methods

This chapter presents three projects that make use of the extensions presented in the last chapter. The extensions have been implemented by a Java-based software and were evaluated in two clinical projects.

3.1 Design and implementation of the rules engine

To test Fuzzy Arden in terms of realizability and applicability, a software named "rules engine" was developed. It provides a runtime environment for Arden Syntax rules. This rules engine is a modular autonomous service which was connected to a separate information system (IS) and supports nearly all elements of Arden Syntax to be used for writing MLMs as specified in [Hls99], and all the extensions defined in this work. The rules engine was implemented using Java[44] 1.3, an object-oriented programming language. The development and verification of the rules engine took about 18 months, starting in the summer of 2000.

The engine is started within a Tomcat servlet-server[45]. Tomcat is an open-source application server based on Java technology, which serves Java servlets and Java Server Pages (JSP). Servlets are Java classes which are dynamically compiled during runtime by the Tomcat server; JSPs are a hybrid of HTML files and Java code, which is also compiled dynamically. The Tomcat server can be used as a stand-alone or in combination with a pure web server, such as Apache, which serves static web pages. For this project the server is used as a stand-alone and starts the rules engine as a separate thread. Further, it provides administrational user interfaces based on HTML and Java servlets.

The basic behavior of the rules engine is to wait for events that occur in the IS and to run those MLMs which were written to react to the events that occurred. Figure 3.1 illustrates the connections between rules engine and IS.

The communication between rules engine and users or the reaction to events or data provided by devices has to be handled by the IS. For this work, the rules engine has been connected to the Siemens MedStage[46] communication platform that provides all functionalities for data storage, communication, and security. As the engine was designed independent of any specific information system and the architecture of the engine was designed in a modular way, only a small part of it was implemented IS-specific, by writing

[44] http://java.sun.com
[45] http://jakarta.apache.org/tomcat/
[46] http://www.medstage.com

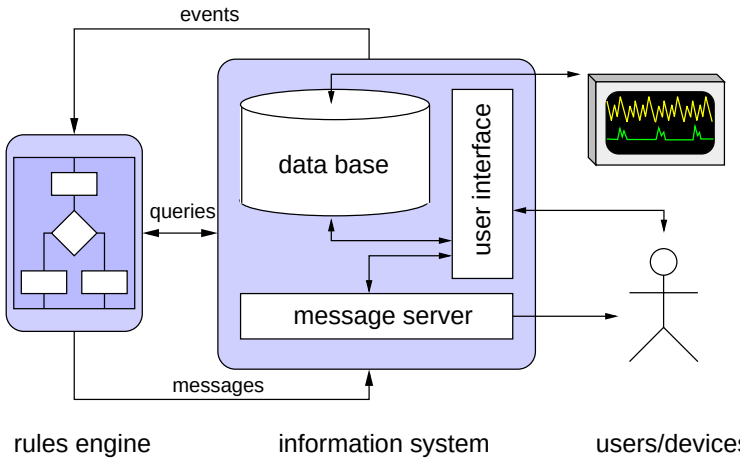

Figure 3.1: Communication between information system and rules engine

interfaces that communicate with the MedStage system. Basically the rules engine is connected to the IS by three interfaces. First, it listens to events that occur in the IS. Next, the MLMs that are executed within the rules engine can access the data base by sending queries to the IS. Finally, the results of the rules can be sent to the message server of the IS, which distributes incoming messages to their receivers.

The interfaces of the rules engine only define the type of functionality in terms of communication, whereas the functionality itself is provided by the institution-specific implementations of the interfaces. The current implementation partly uses MedStage libraries for a secure and reliable communication with the IS. The message system used for communicating events is based on Java Message Service (JMS) technology. If the rules engine has to be connected to a different IS, only this set of interface implementations would have to be rewritten.

In the following, some aspects of the implementation will be presented in detail, starting with the Java representation of Medical Logic Modules.

3.1.1 Java class model

The rules engine is designed to dynamically load, unload, and reload MLMs. To load a module from its textual representation, the content of the file has to be read by a compiler that produces a set of linked Java objects which represent single elements of the MLM and which form an 'object tree' that represents the entire MLM. This object tree is based on a class model that comprises all classes needed to represent and to execute Arden Syntax MLMs[47].

Classes can be grouped by *packages*, for example if they are functionally related. Figure 3.2 shows the hierarchy of the packages used to manage the classes of the rules engine. Packages can be nested; therefore the package 'arden' contains all other packages as top-level package.

The package hierarchy starts with three main packages, namely 'engine', 'parser', and 'mlm'. The first contains those classes that provide the environment to execute Arden

[47] A class represents a data type that can be interpreted as a template that includes data structures (attributes) and program functions (methods), for instance to access and alter attributes of a class. During runtime, individual objects (instances) of a class can be created. An introduction to object-oriented thinking can be found in the introduction to "Thinking in Java" [Eck00] (also available online at http://www.mindview.net/Books/TIJ/).

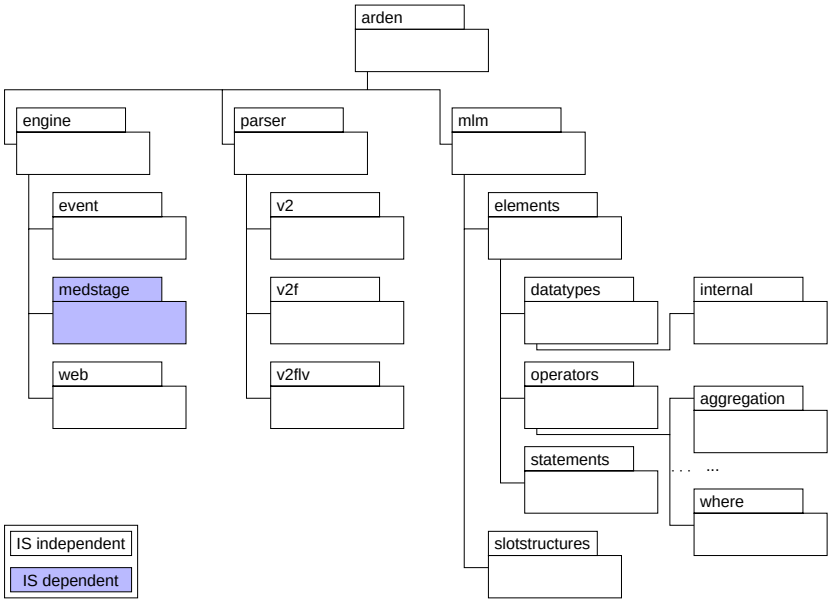

Figure 3.2: Package diagram of the class model

Syntax MLMs, such as interfaces to data base access, the event system, or some features of the web-based user interface. Additionally, it includes the only IS-dependent classes that implement specific parts of the communication interfaces.

The package 'parser' contains a set of parsers for translating the ASCII text representation of an MLM into Java. A parser is a program that analyzes the syntax of an input file, which usually is text-based. The parsers used to translate MLMs are based on the BNF of the Arden Syntax specification and use lexers that pre-process the MLMs transforming the textual representation into a sequence of logical syntactical tokens.

The main 'parser' package contains those parsers that read the constant parts of an MLM, which are largely independent of the applied version of Arden Syntax (all elements of the 'maintenance' and 'library' categories). Based on the value of the 'version' slot in the 'maintenance' category, the parser of the corresponding sub-package is chosen. This separation of the parsers and this structure of the packages provides extreme flexibility in integrating new versions of Arden Syntax on the syntactical level. New parsers have only to be defined within a package whose label equals the version of Arden Syntax that has to be parsed. Two new parser versions ('2f' including fuzzy extensions and '2flv' including linguistic variables) which are able to process the current extensions are defined. The parsing process is described in detail later.

The last package 'mlm' contains all classes needed to represent any Arden Syntax MLM by a Java object tree. The base package contains classes that represent the basic structure of an MLM, such as the categories, or the different types of slots. The 'slotstructures' package provides classes for representing those slots, which are structures but do not include algorithms, such as the 'sets' slot of linguistic variables. The 'links' slot of the library category is another example of such slots, however all slots of this category are stored as plain text.

The 'elements' package models those Arden Syntax structured slots that contain algorithms and contains hierarchic sub-packages according to the structure of the specification. Basically, three categories of elements have been identified: data types, such as

'Boolean' or 'number', operators, and statements. Operators are additionally ordered by sub-packages that correspond to their category in the specification document.

The entire class structure is not described in detail. Two aspects of the class model, concerning the representation of an MLM, are explained next. Figure 3.3 shows an outtake of the class structure as simplified UML[48] class diagram. The diagram only shows selected attributes and methods of the presented classes.

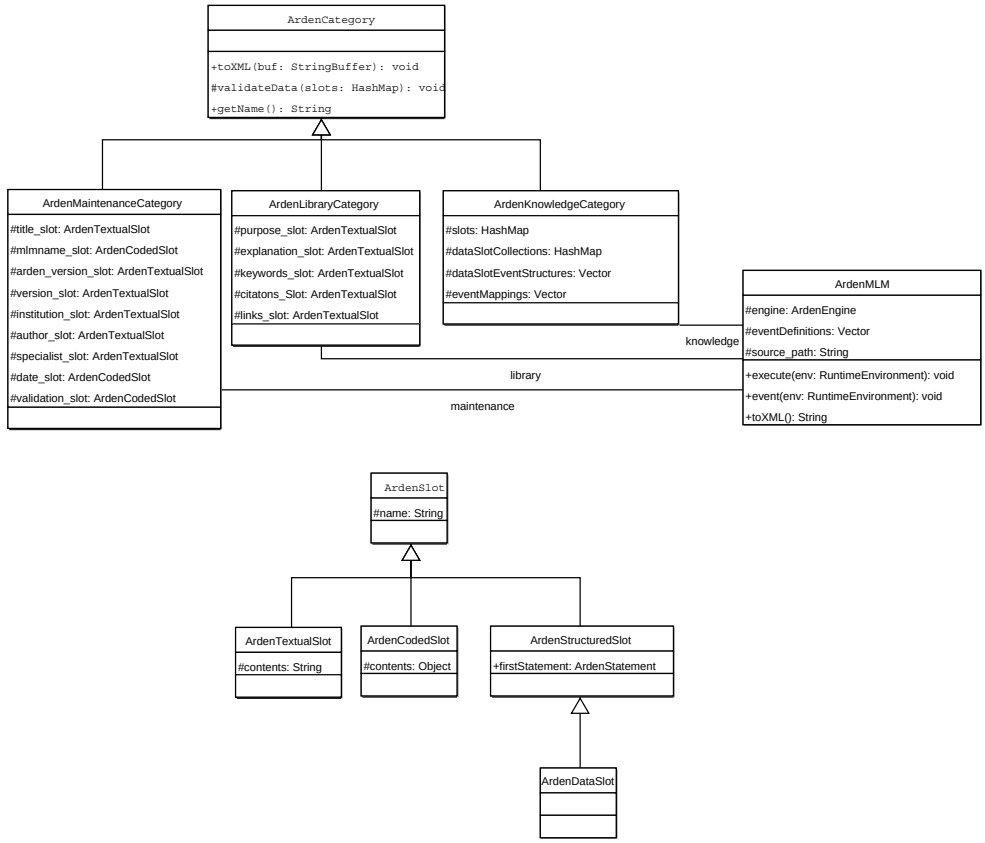

Figure 3.3: Java class structure of an MLM

On the right side of the diagram, the class 'ArdenMLM' represents one MLM as a top-level root object of the object tree. Every MLM representation which is loaded into the engine is referenced by exactly one instance of this class. Despite to some information required during run-time it contains references to three objects, which represent the three categories that belong to an MLM. Each category object references a set of slots that are objects of one of the three possible slot classes.

All category types share certain properties which have been defined in the abstract 'ArdenCategory' class[49]. Each category type is represented by an own class that inherits the properties of the abstract category class. Classes that inherit other ones are also termed *child classes* and are marked by the arrow symbol in the diagram. Objects that repre-

[48] Unified Meta Language, modeling technique used for "Object Oriented Analysis and Design".

[49] Unlike normal classes, abstract classes cannot be directly used to create objects of this type. They may be used to define attributes and methods which are then used by classes that inherit this one. Further, abstract classes may contain abstract methods that define the name and the data types of arguments and results, but not the algorithm of the method that has to be implemented by inheriting, non-abstract classes.

sent textual slots ('ArdenTextualSlot') simply contain their value as String. Coded slots ('ArdenCodedSlot') may reference any kind of structured information using the 'Object' class, which is the top-level class for all Java classes. For instance, entries of the 'links' slot could be represented as a sequence of objects by one Java 'Vector' object. Structured slots ('ArdenStructuredSlot') contain expressions and statements referenced by one object that represents the slot's first statement.

The 'maintenance' and 'library' categories contain a fixed set of slots[50]. The knowledge category was designed to contain a dynamic set of slots to match both, the requirements of classic Arden Syntax MLMs and those of linguistic variable MLMs, which are significantly different.

As the structured content of the 'data', 'logic', and 'action' slot is referenced only by the first statement, statements have to create a sequence by referencing their successor. (For technical reasons, the previous statement is also referenced.) Further, statements have to reference the expressions that are used to create or alter the data. An outtake of the class diagram for statements and operators is shown in figure 3.4.

Analogous to the model of the categories, one abstract class defines some basic attributes and methods of a statement. (In the diagram—which however is incomplete—only two attributes and three methods are shown.) The attributes are used to create the sequence of statements by referencing the previous and following one, the methods include functionalities to execute the statements. Further, some abstract methods which have to be implemented by the child classes are defined. The most important one is the 'eval' method which is used to execute an MLM and will be described in detail later. Further, every child has to provide a method to represent itself as XML text.

Every Arden Syntax statement is represented by one class. As an example, the diagram shows three of them that inherit the abstract statement class ('ArdenAssignment', 'ArdenIfThen', and 'ArdenWrite') representing the common structure of these statements classes.

The assignment includes one variable identifier object of the class 'Identifier' and the expression that represents the value, which has to be assigned to the variable. The expression itself can also be an object tree; only the "root" is referenced directly. An identifier can either be a single identifier or a group of identifiers (for example, assigning data by the 'argument' operator).

Analogously, the 'if-then' statement references an expression that represents the condition and two statements that represent the corresponding first statement of the two alternative code blocks that might be executed. The 'write' statement references an expression that represents a value to be written to a destination, which is represented by a variable (referenced by an identifier).

More exactly, as concrete data values are only available during the execution of an MLM, references to expressions represent only combinations of variables, constants, and/or operators that yield one concrete data value after the execution. Such combinations are represented by an object tree that uses child classes of the abstract root class 'ArdenStructure', which defines some basic functionalities, such as the evaluation method and the XML-export method, which have been mentioned in the context of statements.

Constants can be of any Arden Syntax data type, each represented by an individual class. They all share their attributes 'primary time' and 'degree of presence', which are defined

[50] As for the current use of the rules engine, the coded information of the 'library' slots were not required, they are handled like textual slots.

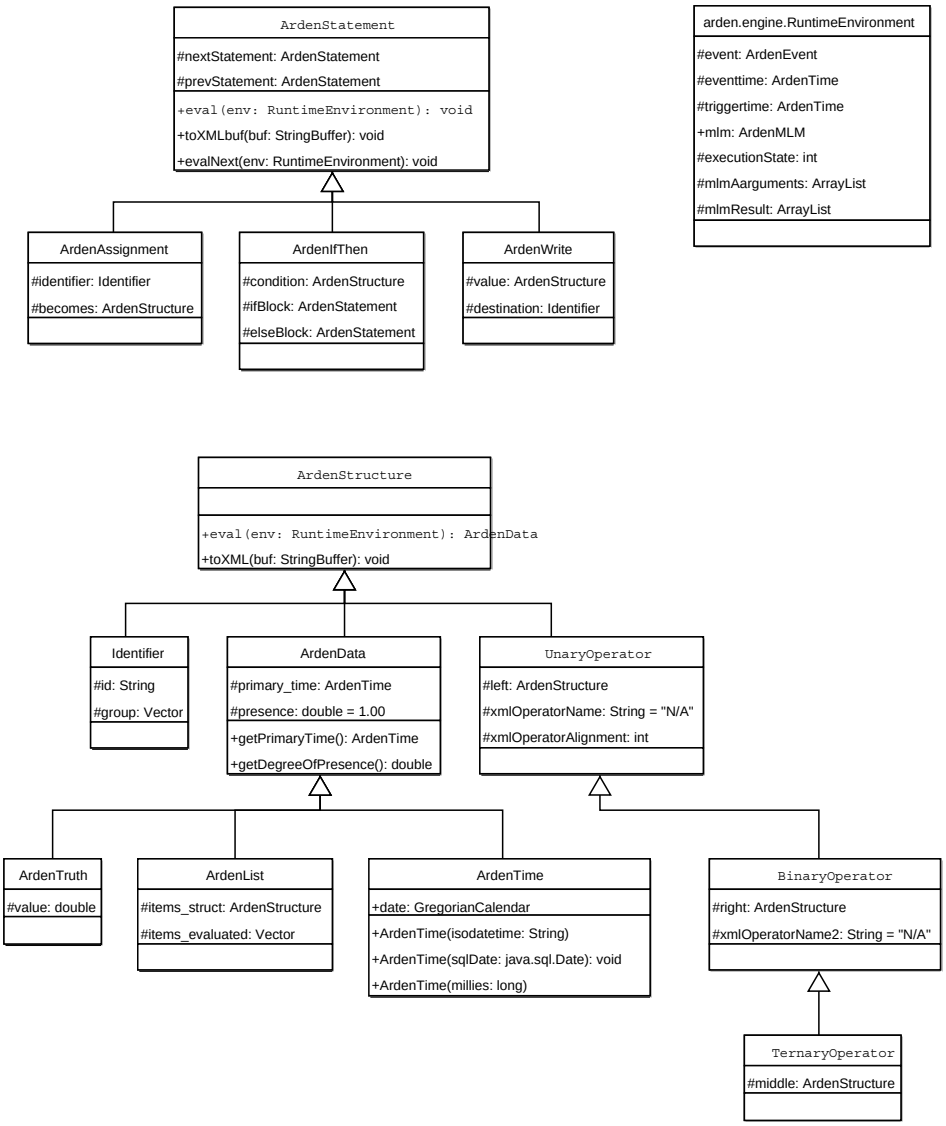

Figure 3.4: Java class structure of statements and operators (outtake)

in the abstract class 'ArdenData'. Constants have no primary time; however, the data type classes are also used to represent dynamic data during the execution of an MLM. As an example, the diagram shows three data type classes for representing truth values ('ArdenTruth'), lists ('ArdenList'), and time values ('ArdenTime').

Operators can be of unary, binary, or ternary type and reference one to three expressions. Every single operator is implemented by a Java class that inherits one of these abstract operator base classes.

In summary, the Java representation of an MLM can be interpreted as an object tree with an 'ArdenMLM' object as the root. Every class that is relevant for the execution of the MLM provides an 'eval' method that has to provide all functionality of the element. For instance, the evaluation method of the binary logical operator 'or' knows how to combine two or more truth values and the ternary comparison operator 'is within to' knows which types of data can be compared and how the comparison should work.

3.1.2 Runtime processes

The MLM ASCII files are compiled by parsers to Java objects, which are based on the class library shown before. All MLMs are loaded during startup of the engine from defined folders in the file system. For this work, other sources, such as databases were not needed but can be easily added, as the parsers only require a stream of characters as input source.

Strictly speaking, the parsers are composed of two elements. A 'lexer' processes a stream of characters and divides it into a stream of logical tokens. A 'parser' uses this stream of tokens as input and validates the stream of tokens in means of syntactical correctness. In parallel, the parser creates the Java object tree as result.

In this document, the lexer and parser are referred to as a single unit by using the term *parser*. All parsers used in this work are based on JLex[51] as lexer and Jay[52] as parser definition tool. These tools use formal description files as input and create Java classes that represent the corresponding lexer and parser.

The parser had to be able to validate different versions of Arden Syntax that use different syntactical rules—Version 2, the fuzzified version that includes the main fuzzy operators, and finally the version that includes the concept of linguistic variables. To keep the parsing process flexible and easily extendable, each version of MLMs is parsed by an own parser. On the other hand, the first two categories of MLMs are mainly constant over the different versions; therefore the parsers have been separated into two parts.

The first part processes the maintenance and library categories and creates a Java object tree with the MLM object and the first two category objects. The entire knowledge category is separately kept as plain text. Then, the value of the 'Arden' slot that defines the Arden Syntax version of the MLM is evaluated and, depending on this value, another parser that compiled the knowledge category is called[53]. All parser classes have the same name but are located in different Java packages[54]. For instance, the knowledge category sub-parser for Arden Syntax version 2.0 is located in package 'arden.parser.v2' whereas the one for the fuzzified version is located in package 'arden.parser.v2f'. Version 2 MLMs define the value "2" in the arden-slot, the fuzzy ones define "2f". The correct class is then referenced by using the Java reflection technology that can be used to dynamically locate the correct class, based on its fully qualified class name (that is, package name plus class name).

The parser ignores all contents within curly braces. These mappings are parsed separately during the initialization of the MLMs by the rules engine. This process is done when each MLM object is added to the knowledge base subsystem. Parsing the mappings is either done by the classes that use the mappings, such as destination statements or event statements, or in case of read statements by a separate parser, as the syntax is rather complex. If the initialization ends without errors, the knowledge base subsystem can be used to easily find, remove, or update single MLM objects[55].

[51] http://www.cs.princeton.edu/~appel/modern/java/JLex/

[52] http://www.informatik.uni-osnabrueck.de/bernd/jay/

[53] Arden Syntax version 1 MLMs do not define the version slot and this version is not supported. However, the differences between version 1 and version 2 are marginal, older MLMs are therefore parsed as version 2 MLMs. As the support of such MLMs is not the scope of this engine, this method has not been tested further.

[54] Each class name must be unique within one package.

[55] The addition of new MLMs can only achieved by restarting the engine, as this feature was not needed for the evaluation of the methods. As the methods for adding and removing MLMs from the knowledge base are implemented, this feature would just require a suitable user interface.

3.1.2.1 Event preparation and MLM call

In general, the evocation of an MLM is initialized by an event. Each MLM defines one or more events in the data slot by event mappings that are assigned to local variables. These variables can then be used in the evoke slot to define evocation conditions.

Event-related processes are illustrated in figure 3.5. An information system broadcasts events that are received by the rules engine and can be optionally sent to a scheduler for delayed MLM execution. The communication layer is not restricted to a specific technology, as the rules engine can use more than one event interface. Currently the Java Message Service[56] (JMS) is used as the communication method.

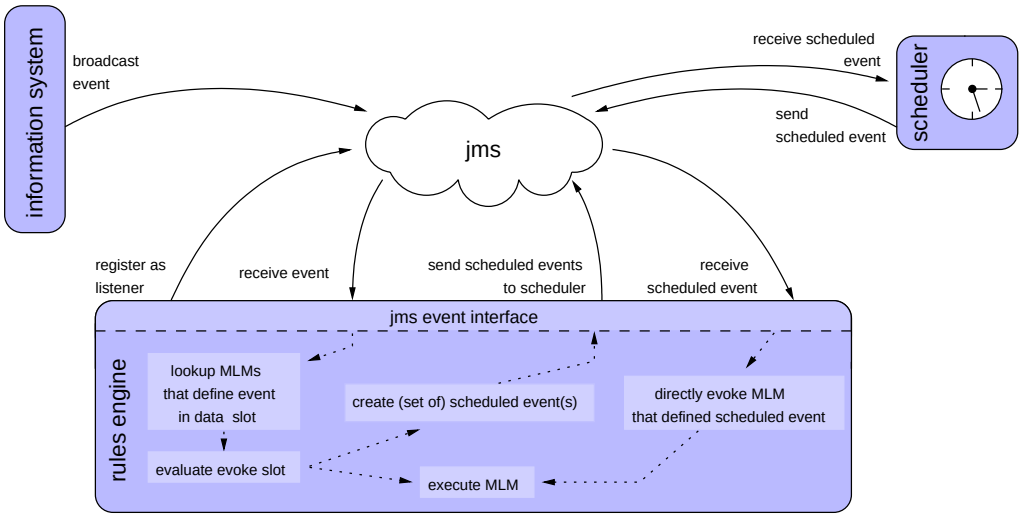

Figure 3.5: Receiving and scheduled events

The JMS can be used to communicate between two specific systems (point to point) or to broadcast from one system to an undefined number of listeners (publish/subscribe). As more than one system may be interested in being notified of events, the latter method is used. An event is sent to a specific channel ('topic') to which listeners, such as the rules engine, have to register initially. From then on, whenever a message is sent to this particular topic, the JMS interface will be notified and will receive the message. This interface and the structure of an event is described later.

Events can be sent by every system that is connected to the JMS topic. For example, an information system can create and fire an event by data base triggers if new data is available. In the current implementation, Java event objects that are defined by the MedStage system are created[57] and sent to the topic used by the rules engine. This Java object includes information about the origin of the event, a time-stamp and, optionally, further information.

The engine is notified by the interface and looks into the knowledge base for those MLMs which define the received event in the data slot. After the MLMs have been selected, the evoke slots which yield either 'true', 'false', or a set of Arden Time objects are executed. If the evoke slot yields 'true', then the particular MLM is executed instantly. If the slot

[56] http://java.sun.com/products/jms/

[57] Therefore, the JMS interface that translates IS-specific events to events that can be processed by the rules engine is an institution-specific part of the rules engine.

yields 'false', then the MLM is not executed. If one or more time-stamps are returned by the evoke slot, the MLM has to be executed later. The task to schedule such events is left to a separate external scheduler. The events are capsuled within a special message type that is sent directly to the scheduler using the JMS. The scheduler fires the events at the time defined by the time-stamps and sends them directly as scheduled events to the engine. Then the particular MLM is evoked directly.

This evocation process results in an asynchronous execution of MLMs. The duration between the action that caused an event and the execution of the associated MLMs depends on various factors, such as the communication of the event and the current load of the engine. Further, events are sent from the information system to an unknown number of receivers and the process that caused the event is continued immediately. A synchronous reaction to an event, for example when prescribed drugs have to be checked for incompatibilities by an MLM and the prescription has to be interrupted in the event of a positive result, is realized differently, for example without using the event system. This mechanism will be described later.

Regardless of whether an MLM is evoked instantly or delayed, each MLM execution is made within an own thread. This ensures that, in the event of execution errors, the rules engine keeps running. During the initialization of a new MLM execution process the system creates a *runtime environment* that is used by the execution instance to communicate with the rules engine; each environment provides its own variable space.

3.1.2.2 Runtime environment

The functionality of each statement and operator is defined, as mentioned before, in one method that each such class has to implement. The evaluation methods

```
public abstract void eval(RuntimeEnvironment env) throws ArdenRuntimeException;
```

and

```
public abstract ArdenData eval(RuntimeEnvironment env) throws ArdenRuntimeException;
```

are defined in the abstract classes of statements and operators and use a runtime environment object as argument. If any error that occurs during the execution of the MLM cannot be handled according to the specification of the current statement or operator, an exception is thrown; the exception interrupts the evaluation of the MLM and provides textual information about the error.

The runtime environment manages all information needed for one individual execution of the MLM, such as the values of local variables. Further, as the elements of the MLM object tree do not have any reference to the rules engine, the runtime environment is used as a link to the functionalities of the rules engine, providing all communication interfaces to the elements.

Whenever an MLM is executed, a new instance of the runtime environment is created and passed to the evaluation method of the first statement of the active slot. This statement calls the evaluation method of its expression (for instance, an operator, an identifier, or a data constant) that returns an object of the 'ArdenData' type.

The result of the expression is used within the evaluation method of the current statement as defined in the specification. For example, it might be assigned to a local variable or it

might be used to select the code block of an if-then statement that has to be executed next. When the current statement has completed its own execution, the next one is evaluated by calling its evaluation method. The current runtime environment is passed as argument. This continues until the last statement is reached or an error occurs, as mentioned earlier.

If a data value has to be stored in a variable, its identifier and the value is passed to the runtime environment that stores both information. Whenever the evaluation method of an 'Identifier' object is executed, which represents a variable during the execution, it calls the runtime environment and passes its own label as argument to receive the current value of the variable. This value is simply the result of the evaluation method and is returned to the expression (or statement) that used this 'Identifier' object as argument.

Therefore, each MLM is represented exactly by *one* Java object tree during runtime, which is used by *multiple* execution processes of one MLM. Yet, each execution process has an *individual runtime environment* that guarantees the separation of its own data from the data of other execution processes.

3.1.3 Design of the interfaces

The execution of Medical Logic Modules is controlled by the Arden Syntax Java class library described earlier. During run-time, these elements of the MLM representations may need interfaces to the information system that embeds the rules engine, for example to read data from the data-base or to send messages. This section describes some design aspects of such interfaces.

Figure 3.6 shows the schematic architecture of the rules engine. The communication to external software and systems is done via four interfaces, which are easily extendable. Each interface can be implemented by one or more modules with different features, for example to use different message transportation systems. So far, only interface features that were needed to evaluate the Fuzzy Arden concepts have been implemented.

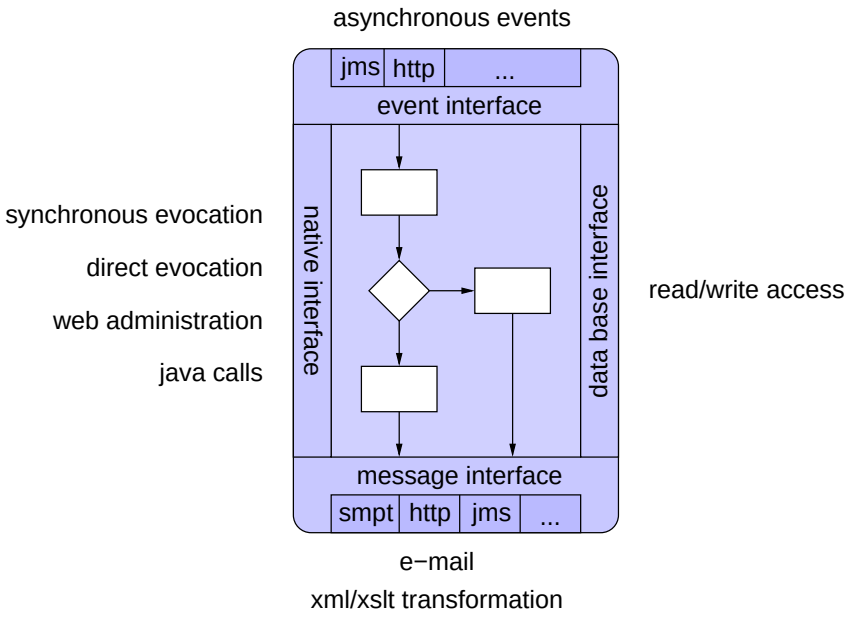

Figure 3.6: Architecture of the rules engine

An event interface receives asynchronous events and schedules events as described earlier. The data base interface provides methods to read and write to the data base. The message interface allows messages to be sent to recipients, using different transport protocols. The native interface provides methods for direct access to MLMs and functions of the rules engine. Further, it provides an internal interface to directly call Java methods from MLMs.

Every interface is defined by a set of methods using the Java *interface* technique. A Java interface defines a set of methods which have to be realized by a class that implements this interface. Therefore, all program interfaces between the core system and the interfaces are constant while the implementation of the interface functionality can be easily altered. More than one interface implementation can be used side by side, as shown in the figure.

All properties of the interfaces, such as the mail host for sending e-mails or the JMS host for the event system, are controlled by a property file that is loaded by the engine during startup.

3.1.3.1 Event interface

The first interface is used to receive events that initialize the evocation process of the MLMs, as described in section 3.1.2.1. Actually, two types of event transportation layers are supported by two implementations of the interface: the Java Message Service (JMS) and HTTP. Both implementations can receive XML-coded events. Additionally, as JMS supports sending and receiving Java objects, events can be received and sent as native Java objects by the JMS interface.

An event, whether it is represented by a Java object or is XML coded, consists of the following information:

name: The name of the event is a unique identifier that can be used to define unambiguous references to the event, for example within the event mapping in MLMs.

event time: This time stamp defines the moment when the event was created and fired. This time need not be identical with the time of the action that caused the event, such as a data base operation that triggered the event or an order entry that caused the data base operation.

user/system id: Events provide information about the originating system and the user whose action might have caused the event. This information can be used, for example, to log actions of the system, to provide additional information when explaining actions of the decision support system, or to identify the user who caused the event as a message receiver.

patient id: Each event is associated with a patient id. Usually, any type of changes in the patient data base are somehow related to an individual patient. The Arden Syntax specification does not define how a patient associated with a given event (and therefore with the individual execution of an MLM) has to be referenced. As an extension, the engine supports the keyword 'patientId' as an Arden Syntax constant that represents this value during the individual execution of the MLM.

additional data: Every event can be used to transport additional data represented by keys/value pairs. The data is automatically mapped to Arden Syntax data types.

Such events are received asynchronously to the actions that caused them; thus processes in the origin system may have continued since the broadcast of the event. Synchronous "events" are supported by the native interface which will be described later.

3.1.3.2 Data base interface

The second interface is used to communicate with the information system to query data from the data base. The rules engine defines an SQL-like language that additionally can use Arden Syntax variables within the queries. For performance reasons it is possible to use constraining expressions—temporal restrictions for instance—within the queries instead of using the corresponding native Arden Syntax operators.

Example 33: (Read statements: SQL-like query)

```
prepared_time := now - 6 hours;
(sys, dia)    := read {
                    $BPObservation, ($Systole, $Diastole)
                       where (($observationTime > %prepared_time)
                              and
                              ($patientId = %patientId))};
```

As shown in the example, local variables have first to be assigned before they can be used within the queries, where they are identified by a leading percent character. Data base tables and columns are identified by a leading dollar character. More than one query can be defined within one read statement to assure that the queries are processed within one data base transaction. Each column that is referenced by the queries is mapped to one local variable. The primary time is assigned automatically by individual mappings that define for every data base table one column whose value has to be used as primary time. These mappings can be added and removed dynamically. If for a data base query no primary time mapping is defined, the primary time of the data is set to 'null'.

Alternatively, data can be directly transported by events as key/value pairs. They can be accessed by appending the key to the keyword 'event' using a dot notation. Further, the primary time of the data value is defined by an additional key and has to be transported in the event too. The reason for creating this transport method was to keep the communication overhead as small as possible when only a few bytes have to be transported.

Example 34: (Read statements: data transported by an event)

```
(sys, dia) := read { %event.sysBP:observationTime%;
                     %event.diaBP:observationTime% };
```

3.1.3.3 Message interface

The Arden Syntax does not specify the definition of message destinations. Like the definition of events or data base queries, destinations are defined in an institution-specific way by mappings[58]. The presented engine basically defines destinations by a message type and the destination, separating these two units of information by a colon:

```
destination{ type:dest }
```

The message type defines the form and the transport layer by which the message has to be delivered. Therefore, the value and syntax of 'dest' directly depends on the message type. So far, the following message types are supported:

[58]The HL7 Arden Syntax SIG defined a new 'structured message' format that includes both the message content and information about the destination. However, this format is not supported yet.

file: The keyword 'file' defines the message to be written as plain text into a file. The filename is defined by the value of 'dest' and can contain 'patientid' as a keyword, which will be replaced by the current patient id.

console: Debug messages can be directly printed on the console in which the rules engine has been started. Three different priority levels, namely 'debug', 'log', and 'error', can be defined; these may be automatically filtered out according to the current run-time settings of the rules engine.

jms: Messages can be sent using the JMS point-to-point communication. The receiver is defined by 'dest' which represents one specific message queue.

mailto: Plain text messages can be sent via e-mail to one specific receiver. The e-mail address is defined by the value of 'dest'.

xmlmailto: If the message content is pure XML, it can be sent by e-mails with their MIME type set to "text/xml". E-mail clients that are able to handle XML mails can process the content or display it using a web browser.

xml2html_mailto: XML coded messages may be converted into HTML by using XSLT style sheets. The value of 'dest' defines both the e-mail address of the receiver and the URL of the XSLT style sheet that has to be used to convert the XML file into HTML. The e-mail is then sent with MIME type set to "text/html" which can be directly displayed by an e-mail client that supports HTML.

The e-mail destinations can only be used for direct communication, as they have not been extended to support roles yet. The transmission of a message to an unknown receiver who is identified by a rule, such as "head nurse", could be realized either by using an explicit message server that covers all communication aspects or even by using JMS message queues.

3.1.3.4 Native interface

The native interface provides different methods for direct communication between the rules engine, the MLMs, and the embedding environment. Communication with other software is basically achieved by servlets or JSPs that receive requests and return the results. Such servlets and JSPs are used for the administration of the rules engine as well as to directly evoke MLMs to realize synchronous events.

Synchronous events (direct evocation)
Usually, MLMs are evoked indirectly by events. For the direct evocation, a special servlet receives the name and institution of the desired MLM by an HTTP request. Optionally, arguments that will be passed to the MLM can also be defined. The servlet uses an internal interface of the rules engine which executes the MLM and returns the result to the servlet, which finally returns it to the requesting system. The internal interface can also be used directly by Java threads that are running in the same Java process. A remote evocation of this Java interface has not been achieved so far.

Regardless of whether the MLM is called by the Java API or via the servlet, the calling thread (or process) is blocked until the MLM stops. The direct evocation can therefore be used for tasks where the continuation of the process directly depends on the result of the MLM, for example the verification of contraindications while administering drugs.

The interface statement
The Arden Syntax defines the 'interface' statement that provides direct access to institution-specific functions from an MLM. As the whole system is based on Java, it is possible to call Java methods directly from an MLM.

An interface is defined in the data slot by a mapping that includes the type of the interface (so far only the keyword 'class' is supported) by the fully qualified class name, and the name of the method name that has to be called. The method has to accept exactly one argument of 'ArrayList' type that contains an arbitrary number of arguments whose data type is one of the Arden Syntax data type Java classes.

The following example shows the use of an interface for computing an Arden Syntax date based on a textual representation of a date, such as '05.07.1975'.

Example 35: (Use of the interface statement to call Java methods)

The interface that is to be called has to be defined within the data slot:

```
stringtodate := interface
  {class;de.medstage.projects.cadiag.ardenifc.StringToDate;stringToDate };
```

It can then be used within the logic slot by passing one argument that represents a date in textual form:

```
birthdateAsDate := call stringtodate with birthdate;
```

The Java class 'StringToDate' implements the method 'stringToDate' as follows:

```
 0: public static Object stringToDate( ArrayList datestringVec ) {
 1:   try {
 2:     ArdenString dateString = (ArdenString)datestringVec.get(0);
 3:     ...
 4:     ...
 5:     ArdenTime dateAsArdenTime = new ArdenTime( dateAsDate );
 6:     dateAsArdenTime.setPrimaryTime( dateString.getPrimaryTime() );
 7:     return dateAsArdenTime;
 8:   } catch (Exception e) {
 9:     ...
10:     return null;
11:   }
12: }
```

The type definitions of the method are restricted exactly as shown in line 0, as the method is looked up dynamically and the lookup mechanism presumes these data types for result and argument. Every argument can be accessed in the ArrayList starting from position 0 (line 2). The result is computed as Arden Syntax data type (line 5) and has, in this example, the same primary time as the argument (line 6).

By making use of this interface technique, complex or performance critical operations can be out-sourced to Java methods.

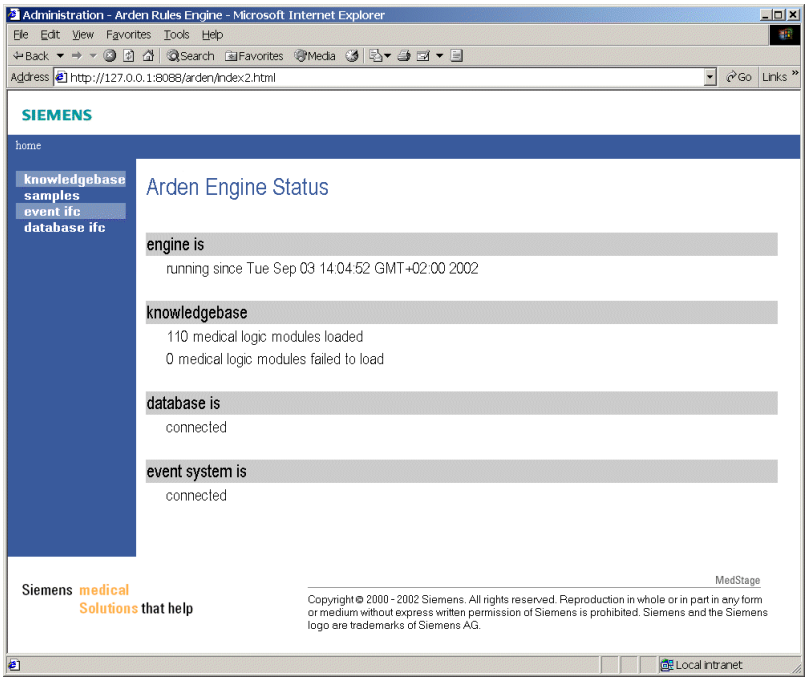

Figure 3.7: Main administration screen of the rules engine

User interface

The administration of the rules engine is entirely web-based and makes use of static HTML pages and dynamic Java servlets and JSPs. The Tomcat servlet server that embeds the rules engine is also used to host the static HTML pages.

The main screen of the engine is shown in figure 3.7. The user interface provides the main navigation bar on the left side and a smaller navigation bar on the top that indicates the current position of the user within the user interface structure.

Starting from the main screen, the user can browse through the knowledge base by clicking on the link in the main navigation bar. The knowledge base is basically structured into valid MLMs, which have been parsed without errors, and into invalid MLMs, which caused errors during the parsing process. The valid MLMs are structured by the directories in correspondence to their origin location in the file system. Figure 3.8 shows the content of one such 'directory' that contains two sample MLMs. The upper navigation bar shows the path that led to the current view.

The view on a set of MLMs provides information about the individual names and, if available, the description as defined in the corresponding slot. The MLMs can be marked by a checkbox and reloaded or removed from the knowledge base by selecting one action in the box at the bottom of the table. An MLM can be displayed in the web browser by selecting its name in the directory view. The MLM Java objects provide methods to generate an XML representation of the MLM that is converted into HTML and displayed in the browser (figure 3.9). The XML representation has been specifically designed for this conversion into HTML and does not use the approach that is currently employed in the HL7 Arden Syntax SIG[59]. In contrast to the detailed definition of the SIG that could be used in the future to check the validity of an MLM, it only defines basic structures such as

[59]The SIG defines an XML representation that can be used to validate the MLM, as when using the BNF.

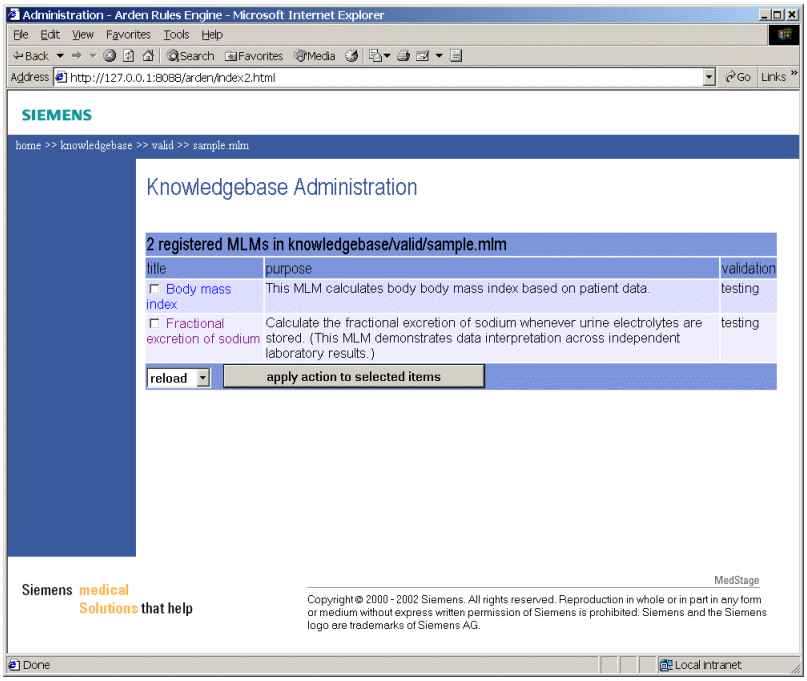

Figure 3.8: Knowledge base user interface of the rules engine

'category', 'slot', or 'operator', which are needed to distinguish single elements that have to be displayed on the screen in different style. The XML representation is transformed by the browser on-the-fly into an HTML page that uses cascading style sheets (CSS). The DTD and XSLT files can be found in appendix D. Thus, changing the appearance of the MLM in the browser does not require changes in the XML-generating methods of the Java classes, as only either the XSLT file or the CSS file has to be modified.

3.1.4 Testing

To test the implementation of the rules engine, 23 MLMs were written. These were designed to verify almost the entire functionality of Arden Syntax as defined by the specification. Only a few features were not implemented and were therefore not included in the test MLMs, such as some features of the 'formatted as' operator.

The test process is structured in two sets of MLMs, each representing one test step. The first set of MLMs has to be executed individually. It tests the functionality of assigning values to variables and retrieving them, including primary time issues. Some of the basic comparison operators as well as the list concatenation are tested. The test results are displayed as plain text and have to be checked by the user manually.

The second set uses those functionalities tested by the first step to automatize the control of the test results. The second step uses MLMs which test the functionality of all operators. Each class of operators, such as aggregation operators or transformation operators, is tested by one individual MLM. For each operator, all permutations of arguments and argument data types are tested according to the specification. The result of every single test is stored in a list. A parallel list stores the correct results (as defined by the specification) as constants. After the execution of all tests, the two lists are compared and erroneous tests results are displayed to the user.

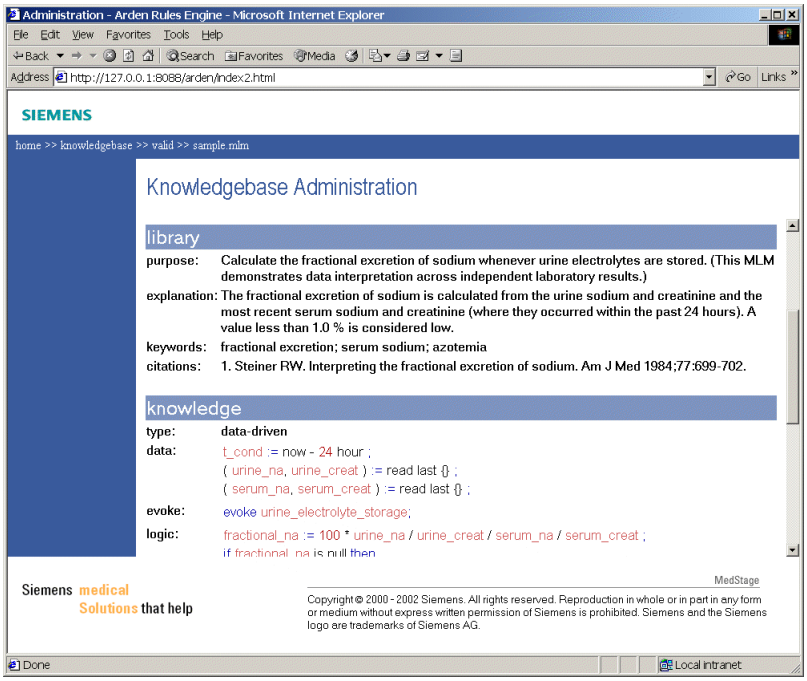

Figure 3.9: MLM representation as XML/HTML

3.2 CADIAG-II/RHEUMA$^+$Arden

The medical expert system CADIAG-II (Computer Assisted DIAGnosis) created at the University of Vienna can assist the differential diagnostic process in internal medicine by indicating all possible disease hypotheses which can explain the given symptoms from the patient data and which include frequent as well as rare disease definitions. The rheumatological part includes more than 200 disease profiles, more than 2.000 findings, more than 50.000 finding-disease-relationships, and more than 160 complex rules.

In its original table-based form the knowledge base is difficult to read and therefore difficult to maintain. As an alternative representation, a Fuzzy Arden-based version of the knowledge base was created and tested for routine applicability.

3.2.1 Introduction

The historical roots of CADIAG-II lie in the development of CADIAG in the early 70's. The first version of the knowledge base represented relationships between medical entities such as symptoms, symptom combinations, and diseases, and used a three-valued logic that was able to represent 'true', 'false', and 'unknown' entities. The relationships between the entities defined whether the presence of one entity, such as a symptom, is either obligatory or facultative for the presence of another one, for instance a disease, and whether an entity is proved by the other one or not. As CADIAG has been integrated into the hospital information system WAMIS[60] of the Vienna General Hospital, it could be used in clinical routine to propose diagnoses for patients suffering from one or more rheumatological diseases [AKL$^+$82].

[60] WAMIS is the German acronym for "Wiener Allgemeines Medizinisches Informations System" (Vienna General Medical Information System).

In the early 80s the underling logic was extended to a fuzzy logic, which additionally included two further values representing unknown facts (in the following *null* is used) and discrepant ones (in the following ω is used) [Adl80]. The inference process is based on disease profiles that consist of relationships between entities as mentioned earlier and on complex diagnostic rules that combine entities.

The relationships define, like in CADIAG, a degree (or strength) of evidence and a degree (or frequency) of occurrence between two entities, such as symptom and diagnosis. In contrast to CADIAG these degrees may be defined fuzzily; the evidence of a symptom to a diagnosis could then not only be 'true' or 'false', but take on any intermediate degree. For example, if a symptom proves a diagnosis *often*, then it would imply a reduced degree of evidence for the relationship between symptom and diagnosis. The definition of the relationships are either based on expert knowledge, which was initially represented by terms such as "always", "almost always", "very often", or "very strong", or by stochastic evaluation of patient data [AK82].

Like CADIAG, the successor has been fully included in the hospital information system WAMIS of the Vienna General Hospital [AKSG86, ALK96]. Currently the CADIAG knowledge base is to be re-implemented by integrating it into the MedFrame platform as CADIAG-IV [KAR01].

3.2.1.1 Structure of the knowledge base

The knowledge base is represented by a set of relational data base tables. They define disease definitions whose structure is outlined in figure 3.10. Every entity, such as a diagnosis, is rated during the inference process in terms of presence by using a fuzzy truth value from $F0.0$ ("not present") to $F1.0$ ("present") or can be 'null' ("unknown") or ω ("discrepancy")[61].

The **diagnoses** are the most abstract entities that are defined by relationships to other entities, such as symptoms, symptom combinations, and other diagnoses. Every relationship is based on logical implications and is defined by a *degree of evidence* and a *degree of occurrence*[62].

A relationship between two entities is defined by the antecedent, the consequent, a degree of evidence and a degree of occurrence. Such a relationship can be represented as an implication operator:

consequent :- implication(*antecedent, degree of evidence, degree of occurrence*)

This implication from one entity e_j (antecedent) to another entity e_i (consequent) with the degree of evidence b_{ji} and the degree of occurrence a_{ji} is an aggregation of two individual relations:

$$e_j \xrightarrow{b_{ji}} e_i \tag{3.1}$$

$$e_j \xleftarrow{a_{ji}} e_i \tag{3.2}$$

[61] In the following, the expressions "the symptom is rated" and "the symptom is" will be used synonymously.

[62] In the original publications, the terms "strength of confirmation" and "frequency of occurrence" are used.

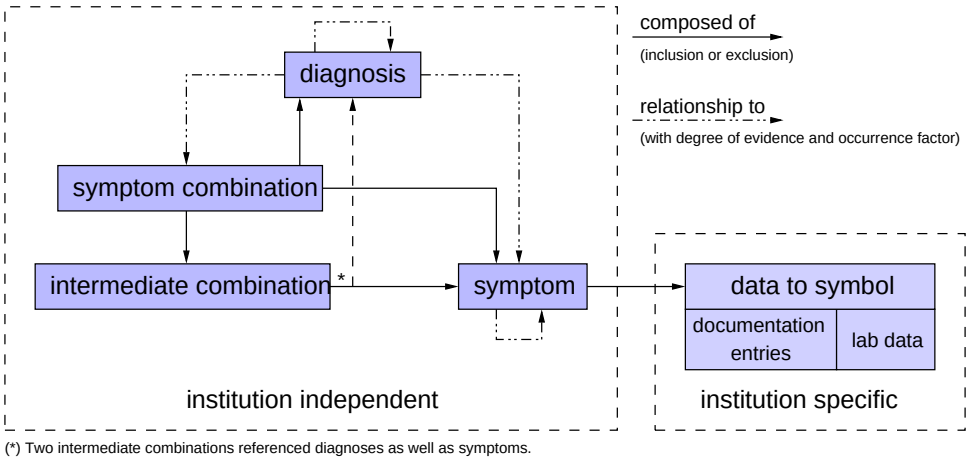

Figure 3.10: Hierarchical structure of a CADIAG-II disease definition.

As CADAIG-II is based on fuzzy logic and entities can be rated gradually, both degrees can be any value from 0.0 to 1.0. In the knowledge base the relationship between two entities is defined symmetrically—the degree of evidence of e_i to e_j is the degree of occurrence between e_j and e_i. The computation of the result mainly depends on the value of the antecedent. Basically, four different constellations are possible:

- If e_j is rated 'false' and the degree of occurrence is 1.0, then the consequent e_i can be excluded by using relation 3.2.

$$F0.0 \xleftarrow{a_{ji}=1.0} e_i \Rightarrow e_i = F0.0$$

- If the antecedent is 'true', the degree of evidence is 0.0 and the degree of occurrence is 0.0, both relations exclude the consequence:

$$\left.\begin{array}{l} F1.0 \xrightarrow{0.0} e_i \\ F1.0 \xleftarrow{0.0} e_i \end{array}\right\} \Rightarrow e_i = F0.0$$

- If the antecedent is greater than $F0.0$ then the relation 3.1 is used, as the antecedent proves the consequent:

$$F1.0 \xrightarrow{b_{ji}} e_i \Rightarrow e_i = \min(e_j, Fb_{ji})$$

- If the antecedent is either 'null' or ω, then the consequent cannot be rated and is set to 'null'. Any other constellation yields also 'null', except for discrepancies in the knowledge base that yield ω.

The implication operator that provides this functionality is defined in detail in the appendix C.1 (page 169).

Symptoms generally are based on data-to-symbol conversion rules and are additionally defined by relationships to other symptoms. The data-to-symbol conversion rules are either fuzzy sets or logical expressions that use Boolean documentation entries. A small set of symptoms is classified by crisp sets.

Symptom combinations are logical expressions that use symptoms, intermediate combinations, and diagnosis as operands. In addition to the common logical operators 'and',

'or', and 'not', two operators 'at least of' and 'at most of' are used to build the expression. **Intermediate operators** are structured like symptom combinations, but mainly include only symptoms as operands (except for two intermediate combinations which, however, include also diagnoses).

3.2.1.2 Inferencing

The inference process of CADIAG-II has been described in detail in [Fis94]. The structure shown in figure 3.11 differs slightly from the one described in that publication but describes the same process.

As—because of the symmetric definition of relationships—symptoms, diagnoses, and symptom combinations may influence each other whenever the rating of one entity is changed, some parts of the inference process are iterative.

After the initial data-to-symbol conversion, which is based on three components (numeric data converted by fuzzy sets, numeric data converted by crisp sets, and Boolean data converted by logical expressions), the symptom to symptom relationships are evaluated. This iterative step is repeated until no symptom gets a higher rating than the preceding iteration. When it has terminated, all ratings of the symptoms are fixed and will not change for the remaining inference process[63]. In addition, as described by the original creators of the CADIAG-II system, some radiological findings need further post-processing. Those findings that are still undefined ('null') after the computation of the symptoms (as described in the next step) have to be set to 'false'. The corresponding MLM is shown in appendix C.3.

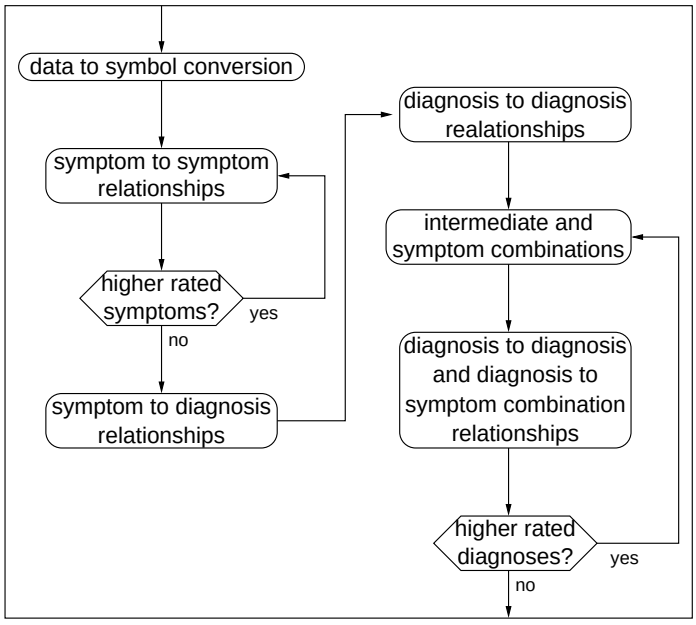

Figure 3.11: Structure of the CADIAG-II inference process

Then, in one non-iterative step, the first ratings of the diagnoses are computed by evaluating the symptom-to-diagnosis relationships. Next, all diagnosis-to-diagnosis relationships

[63] As all relations are defined symmetrically, symptom-to-diagnosis relationships have a corresponding diagnosis-to-symptom relationship. This type of relationship is not intended to recompute symptoms, but can be used to easily identify symptoms that were rated unknown but might reinforce a diagnostic hypothesis.

are evaluated iteratively until their ratings are constant. After these first ratings of the diagnoses, the intermediate combinations and symptom combinations, which partially use diagnoses in their definitions, are computed. The last (iterative) step recomputes the diagnoses by additionally including the diagnosis-to-symptom combination relationships. The iteration is repeated until all diagnoses are constant.

As every diagnosis or symptom can have more than one relationship to other entities, all single values which have been computed by the implication operator have to be aggregated to one new rating of the current entity. The 'inference operator' $\oplus$ computes this aggregation and additionally considers the recent rating of the entity in the last iteration. The new rating (in iteration $n+1$) for an entity e_i^{n+1} is computed on the basis of its old rating e_i^n on the one hand, and on the results of the relationships to other entities on the other. One important characteristic of $\oplus$ is that the new rating on an entity cannot be below the old one—the rating of a symptom or diagnosis can only be increased by additional evidences but never decreased. If the entity or any related one is rated ω the new rating is also ω. The operator is formally defined in appendix C.2.

3.2.1.3 Logical operators

As mentioned in the introduction, fuzzy set theory and fuzzy logic allow the definition of intersections ('and'), unions ('or'), and negations ('not') in multiple ways. Fuzzy Arden uses the most common methods, the 'min' operator for realizing 'and', the 'max' operator for realizing 'or', and the $1-x$ operation to compute the negation of a fuzzy truth value x (compare table 2.3 on page 63).

CADIAG-II allows the user to chose between the set of logical operators as defined earlier and an alternative set of operators, which never return unknown values (table 3.1)[64]. These operators assume that symptoms which were not examined and therefore are rated as 'null' can be interpreted as 'false'[65]. This optimistic assumption implies that unknown facts are only unknown because they either do not contribute to the result (and are therefore not needed) or because they are very likely not present as they would only be examined in case of strong suspicion. This could be the case when invasive methods such as biopsies would be required to exclude an symptom. However, this optimistic approach might fail if unknown facts are only unknown because the examining person failed to perform one or more examinations.

The reason to use modified operators can be shown, for example, on the basis of the typical structure of a symptom combination (figure 3.12). The combination consists of a set of indicative entities and a set of contradictory entities. To prove the symptom combination, all contradictive entities have to be 'false'; then the 'at least 1 of n' operator would return 'false' and the 'not' operator would invert this result yielding 'true'. If the indicative condition is rated positively, the whole expression yields 'true'.

In case all contraindicating elements are 'false' except one, which is unknown, the whole expression is rated 'null', independent of the indications. The reason is that the contraindication could not be excluded as elements were unknown and therefore the final 'and' operator would yield 'null' independent of the indications.

[64] Further, the definition of the logical operators equates unknown facts with contradictory facts (that is, $\epsilon = \omega$). Bold entries mark changes compared to the classic operator set.

[65] This method is similar to the concept of the closed world assumption where unknown facts are interpreted automatically as being 'not present'. However, as facts derived from a numerical laboratory values can still be rated unknown, the closed world assumption does not fully apply.

Table 3.1: Fuzzy logical operators truth-tables used by CADIAG-II

x_1 and x_2	true	($F0.0$, $F1.0$)	false	other
true	true	x_2	false	**false**
(F0.0,F1.0)	x_1	fuzzy and	false	**false**
false	false	false	false	false
other	**false**	**false**	false	**false**

x_1 or x_2	true	($F0.0$, $F1.0$)	false	other
true	true	true	true	true
(F0.0,F1.0)	true	fuzzy or	x_1	x_1
false	true	x_2	false	**false**
other	true	x_2	**false**	**false**

not x_1	true	($F0.0$, $F1.0$)	false	other
	false	$1 - x_1$	true	**true**

If such a missing symptom can only be determined by invasive examinations such as a biopsy, it may be justifiable to interpret an unknown symptom as 'false'. It may be assumed that an invasive examination would be ordered only if the likelihood of validating the suspicion for a symptom is rather high. If the suspicion is not enough to order the examination, the operator set used by CADIAG-II would rate the sample symptom combination as 'false', even if one or more contraindicating symptoms were unknown.

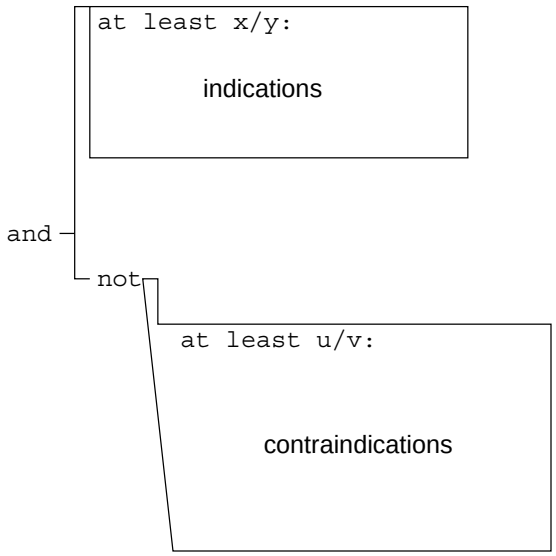

Figure 3.12: Structure of typical symptom combination

In addition to the regular logical operators CADIAG-II uses two operators, namely 'at least of' and 'at most of'. Both operators use a set with m elements as argument and include a condition n (where $n < m$) that defines how many elements have to be true at least (or at most). The 'at least' operator is 'true', if any permutation of n elements out of the set of m elements includes only 'true' elements. As the number of permutations increases fast even for rather small sets, a simple algorithm is used. The result is computed by

selecting the smallest element from the n greatest elements. To sort the lists of elements, the following order is used:

$$F1.0 > F0.0 > null$$

The at 'most of' operator is implemented as:

$$\text{at most } n \text{ of } N := \text{not (at least } n+1 \text{ of } N)$$

3.2.2 Pre-processing of the input files

Before the expert system could be represented and implemented by Fuzzy Arden, the source files had to be pre-processed for an easier conversion process.

3.2.2.1 Knowledge base files

The original table-based CADIAG-II knowledge base has been exported from the WAMIS system to ASCII files in dBase format, which were converted by Dieter Kopecky (Department of Medical Computer Sciences of the University of Vienna) to a comma-separated file format (CSV). Therefore the files were already available as pre-processed data.

From the original 17 files, six files where used to create the Arden Syntax-based representation of CADIAG-II/RHEUMA:

- WAMIS and CADIAG use different codes for the same medical entities. The necessary mappings are defined in the file 'wbstamm'. To support the creation of natural language reports, the file 'wbtext' contains German and English terms for most CADIAG codes.
- The data-to-symbol conversion layer is represented by entries in 'wbbef' and 'wbdoku'. The first file contains all numerical data-to-symbol conversions based on fuzzy sets; the second one defines all data based on binary documentation entries. Additionally, some crisp sets are defined hidden in another file 'wbdgw' whic, however, only contains the units of the fuzzy sets in textual form.
- The symptom combinations and the intermediate combinations are defined by entries in 'wbkomb' while all needed relationships are defined by entries in 'wbrel'.

3.2.2.2 Patient data files

The Fuzzy Arden representation of the knowledge base has been evaluated by a set of employed patient data consisting of 3277 cases. The data is represented by four data base tables: 'pkdoku' contains all master data for each patient. Each patient is identified by a numerical code that defines the 'Arbeitssnummer' (the correct translation would be 'work number'; the term 'patient id' is used in the following) and a numerical entry for the stay of the patient. Both units of information link all other patient data to one specific stay of one specific patient.

The clinical data is separated into the following classes:

- Documentation entries are stored in 'pdoku'. Every entry is referenced by the patient id, the patient stay, and an individual code. The entries were entered on a screen on which the clinical staff had the possibility to select or deselect single symptoms.

Therefore the values of these symptoms are binary. The data base contains only those symptoms which where selected; all non-selected are not present in the data base and will be later rated as 'false'. It is therefore not possible to mark unknown documentation entries. Some documentation entries which are numerical and used to classify the symptoms based on crisp sets are additionally stored in 'pkdoku'.

- Numerical results from the lab are stored in 'plabor'; again, each value is identified by the patient id and stay and a code.
- Some additional findings that could not be stored in the WAMIS system where stored in an additional data base named 'proe'. They include radiographic findings that need additional pre-processing.

The MedStage platform that has been used to manage the data does not define a particular data base schema. A scheme can be defined by a tool that uses an XML file containing all necessary definitions. Therefore, a data base scheme equivalent to the WAMIS system could very easily be defined by four tables 'pkdoku', 'pdoku', 'plabor', and 'proe'. The patient data were imported in a nearly identical format as the one originally used. The files were pre-processed: for example, 'pdoku' stored up to ten documentation values in one table entry. One such entry in 'pdoku'

```
67035955;1;;S03;06.06.1990;;A;202;209;215;220;227;237;242;252
```

has been converted in up to ten entries for easier and faster data base queries.

```
67035955;1;1990-06-06;S03;A;202
67035955;1;1990-06-06;S03;A;209
67035955;1;1990-06-06;S03;A;215
...
```

3.2.3 Creation of the MLMs: highly modular approach

The representation of the knowledge base by Arden Syntax MLMs was achieved by two different approaches. The first one, which is described in this section, created a very modular knowledge base where nearly every single entity, such as a symptom, a combination, or a diagnosis, was represented by one individual MLM. The second one that is described after this section created a set of nine MLMs that represented single inference steps (as shown before in figure 3.11) instead of representing each entity as an individual MLM. Both approaches additionally use a set of helper MLMs that provide an interface to the data storage system and some further functionalities, such as report formatting.

One principle of the Arden Syntax is the modularity of the knowledge base. This was one reason to create a representation of the knowledge base in which the knowledge nuggets, such as diagnoses, symptoms, or combinations, are defined by individual MLMs. The intention was to increase the overall readability of the disease definitions and thus achieve easier maintainability of the knowledge base.

At the beginning of the inference process all entities are undefined ('null'). To compute the rating of an entity it is therefore required to call those MLMs that define the other entities and to process their results. The implication operator and the inference operator that process these results were implemented by two MLMs. Their implementation follows the definition given in the introduction and is explained in detail in the appendix C. The main inference process is driven by one inference MLM which calls sequentially the MLMs that define the diagnoses.

3.2.3.1 Structure

Every diagnosis is defined by one individual MLM. Such a diagnosis MLM defines references all those MLMs that represent other diagnoses, symptoms, or combinations, which have a relationship to the current one (see figure 3.13). Analogously, a combination MLM references all MLMs that represent the single elements of the combination, and a symptom MLM references the data-to-symbol conversion MLM that basically defines the symptom and those symptom MLMs that are related to the current one.

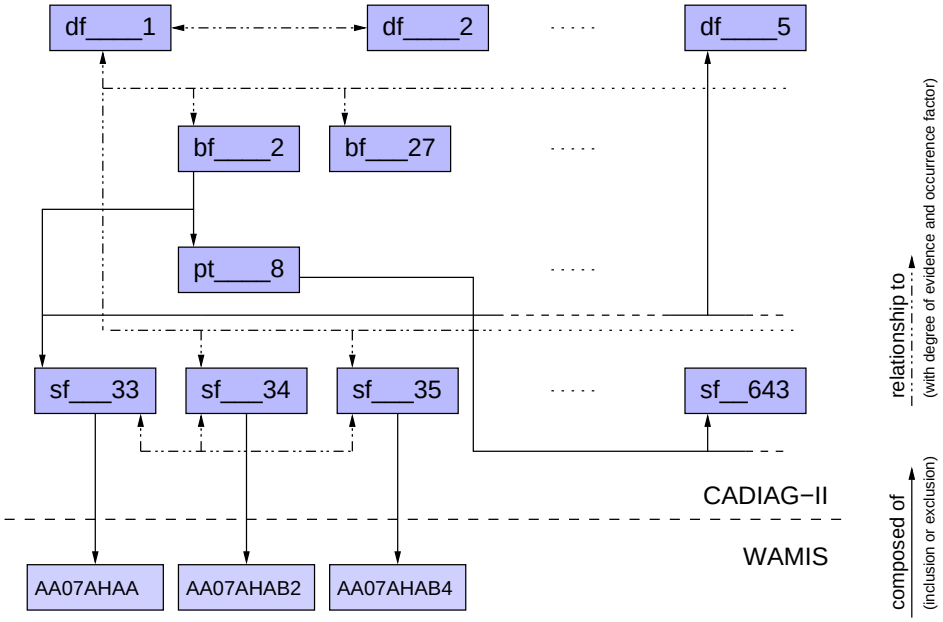

Figure 3.13: Structure of the modular CADIAG-II/Arden representation

The knowledge base is evaluated by one main MLM that calls all diagnosis MLMs, which evaluate their relationships by calling the corresponding MLMs. One issue of this inference process is the symmetric definition of relationships between entities: Whenever an entity e_i has to evaluate its relationship to an entity e_j it calls the corresponding MLM which defines the same relationship. As this MLM alone has the information to evaluate the relationship, it would call the first MLM despite the fact that it has just been called by it (figure 3.14).

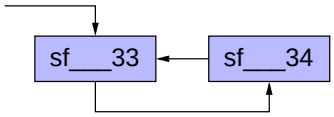

Figure 3.14: Recursiveness of the knowledge base

Such endless calls from one MLM to another one, or, more generally, calls of MLMs that already have been called once, have to be avoided in some way. This issue has been resolved by using a list that contained the names of all those MLMs that have previously been called once. Whenever an MLM call has to be executed, it is first checked whether the name of the MLM is an element of this list. If so, it is assumed that that entity has already been rated and its current rating is used instead of calling the MLM again.

Example 36: (Anti-recursion mechanism)

The first diagnosis that is called by the inference MLM is "Rheumatoid Arthritis, Unspecified" (DF 1). It includes a relationship to "Felty's Syndrome" (DF 2). First, it marks itself as already called by including its own label in the 'calledMLMs' list. During the execution of the logic it checks if the MLM 'DF___2' has already been called and calls it (as at this point the list only contains the label of DF 1).

```
calledMLMs := calledMLMs, "DF____1";
...
if "DF____2" is not in calledMLMs then
    calledMLMs := call mlmDF____2 with calledMLMs;
endif;
```

The MLM of DF 2 has defined the same relationship (with swapped degree of evidence and degree of occurrence that) and tries therefore to call the MLM of DF 1. As this MLM already is an element of the list 'calledMLMs', it is not called again.

```
calledMLMs := calledMLMs, "DF____2";
...
if "DF____1" is not in calledMLMs then
    calledMLMs := call mlmDF____1 with calledMLMs;
endif;
```

Arden Syntax defines all variables locally; one MLM does not know the variables of another. One way to get the rating of an entity could be, as shown in the last example, to return the rating as a result of the MLM call. But whenever the anti-recursion mechanism blocks an MLM call, no rating for the referenced MLM could be returned. To access the current value of an entity it is therefore required to store all ratings in intermediate variables.

3.2.3.2 Intermediate variables

As the argument/result data transfer between two MLMs cannot be used effectively because of the anti-recursion mechanism, the system needs to use 'intermediate variables' that are passed from one MLM to another and represent the rating of the entities. One implementation is to carry on two lists as arguments for every MLM call where one contains the labels of the entities and the other, the corresponding values (ratings).

Example 37: (Sharing intermediate variables as arguments)

```
 0: calledMLMs := calledMLMs, "DF____1";
 1: ...
 2: if "DF____2" is not in calledMLMs then
 3:     (calledMLMs, entityLabels, entityValues) := call mlmDF____2
 4:         with calledMLMs, entityLabels, entityValues;
 5: endif;
 6: ...
 7: /* lookup position of label in entityLabels and get
 8:    the corresponding value in entityValues */
 9: ...
10: elements := count entityLabels;
11: position := 1;
12: found := false;
13: while (found = false) and (position <= elements) do
14:     if entityLabels[position] is equal to "DF____2" then
15:         found := true;
16:     endif;
17: enddo;
18: ...
19: if found = true then
20:     DF____2 := entityValues[position];
21: else
22:     DF____2 := null;
23: endif;
```

The intermediate variables are stored in two lists that are passed when calling an MLM as argument (line 4) and returned with updated values as a result of the call (line 3). To look up one particular value the corresponding label has to be located in the list that stores the labels (lines 10 to 17). If it is found the value can be mapped to a local variable; if not it is set to 'null' (lines 19 to 23).

However, when dealing with 2000 entities that are used by more than 50000 relationships and rules, the costs of searching one specific value in the list would significantly slow down the system. On average, an implementation as shown in the last example would have to execute 1000 comparison operators before the position of the corresponding value in the other list is found[66]. As a tribute to the performance, the intermediate variables have been implemented by using the 'interface' methodology of Arden Syntax. A set of value storage MLMs can be used to store and retrieve intermediate variables. These MLMs capsule interfaces to a Java class, which stores the values by their labels more efficiently than any Arden Syntax-based solution could do.

Example 38: (Sharing intermediate variables by interfaces)

```
0: storageId := argument; ...
1: calledMLMs := calledMLMs, "DF____1";
2: ...
3: if "DF____2" is not in calledMLMs then
4:     calledMLMs := call mlmDF____2 with calledMLMs, storageId;
5: endif;
6: ...
7: DF____2 := call storage_extract with "DF____2", storageId;
```

[66] A 'for' loop that could have been used to implement the search more readably would iterate the whole list, as it does not provide any means of interrupting its execution.

In contrast to the method shown in the preceding example, the intermediate variables are not passed as argument when calling another MLM, but are referenced by a unique storage id (which was created at the beginning of the inference process) that identifies the location of all intermediate variables. The only result of such an MLM call is the list that represents those MLMs which have been previously called for the anti-recursion mechanism (line 4).

The lookup method of the last example is simply replaced by one MLM call that needs the label of the intermediate variable and the storage id (line 7).

3.2.3.3 Data base cache

As shown before in figure 3.13, the definition of a disease may be regarded as a tree with the name of the diagnoses as the root and the data-to-symbol conversion rules as leaves. Since many symptoms are used by more than one disease definition or symptom combination, the corresponding data-to-symbol conversion rules would be executed more than once. As the results of these rules are constant and a multiple execution would be redundant, all data-to-symbol conversion rules are executed once initially.

As every data-to-symbol MLM makes a query to the data base, the data-to-symbol conversion became one performance bottleneck of the inference process. Intermediate variables were used to enhance the performance of the data base access. The concept is similar to the value storage explained earlier: At the beginning of an inference process, all data of the patient are retrieved from the data base and stored by a Java class. Then, every MLM can access the stored data by calling a Java method by the interface statement. This interface is additionally hidden behind a set of MLMs that provide the functionality to access the data. An example is given later.

3.2.3.4 Inferencing

At the beginning of the inference process all entities are rated 'null'. If the rating of an entity e_i is influenced by the rating of another entity e_j then the corresponding MLM that has to compute the rating of e_j is called. For the computation the rating of e_i is also needed, as the relationship is defined symmetrically; however, as its MLM had been previously called once, the current rating 'null' of e_i is used to compute e_j.

The rating of e_j, which is returned by intermediate variables to e_i, could be incorrect as e_i could immediately change its own rating and the value 'null', which has been used to compute e_j, is no longer correct anymore.

Thus the first computations of entities might not yield the correct values as they might depend on other entities that may still be undefined. As mentioned, the entire inference process is controlled by a high-level MLM that has therefore to call iteratively every diagnosis MLM (except for those that have been previously called by other diagnosis MLMs), until the values of all diagnoses are constant.

In the second iteration e_j^2 can use the rating of e_i^1 to compute its rating, and e_i^2 can therefore consider this new rating.

In summary, the modular knowledge base worked, but the quantity of MLM calls proved a significant bottleneck. Despite all improvements by Java classes one iteration lasted more than one minute; therefore this approach was discarded. The purpose of the second approach was to reduce the time needed for the complete inference.

3.2.4 Creation of the MLMs: compact knowledge base

In contrast to the highly modular knowledge base, the compact one comprises only nine MLMs for the inference process. In addition, the helper MLMs that define the operators and provide the intermediate variables and data base cache are used in the same way as described earlier.

The entire inference process is controlled by one MLM that initializes the data base cache and the value storage and calls the single inference MLMs (figure 3.15). It starts with the data-to-symbol conversion which includes three MLMs. Next, the symptoms are computed by one iterative MLM. Then the diagnoses are rated by the symptoms, and the final ratings are computed iteratively by three MLMs. The inference process is concluded by an MLM that computes some additional scores which can optionally be used to sort the results.

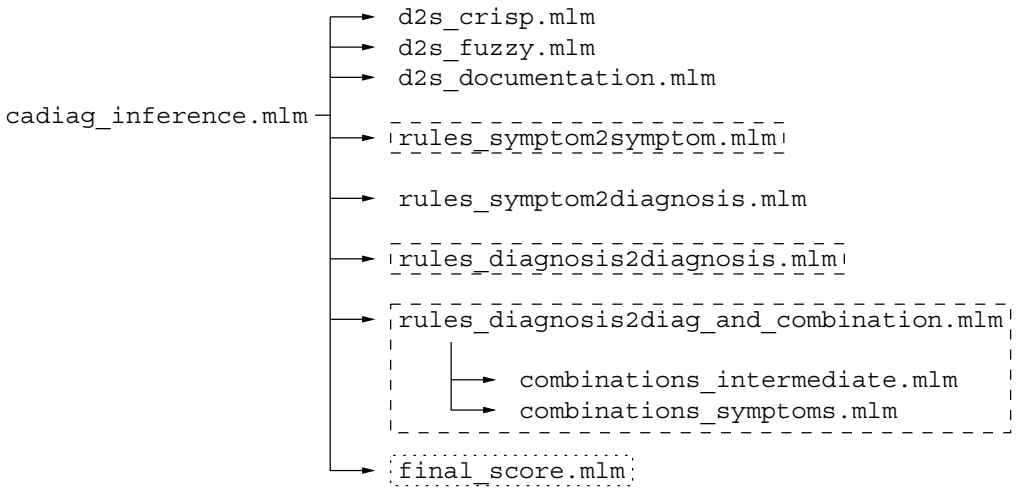

Figure 3.15: Inference MLMs used by the compact knowledge base

3.2.4.1 Data-to-symbol conversion

During the data-to-symptom conversion the Arden Syntax knowledge base still uses WAMIS codes for the findings instead of CADIAG codes. Basically, the data-to-symptom conversion is executed by two MLMs that implement the fuzzy classification of symptoms and the determination of symptoms based on Boolean documentation entries. Additionally, 16 rules are defined in a separate MLM to classify symptoms based on crisp sets[67].

Example 39: (Crisp classification of symptoms)

```
0: /**
1:  * symptom C05BP1 (RHEUMATIC-LATEX-TEST, NEGATIVE)
2:  * is set to true, if
3:  * B0834,N07 is 1: NEGATIVE
4:  **/
5: classification := (B0834_N07 = 1);
6: labels         := labels, "AA07C05BP1";
7: results        := results, classification;
```

[67]Strictly speaking, as shown in example 39, the crisp classified symptoms are already represented by symbols. The numerical entries are codes for certain ranges of the test result which have been selected by using a form. For example, a rheumatic-latex test result of 2 means that it is slightly elevated (+).

```
 8:
 9: /**
10:  * symptom C05BP2 (RHEUMATIC-LATEX-TEST, POSITIVE)
11:  * is set to true, if
12:  * B0834,N07 is 2: +
13:  * B0834,N07 is 3: ++
14:  * B0834,N07 is 4: +++
15:  **/
16: classification := (B0834_N07 = 2) or (B0834_N07 = 3) or (B0834_N07 = 4);
17: labels         := labels, "AA07C05BP2";
18: results        := results, classification;
```

The first ratings of symptoms based on crisp classifications are computed by checking whether the pre-classified findings are within a certain range (lines 5 and 16). The resulting truth value is stored as intermediate variable by adding the value and a CADIAG finding code to the corresponding lists (lines 6, 7 and 17,18).

Each symptom that is based on a fuzzy classification is defined as in the following example. It shows the data-to-symbol conversion for two symptoms, which basically are two characteristics of the same concept[68].

Example 40: (Classification of symptoms by fuzzy sets)

```
 0: /* compute value for symptom AA07C01BU2
 1:    (GPT, SERUM, INCREASED) */
 2: if male = true then
 3:     resB0832_N03 := B0832_N03 >= 24.0 fuzzified by 2.0;
 4: else
 5:     resB0832_N03 := B0832_N03 >= 19.0 fuzzified by 2.0;
 6: endif;
 7:
 8: results := results, resB0832_N03;
 9: labels  := labels, "AA07C01BU2";
10:
11: /* compute value for symptom AA07C01BU3
12:    (GPT, SERUM, SIGNIFICANTLY INCREASED) */
13: resB0832_N03 := B0832_N03 >= 270.0 fuzzified by 40.0;
14:
15: results := results, resB0832_N03;
16: labels  := labels, "AA07C01BU3";
```

The first characteristic 'increased' depends on the gender of the patient (line 2), the second does not[69]. The result of the fuzzified comparison (lines 3, 5, and 13) is either a fuzzy truth value or 'null', if the value (identified by the data base code 'B0832_N03' in this example) was not found in the set of patient data. The result of the classification is stored as an intermediate variable; both characteristics use their own CADIAG finding code.

Fuzzy classifications of symptoms may additionally be age dependent, as shown in the following example. The variable 'age' represents the age of the patients at the moment

[68] Such a concept could also be represented by one "knowledge nugget" as linguistic variables that aggregates characteristics of one concept. However, as in the original system, such characteristics are represented by separate codes.

[69] The variable 'male' is assigned at the beginning of the inference process.

of measurement of the finding. The age is determined at the beginning of the inference process, based on the day of admission of the patient's current stay.

Example 41: (Age dependent classification of symptoms)

```
 0: /* compute value for symptom AA07BAUBBA
 1:    (EXTREMITIES, RIGHT HAND, LOSS OF FORCE) */
 2: /* fuzzy sets are age dependent (crisp) */
 3: if age is within 16.0 to 65.0 then
 4:     resS04_511 := S04_511 <= 6.0 fuzzified by 1.0;
 5: elseif age is within 0.0 to 15.0 then
 6:     resS04_511 := S04_511 <= 5.0 fuzzified by 1.0;
 7: elseif age is within 66.0 to 120.0 then
 8:     resS04_511 := S04_511 <= 5.0 fuzzified by 1.0;
 9: endif;
10:
11: results := results, resS04_511;
12: labels  := labels, "AA07BAUBBA";
```

So far the age comparison is defined crisply (as it has been in the original knowledge base) (lines 3, 5, and 7), whereas the data to symbol conversion is fuzzy (lines 4, 6, and 8).

Symptoms based on documentation entries are either 'true' or 'false'. Their data to symbol conversion rules are based on the existence of entries in the data base. Generally, if an entry exists the symptom is rated 'true', else it is rated 'false'. This classification can be easily implemented by the 'exists' operator. Most definitions are based on only one entry, some are based on more than one (as shown in the next example), and few are based on large logical constructs.

Example 42: (Crisp classification of symptoms based on document entries)

```
0: /* SKIN AND MUCOUS MEMBRANES, REDDENED */
1: result :=
2:     (
3:         (exists S03314)
4:     and
5:         (exists S03344)
6:     );
7:
8: results := results, result;
9: labels := labels, "AA07BABEM3";
```

Such findings cannot be 'null', as the 'exists' operator returns 'false' if its argument is 'null' (lines 3, 5).

3.2.4.2 Computation of the symptoms

After the data-to-symbol conversion of the findings, the ratings for the symptoms are computed by one MLM in three steps. The current ratings are read from the value storage and assigned to local variables whose labels are equal to the CADIAG symptom codes.

Example 43: (rules_symptom2symptom.mlm: code fragment of the data slot (I))

```
0: /* get initial values for symbols defined by data2symbol conversion MLM */
1: labels := "AA07C75KN12", "AA07BAUAK", "AA07BAUET4", "AA07BAUET3", ...
2:
3: /* get values from value storage */
4: resList := call valueExtract with valuesId, labels;
5:
6: /* assign to local variables */
7: SF_1000 := resList[1]; SF_1012 := resList[2]; SF_1035 := resList[3];
8: SF_1049 := resList[4]; ...
```

The value storage access is prepared by defining a list of labels (line 1) and done by calling the helper MLM (line 4). Then the results are assigned to local variables (lines 7, 8)[70].

Instead of mapping the results to local variables the values could be accessed directly from the list that is returned by the value storage MLM. Looking at the last example, instead of accessing variable 'SF_1049' it would be possible to access the value by using `resList[4]`. However, the slight benefit of performance does not justify the worsened readability if the symptom code variables were replaced by abstract list positions.

Some CADIAG symptoms are based on data that are not stored in WAMIS and have therefore no WAMIS code. These symptoms are defined by external Boolean data, as shown in the next code fragment.

Example 44: (rules_symptom2symptom.mlm: code fragment of the data slot (II))

```
0: /* get initial values for symbols defined by PROE database */
1: dbCode := "KRM", "BATTB", "BAUTBD", "FWDBA", ...
2:
3: /* get values from data base cache */
4: resList := call db_access with valuesId, dbCode;
5:
6: /* assign to local variables */
7: SF_1001 := resList[1]; SF_1011 := resList[2]; SF_1013 := resList[3];
8: SF_1014 := resList[4];...
```

The structure is similar to the one used for WAMIS symptoms as shown in the previous example. 409 symptoms are based on external data.

As mentioned earlier, some radiological symptoms that are unknown at this point of the inference process are separately set to 'false' (compare appendix C.3).

At this point all symptoms are pre-rated by the result of the data-to-symbol conversion or by the existence of additional data. In addition, 358 symptoms have relationships to other symptoms. As explained in the first modular approach for the relationships between entities, these relationships are evaluated by using the implication and inference operators iteratively until all symptoms are constant.

[70] Three dots in a line indicate further elements or assignments that have been removed from the example as the complete list of 717 symptoms, which are based on the data-to-symbol conversion, would have been to large to be printed.

Example 45: (rules_symptom2symptom.mlm: code fragment of the logic slot)

```
 0: changes := true;
 1: firstIteration := true;
 2: iterations := 0;
 3:
 4: symptomsToRecompute := ();
 5: symptomsAffected := ();
 6:
 7: while changes = true do
 8:     /* EXTREMITIES, HAND, GOUT TOPHUS AT SEVERAL FINGERS (SF    1)*/
 9:     /* if this Code is set to omega, do not apply more rules */
10:
11:     is_not_omega := not((SF____1 is string) and (SF____1 <> "omega"));
12:     if (firstIteration = true
13:       or ("SF____1" is in symptomsToRecompute and is_not_omega = true)) then
14:
15:         /* relationship with other symptoms: eval all occur/evidence factors */
16:         impValue      := SF__474;
17:         impEvidence   := null;
18:         impOccurrence := fuzzy 1.0;
19:         impResults    := call imp with impValue, impEvidence, impOccurrence;
20:
21:         /* now compute result with scalar inference operator */
22:         entitylist := SF____1, impResults;
23:         newVal := call iop with entitylist;
24:         both_not_null := not((newVal is null) and (SF____1 is null));
25          both_equal := newVal = SF____1;
26          if (both_not_null = true) and (both_equal is null or both_equal = false) then
27:             SF____1 := newVal;
28:             symptomsAffected := symptomsAffected, "SF__474";
29:         endif;
30:
31:     endif;
32:
33:     ...
34:
35:     firstIteration := false;
36:     if ((count symptomsAffected) = 0) then changes := false; endif;
37:     symptomsToRecompute := symptomsAffected;
38:     symptomsAffected := ();
39:     iterations := iterations + 1;
40: enddo;
```

During the first iteration every relationship is evaluated (line 12). For each relationship the value of the entity (line 16), the degree of evidence (line 17), and the degree of occurrence (line 18) are stored in three lists that are passed as argument to the MLM which implements the implication operator (line 19). The result is a list that contains the ratings of all relationships to other symptoms. Together with the current value for the symptom, they are used by the inference operator-MLM to compute a new value for the current symptom (lines 22, 23).

If the rating of the current symptom has changed all related symptoms have to be recomputed in the next iteration, as all relationships are defined symmetrically (lines 24 to 29). Their codes are stored in the list 'symptomsAffected'

(line 28). After each iteration the symptoms stored in the list 'symptomsAffected' are copied in the list 'symptomsToRecompute' which is used in the next iteration as an additional condition for the computation of a symptom (line 13). Further, if a symptom is rated ω it does not need to be recomputed (lines 11, 13). As Arden Syntax does not bind variables to a certain data type, the representation of this additional rating could easily be realized by assigning the string "omega" to the variable.

After this step all symptoms are stored as intermediate variables by their CADIAG code in the value storage.

3.2.4.3 Computation of the diagnoses

The first non iterative step for computing the diagnoses is the evaluation of their relationships to the symptoms, represented as individual MLM. As the symptoms cannot change anymore this step does not include any cross-influences by relationships that would require an iterative computation.

Example 46: (rules_symptom2diagnosis.mlm: cutout of the logic slot)

```
 0: /* CHRONIC POLYARTHRITIS (DF    1)*/
 1:
 2: /* relationship with symptoms: process all occur/evidence factors */
 3: impValue := SF___34, SF___35, SF___36, SF___37, SF___38, SF___39, SF___43,
 4:             SF___48, SF___50, SF___53, SF___54, ...
 5: impEvidence := fuzzy 0.1, fuzzy 0.05, fuzzy 0.05, fuzzy 0.1, fuzzy 0.1,
 6:                fuzzy 0.05, fuzzy 0.05, fuzzy 0.05, fuzzy 0.05, fuzzy 0.05,
 7:                fuzzy 0.05, ...
 8: impOccurrence := fuzzy 0.07, fuzzy 0.4, fuzzy 0.39, fuzzy 0.2, fuzzy 0.46,
 9:                  fuzzy 0.05, fuzzy 0.08, fuzzy 0.05, fuzzy 0.39, fuzzy 0.04,
10:                  fuzzy 0.12, ...
11:
12: impResults := call imp with impValue, impEvidence, impOccurrence;
13:
14: /* now compute result with scalar inference operator */
15: entitylist := impResults[1], impResults[2], impResults[3], impResults[4],
16:               impResults[5], impResults[6], impResults[7], ...
17:
18: newVal := call iop with entitylist;
19:
20: /* store result (lists used later by storage MLM) */
21: diagnoses_labels := diagnoses_labels, "DF____1";
22: diagnoses_values := diagnoses_values, newVal;
23:
24: /* prepare score of the diagnosis */
25: counter := 1; score := 0;
26: for anEntity in impValue do
27:   if anEntity is not null then
28:     if impEvidence[counter] is null then impE := fuzzy 0;
29      else impE := impEvidence[counter]; endif;
30:     if impOccurrence[counter] is null then impO := fuzzy 0;
31      else impO := impOccurrence[counter]; endif;
32:     score := score
33               + (score_weight_evidence   * ((minimum (anEntity, impE)) as number))
```

```
34:              + (score_weight_occurrence * ((minimum (anEntity, impO)) as number));
35:   endif;
36:   counter := counter + 1;
37: enddo;
38: diagnoses_score_labels := diagnoses_score_labels, "SCR_DF____1";
39: diagnoses_score_values := diagnoses_score_values, score;
```

335 relationships to symptoms are defined for the diagnosis 'chronic polyarthritis'. Analogous to the relationships between symptoms, all required information for the implication operator is stored in three lists and passed to the implication MLM (lines 3 to 12). The result is aggregated by the inference operator MLM (lines 14 to 18).

Additionally, the scoring of the results is prepared (lines 25 to 37). The scoring mechanism (which is defined by a weighted sum of the entities, their degree of evidence, and their degree of occurrence) is defined in detail later.

The relationships between diagnoses are evaluated by the same iterative process as the relationships between symptoms, which is therefore not explicitly shown here.

The final iteration additionally includes the evaluation of intermediate and symptom combinations. These combinations are structured like the data-to-symbol conversion rules for documentation entries, i.e. by logical expressions that use 'and', 'or', 'not', 'at least', and 'at most' operators.

Example 47: (Symptom combination: chronic polyarthritis)

```
 0: /* REITER-SYNDROME (B=0.90) (BF___15) */
 1:   is_not_omega := not((BF___15 is string) and (BF___15 <> "omega"));
 2:   if is_not_omega = true then
 3:     newVal :=
 4:     (
 5:         (
 6:         not
 7:             at least 1 from (
 8:                 SF_2166    /* EXTREMITIES, AFFECTION OF FINGER IN RAY */,
 9:                 SF__234    /* SKIN, ERYTHEMA NODOSUM */,
10:                 PT___36    /* MUCOCUTANEOUS APHTHAE */,
11:                 PT___35    /* IRITIS OR IRIDOCYCLITIS */,
12:                 PT___34    /* CHRONIC DIARRHEA WITH LOSS OF BLOOD OR
13:                               DISCHARGE OF MUCUS */,
14:                 PT___33    /* PROOFED PSORIASIS */
15:             )
16:         )
17:     and
18:         at least 3 from (
19:             PT___32        /* BALANITIS OR KERATODERMIA */
20:             SF___68        /* CURRENT COMPLAINTS, STOOL, DIARRHEA */,
21:             SF__955        /* CURRENT COMPLAINTS, URETHRITIS */,
22:             PT___31        /* CONJUNCTIVITIS */,
23:             PT___30        /* PAIN AND SWELLING OF AT LEAST ONE JOINT */
24:         )
25:     );
26:   both_not_null := not((newVal is null) and (BF___15 is null));
27:   both_equal := newVal = BF___15;
28:   if (both_not_null = true) and (both_equal is null or both_equal = false) then
29:         BF___15 := newVal;
```

```
30:         diagnosesAffected := diagnosesAffected, "DF___10";
31:     endif;
32: endif;
```

Because of the stack-based conversion of the knowledge base, the indicating entities are moved to the second half of the expression while the contraindicating entities are moved to the first half (compare figure 3.12). The expression is only evaluated if the combination has not been rated contradictory before (line 1).

If any diagnosis has a relationship to a combination that has changed its rating (line 26 to 28), it is marked as an affected diagnosis for the next iteration (line 30). The list of affected diagnoses is returned as a result of the MLM to the iterating MLM.

After the last iteration the inference process is terminated and the final scores of the diagnoses are computed.

3.2.4.4 Scoring

As the result of one inference process may include many diagnoses with the same rating or with low ratings, such as $F0.1$, a scoring mechanism that defines a sort criterion to display the "best" diagnoses first was implemented. This mechanism is based on the ratings of related entities and their degree of evidence and occurrence. As the diagnoses have initially been scored on the basis of the symptoms, the final scores have only to consider the relationships to diagnoses and symptom combinations. This scoring mechanism is well described in [Fis94].

The score of an entity e_j is the sum of the influence of all related entities e_i in terms of evidence and occurrence. The influence of one entity e_i to e_j is defined by the rating and the degree of evidence

$$H^e_{ij} = \min(e_i, \mathrm{evd}_{ij})$$

and by the rating and the degree of occurrence

$$H^o_{ij} = \min(e_i, \mathrm{occ}_{ij})$$

As the 'min' operator is used, entities that are rated 'false' have no influence on the score. The higher the entities are rated and the higher the degree of evidence and the degree of occurrence are, the higher the entity e_j gets scored.

Additionally, two weights are used to control the influence of evidence (α) and occurrence (β) on the score. The overall score of an entity e_j is defined by

$$P_j = \sum_{i=1}^{n_j} (\alpha \min(e_i, \mathrm{evd}_{ij}) + \beta \min(e_i, \mathrm{occ}_{ij})) \tag{3.3}$$

where n_j are the indices for all symptoms, diagnoses, and symptom combinations related to the entity e_j.

3.2.5 Report generation

All results are represented by an XML file. The generation of the XML structure is done for performance reasons by an interface that has been implemented in Java. Figure 3.16 sketches the structure of the result; the DTD is defined in appendix C.5.1.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="http://127.0.0.1:8088/cadiag/CadiagResults.xsl"?>
<!DOCTYPE MLM SYSTEM "http://127.0.0.1:8088/cadiag/CadiagResults.dtd">
<CADIAG id="67035955">
    <DIAGNOSES>
        <TRUE>
            <ENTRY>
                <LABEL code="AA0871400" cadiag="DF____1">
                    <TEXT TOPIC="">CHRONISCHE POLYARTHRITIS</TEXT>
                </LABEL>
                <VALUES score="877.3">
                    <VALUE>0.90</VALUE>
                    <VALUE>0.80</VALUE>
                </VALUES>
             </ENTRY>
             ...
        </TRUE>
        <FALSE>...</FALSE>
        <NULL>...</NULL>
        <OMEGA>...</OMEGA>
    </DIAGNOSES>
    <SYMPTOM_COMB> ... </SYMPTOM_COMB>
    <INTERM_COMB> ... </INTERM_COMB>
    <SYMPTOMS> ... </SYMPTOMS>
</CADIAG>
```

Figure 3.16: Structure of the CADIAG XML result file

The XML file is structured into four categories: diagnoses, symptom combinations, intermediate combinations, and symptoms. Every category is structured into four parts: hypothesis ("true" entities with truth value greater than $F0.0$), excluded entities (truth value equal to $F0.0$), unknown entities ('null'), and errors ('omega').

Every part contains entries that are composed of the textual label, the value, and the score. Entries of hypothesis may contain more than one value if their value did change during the inference process; the first value in the list is the last computed result. The label is additionally separated into a fixed "topic" that can be constant if entities belong to one thematic group (for example, symptoms that belong to a group of symptoms). The topic is used to facilitate conversion into a clearly readable output format.

This XML file contains the values of *all* entities and can be processed before being displayed. For example, unknown entities may be removed and the diagnoses may be filtered in order to display only those that have a high value and a high rating. A simple way to display the XML file is to use a XSLT stylesheet that converts the XML file to HTML[71]. A sample output is shown in figures 3.17 and 3.18, the used XSLT stylesheet is shown in appendix C.5.2.

[71] XSLT is a specialized language for transforming XML documents.

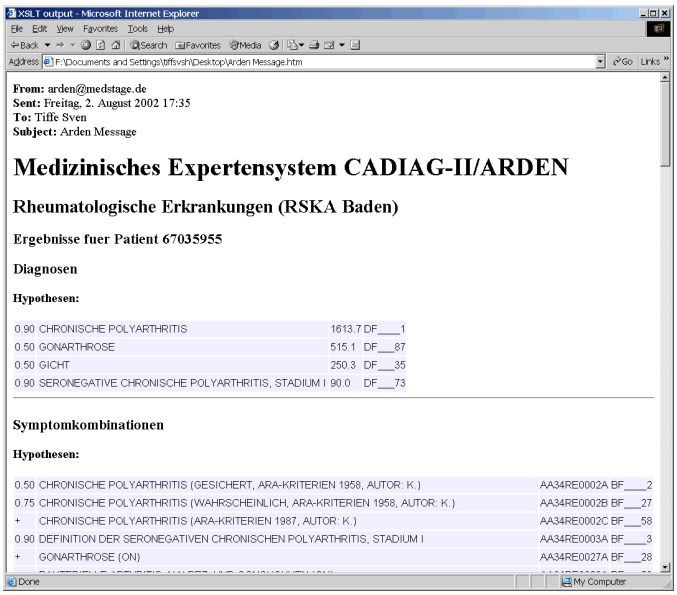

Figure 3.17: Cadiag sample result displayed as HTML e-mail (1)

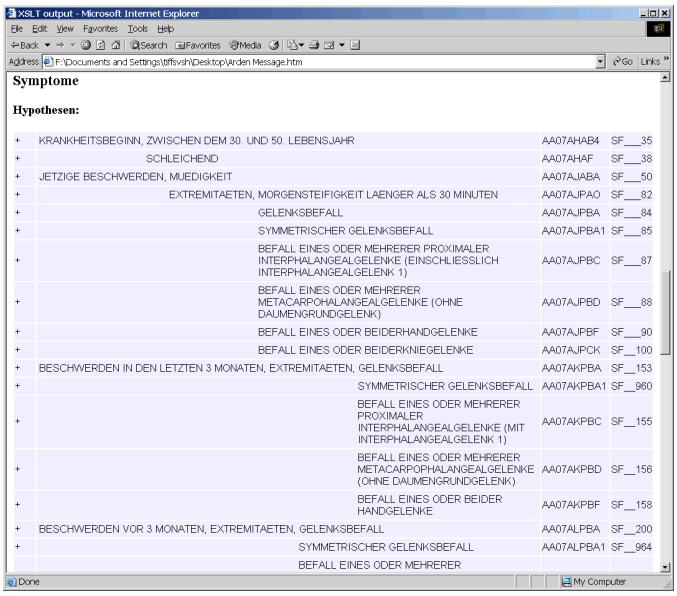

Figure 3.18: Cadiag sample result displayed as HTML e-mail (2)

3.2.6 Helper MLMs and interfaces

Both approaches to represent the CADIAG-II knowledge base by Fuzzy Arden make use of helper MLMs and Java interfaces. The implication operator and the inference operator have been implemented as individual MLMs. They have been described earlier, their definition and Arden Syntax implementation can be found in appendix C. The data base cache and the value storage system are both realized by a set of MLMs used to evoke Java interfaces.

Data base cache

The data base access was realized by using a set of MLMs that read all patient data into a cache and provide a method to retrieve individual values from it (figure 3.19). Initially, four MLMs read the data from the data base tables 'pdoku', 'pkdoku', 'plabor', and 'proe'. The data base queries, which are implemented by read statements that use a SQL-like syntax within the curly braces, read all data related to the current patient. After the data have been retrieved from the data base they are stored in the value storage system that is implemented by a Java class and uses internally Java HashMaps and Java ArrayLists to store the data.

Two MLMs provide access to the stored data, one for numerical lab values and the other for documentation entries (both are represented by 'get data base entry' in the figure). Internally both MLMs use the same Java interface and the same data storage. The difference between them is that absent values in the documentation entry data base are rated as 'false' while absent values in other data bases are rated as unknown ('null')[72]. Both MLMs are used by the data-to-symbol conversion MLMs only.

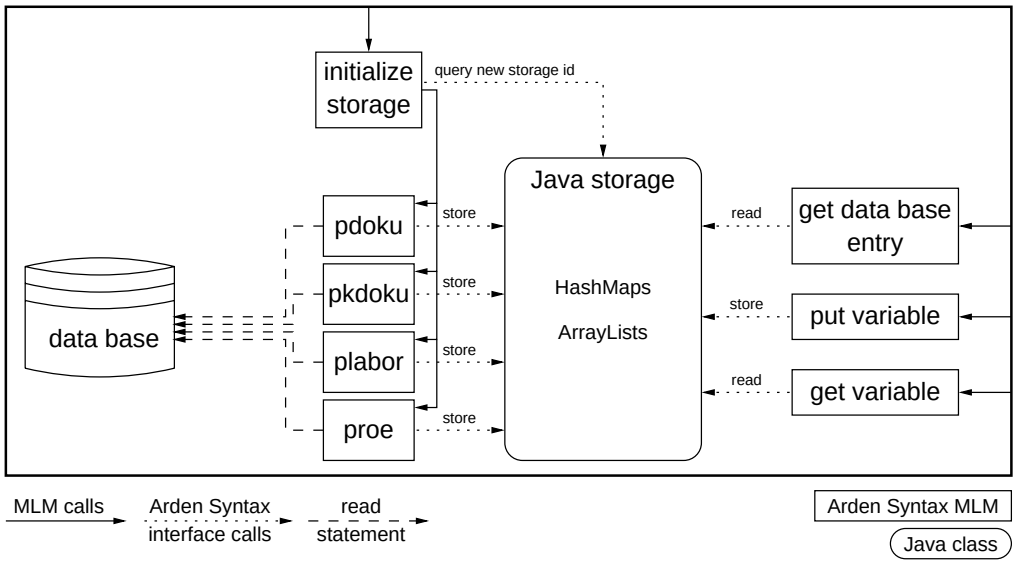

Figure 3.19: Structure of the intermediate variable storage

Value storage

The value storage is initialized at the beginning of the inference process by evoking a Java method that instantiates a new data structure within the Java class and returns a unique id to the MLM. This id is needed for all operations on the storage and assures that parallel inference processes have access only to the data of their patient. The initialization MLM also calls those four MLMs that initialize the patient data, which have been described earlier, as they need the id as well.

Intermediate variables—all data needed in more than one MLM—have to be stored at the end of an MLM that altered shared variables and have to be read at the beginning of the next MLM that needs the current value of such variables. Two MLMs provide these functions ('put variable' and 'get variable' in figure 3.19); they can be evoked either with

[72]The documentation entry data-to-symbol conversion rules make use of the 'is present' operator that returns automatically 'false' for unknown entities. Therefore the separation is due to technical reasons and is kept only to point out the difference.

one label (and the value of the variable if it has to be stored) or with a list of labels (and a list of values); the latter accelerates the performance of the system. The MLMs are called in a similar way to the data base cache MLM as shown in recent examples 43 and 44.

3.2.7 New operators

Two new operators 'at least' and 'at most' were needed to implement the logical combinations. As the usability of these two operators is not restricted to the CADIAG-II knowledge base, they have been implemented as native Arden Syntax operators. The addition of these two new operators to the aggregation operators of the Arden Syntax is proposed.

3.2.7.1 at least of (binary)

The 'at least of' operator checks whether at least n of m elements of a list with truth values are true. The list should consist of truth values, however other values can be contained, too. The list is sorted descending, where 'true' is greater than 'false' is greater than any other data type. If the first n elements are 'true', then 'true' is returned. If the n^{th} element is a fuzzy truth value or 'false', then it is returned as result. If this element is 'null' or of any other data type, then 'null' is returned. If n is a fractional number, the lower integer is used[73].

<1:fuzzy> := **at least** <1:number> **of** <n:any-type>

Its usage is:
```
true   := at least 2 of (true, false, null, true);
false  := at least 3 of (true, false, null, true);
null   := at least 3 of (true, null, null, true);
F0.5   := at least 2 of (true, fuzzy 0.5, fuzzy 0.2, false);
F0.2   := at least 3 of (true, fuzzy 0.5, fuzzy 0.2, false);
false  := at least 2 of (true, fuzzy 0.5, fuzzy 0.2, false);
```

3.2.7.2 at most of (binary)

The 'at most of' operator checks whether at most n of m elements of a list with truth values are true. The list should consist of truth values, however other values may also be contained. The list is sorted in descending order, where 'true' is greater than 'false' is greater than any other data type. The at most of is implemented as 'not (at least $n+1$ of...)'.

<1:fuzzy> := **at most** <1:number> **of** <n:any-type>

Its use is:
```
false  := at most 2 of (true, false, null, true);
true   := at most 3 of (true, false, null, true);
null   := at most 3 of (true, null, null, true);
```

[73] Currently, the use of fractional numbers as arguments of operators that expect integer numbers is not defined by the specification of the syntax. This is a working item of the HL7 Arden Syntax SIG, where a general solution of this problem is being worked out.

```
F0.5  := at most 2 of (true, fuzzy 0.5, fuzzy 0.2, false);
F0.8  := at most 3 of (true, fuzzy 0.5, fuzzy 0.2, false);
true  := at most 2 of (true, fuzzy 0.5, fuzzy 0.2, false);
```

3.2.8 Acknowledgements

In addition to the acknowledgements at the beginning of this work, special thanks go to Dieter Kopecky for his commitment, assistance, and advice in understanding CADIAG-II. His preparatory work on the internals of the original representation of the knowledge base was essential for realizing the Arden Syntax representation of this knowledge base.

3.3 Glaucoma monitoring

The second project that used the fuzzy extensions of Arden Syntax was a glaucoma classifier that was part of a telemedicine project.

Monitoring glaucoma-related changes within the eye status of a patient requires the following steps: To decide whether the patient's ophthalmic data set is indicative of critical or suspicious situations by way of differential diagnosis, and to detect changes in the status over time.

In recent work, a knowledge base founded on an artificial neuronal network and a set of fuzzy control rules was defined in order to aid in differential diagnosis in this setting [ZSW97]. The fuzzy control rules were implemented using Fuzzy Arden.

3.3.1 Introduction

According to the World Health Organization, glaucoma is one of the three major causes for blindness worldwide [Who98]. Early detection of glaucomatous changes in the eye status may help in the prevention of a significant risk factor. This purpose can be achieved by measuring the ophthalmic parameters of a patient, which are monitored by an expert system that classifies the data sets and generates alerts when registering data that indicate a suspicious or critical status of the eye.

One means of classifying the eye status of a patient is simply to monitor the intraocular pressure (IOP), which is frequently elevated in patients with glaucoma, as the ability of the eye to drain the intraocular fluid is reduced. Whenever the current IOP value exceeds a threshold value (the normal eye maintains an internal pressure of 12 to 22 mm of mercury), such a system would issue a warning. However, a more sophisticated classifier would include additional parameters in the classification process as medical experts do.

3.3.1.1 Structure of the classifier

The knowledge base for such an extended classifier was defined on the basis of an artificial neuronal network (ANN) and a fuzzy rule set (FRS), as illustrated in figure 3.20. The ANNs generate a classification of perimetry data sets, which is then used as one input source for the FRS during the monitoring process.
Perimetry data describe the status of the visual field of the patient and are measured by perimeter devices that detect the loss of light sensitivity at various stimulus points of the

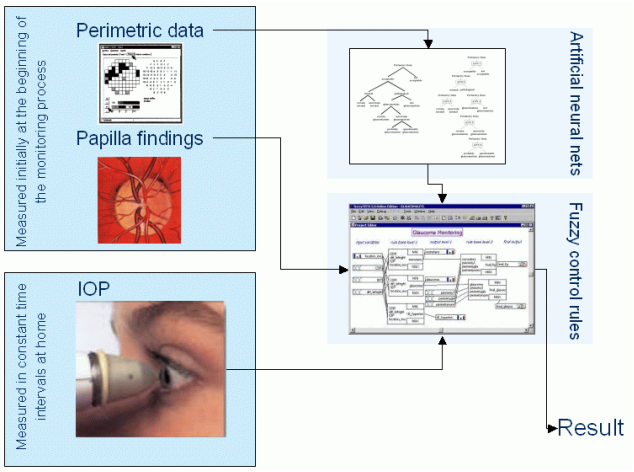

Figure 3.20: Structure of the classifier

retina. As the patient presses a response button to report whether or not he sees a stimulus, the results are subjective and generally uncertain. The ANNs that classify perimetry data were trained with training sets defined by four medical experts. The classification ranges from 'normal' or 'pathological' (but not glaucomatous) to 'questionable' or 'probably' glaucomatous.

Further input values for the FRS are estimations of papilla description parameters. These are measured by observing the papilla during an eye examination, and consist of the cup-disc ratio (CDR), the location of the excavation, and the difference between the CDR values of the two eyes. Whereas the CDR and the difference are given as real numbers, the location is given in linguistic terms such as 'central', 'inferior', or 'superior'.

The last parameter that is used as input for the FRS is the intraocular pressure. This value is measured in constant time ranges, for example daily or weekly. The remaining parameters are measured once or twice a year at the beginning of a monitoring period and are stored in the data base. Thus perimetric classification by the ANNs is done after each new measurement, whereas the fuzzy rules have to be evaluated each time the IOP is measured.

3.3.1.2 Technical structure of the monitoring application

The Arden Syntax knowledge base is used by the Java-based Arden Syntax rules engine that is connected to the Siemens MedStage communication platform as described in chapter 3.1.

Patients participating in the monitoring program are first examined by a medical expert who transmits the observed perimetric values and papilla observations via the internet to the MedStage data base. Then the patients have to measure their IOP at home using a tonometric device with an attached communication unit. The frequency of measurements depends on the particular case; it may range from several times a week to several times daily. This tonometric device stores the measured values as well as the time stamps and communicates them to the MedStage server where the values are stored in the data base.

Data base triggers fire an event that is communicated to the rules engine. The rules engine evokes an Arden Syntax MLM that reads the new IOP values as well as all other needed input values and evokes the fuzzy rules. Finally a message is generated and communicated by the message server.

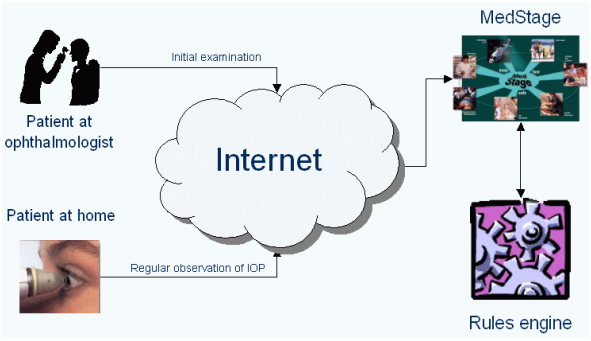

Figure 3.21: Structure of the monitoring application

3.3.2 Creation of the MLMs

The fuzzy rules set was initially modeled using the FuzzyTech[74] software, which is a commercial tool for the definition and evaluation of fuzzy control rule sets. The knowledge base can be saved as an ASCII text file in a proprietary format and can additionally be converted into a program code, such as a Java class file, which may then be directly used by a software application.

Using the program code representation of the FRS would imply linking the code directly to the program, which therefore would have to be recompiled every time the rules are altered. A more flexible way of implementing the rules is to use the Arden Syntax rules engine. A set of Arden Syntax MLMs has been generated, which represents the FRS, and was used for the glaucoma monitoring program.

The FuzzyTech file format is based on a structured text format. A parser based on JLex/Jay directly generates MLMs from a FuzzyTech project file. The parser has generated a total of 28 MLMs.

Linguistic variables are represented by 17 MLMs, including seven input variables, one overall output variable, and nine intermediate variables. Each set of production rules is represented by its own MLM; in all there are 11 production rules MLMs (figure 3.22). Although the MLMs were created automatically by the parser, some linguistic variable MLMs had to be modified and two additional control MLMs had to be written. One such linguistic input variable is shown in figure 3.23.

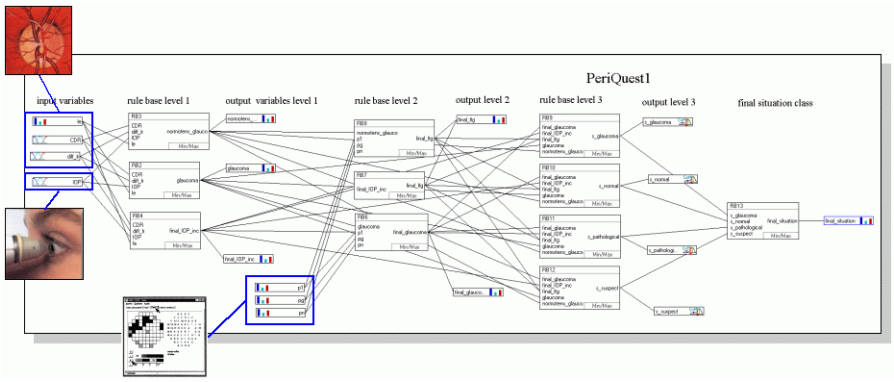

Figure 3.22: Structure of the fuzzy control rule set

[74] http://www.fuzzytech.com

```
maintenance:
  title: linguistic variable IOP;;
  mlmname: LV_IOP;;
  arden: Version 2flv;;
  version: 1.0;;
  institution: Siemens Medical Solutions;;
  author: Sven Tiffe;;
  specialist: ;;
  date: 2002-01-03;;
  validation: testing;;

library:
  purpose: This linguistic variable represents an input value of
           the TOSCA glaucoma classification rules.
           It is used as input by the rule blocks: RB2, RB3, RB4.;;
  explanation: ;;
  keywords:  ;;
  citations: ;;
  links:  ;;

knowledge:
  type: linguistic variable;;
  values: 'normal', 'increased';;
  input: read { %event.iop:date% };;
  range: 0.0, 70.0;;
  unit: 'mmHg';;
  sets:
    'normal'    := linear((0.0, 1.0), (20.0, 1.0), (22.0, 0.0), (70.0, 0.0));
    'increased' := linear((0.0, 0.0), (20.0, 0.0), (22.0, 1.0), (70.0, 1.0));;
end:
```

Figure 3.23: Arden Syntax linguistic variable MLM: IOP

Modifications of the linguistic variable MLMs affected the fuzzification of the seven input variables. As the FuzzyTech project file does not provide any information about data sources for the numerical input values, all necessary information about data base queries had to be added to the input slots.

The intermediate linguistic variables are the result of intermediate production rule sets. Thus the input slot is already generated by the parser, as the variables simply reference those MLMs that represent the intermediate production rule sets in their input slots. The final result of the FRS is one output variable that is defined analogously.

In an Arden Syntax fuzzy control knowledge base structured like the present one, the classifier can be called in two ways, namely by backward and forward inferencing.

The backward inference process is started by referencing the final output variable and initializing it automatically[75]. The variable would then call the production rule MLMs that are referenced in the input slot. These MLMs would initialize their linguistic variables, which would have to resolve their input slots, and so on. Those input variables that are not the result of intermediate rule sets have to define a data base query to retrieve the initial numerical input values.

The forward inference process is controlled by an additional MLM that first has to query

[75]This presumes that every linguistic variable defines the input slot.

all required data from the data base and initialize the input variables manually by passing the numerical values as arguments. Then every production rule MLM is evoked with input variables as arguments. These MLMs return the intermediate variables, which are used as input variables for the next production rules MLMs, and so on.

As some input parameters had to be prepared before they could have been used for the linguistic variables, such additional MLMs have been used to create a forward inferencing process. Each ophthalmic data set is classified twice, first based on values of the right eye, then based on values of the left eye. If the patient only measured the IOP of one eye, the contralateral data set is not classified. The result is coded in an XML message, which can be used to generate a text message.

Chapter 4

Results and discussion

The Arden Syntax for Medical Logic Systems was extended by concepts of fuzzy set theory and fuzzy logic, which can be used to represent medical knowledge with inherent vagueness. The representation of such uncertain knowledge is based on fuzzified comparison operators that define a gradual transition from 'true' to 'false' instead of returning crisp truth values. Therefore, the data type of truth values was extended to represent, in addition to Boolean values, such gradual truth values.

As truth values may be used to control the execution of the algorithm by conditional statements, all related elements of the Arden Syntax have been extended. Fuzzy truth values can result in the execution of statements and operators within different conditional contexts that are a degree for the applicability of the current operations. As a result, data may have different degrees of presence if they were created or altered within a conditional context whose initial condition was only partly fulfilled.

Further, a new type of MLMs that represents linguistic variables has been defined. These linguistic variables can be used, for instance, within the logic of data driven rules as part of conditional expressions or as part of fuzzy control rules.

All extensions that were presented in this work have been discussed within the HL7 Arden Syntax SIG (compare the notes about the current development of Arden Syntax in the introduction). The group agreed that the representation of uncertain knowledge is an important issue and that future versions of the Arden Syntax should include such extensions.

The extended syntax was implemented by a Java-based rules engine. This engine was used to realize two knowledge bases for computer assisted diagnoses and to evaluate them. The first one represented the rheumatological knowledge of the CADIAG-II expert system, the second one represented a glaucoma classifier. This chapter summarizes the evaluations of the application of the methods and presents their results.

4.1 General aspects

The main goal of this work was to represent uncertain knowledge in the sense of vaguely defined concepts. Such concepts are the result of an abstraction process where facts are represented by symbols (for example by terms) and symbols are aggregated to other more abstract symbols (for example, concepts such as disease definitions). When real-world facts, such as observations or measurements, are described linguistically, data gets

mapped to terms. In the special case of describing numerical values, often a range of values is assigned to a certain term. One example is the widely used term 'fever' that often represents in common speech all elevated temperatures in a patient; more specifically it may be classified into various types of increased temperature. Another example is the concept of 'adult' that may depend on the context in which it is used.

In any case, to represent a category or concept by a formal syntax it is necessary to compare facts to the conditions that separate the category from others. Most common knowledge representation formats, which are based on a two-valued logic, support only the definition of concepts by sharp boundaries. Therefore a fact either belongs to the category or not, which works fine for unambiguous situations where the fact clearly belongs to a certain category but which becomes unintuitive if facts are near to the borders of the category. To model categories with vague boundaries fuzzy set theory was selected as the concept to represent such uncertain knowledge, and fuzzy logic was used to reason with fuzzy truth values.

Another aspect was the selection of a knowledge representation format that meets the requirements of being used in the medical domain. The Arden Syntax for Medical Logic Systems was explicitly designed to be used by medical experts, who do not have the knowledge or experience of programmers. The Arden Syntax includes a crisp logic, therefore the representation of knowledge with inherent vagueness by Arden Syntax was realized by applying concepts of fuzzy set theory and fuzzy logic to the syntax.

One of the key aspects of Arden Syntax was the readability and simplicity of its syntactical representation. Therefore, all syntactical extensions were chosen carefully in order to keep the syntax intuitively readable. The terms which were finally used for the syntactical extension were discussed over long periods with members of the HL7 Arden Syntax SIG and other scientists from different universities. As long as the author or reader of the MLM is familiar with the most basic concepts of fuzzy sets, the chosen keyword 'fuzzified by' should be intuitively understandable. Recently this method of "fuzzified" comparison operators has also been applied, independent of this work and Arden Syntax, on decision criteria in clinical guidelines [JJC+01].

4.1.1 Fuzzy comparisons as model of vague categories

Determining whether a fact belongs to the category or not can be realized by comparing it to the inherent conditions of the concept definition, where the boundaries of the concept definition may be defined fuzzily. Such a comparison can be realized using Fuzzy Arden Syntax comparison operators that yield a gradual truth value that specifies whether the current fact meets the definition of the concept absolutely, partly, or absolutely not.

Example 48: (Fuzzy comparison)

> The question whether a patient is adult or not can be formalized by the following Arden Syntax expression.
>
> ```
> age => 17.8 years fuzzified by 0.4 years
> ```
>
> A patient who is exactly 18 years old would neither be termed an adult nor a child, whereas a patient who is 18 and a half years old would be termed an adult in this example.

The core range of the elements that absolutely belong to the category (or, in other words, which are absolutely compatible with the definition of the category) was defined in the last

example to begin at 18.2 years. The "fuzzification" of the comparison can be interpreted as *acceptable left or right hand variation* of the input value compared to the core range of the concept definition. In that example, only persons who are younger than 17.8 years were defined as children. All other patients are in between and are termed 'adult' by a degree between 'true' and 'false'.

A subset of the Arden Syntax comparison operators have been extended by such a variation that is represented by the keyword 'fuzzified by'. These can be used to represent all categories and concepts that are based on a numerical universe of discourse, by comparing sample data values to the vague boundaries of the represented concept. Other concepts that are not based on a numerical universe of discourse cannot be explicitly represented by such fuzzified comparison operators. One way to represent such concepts would be to implicitly define the degree of compatibility by the degree of presence of data values. A list of strings, for example, could represent a set of non-numerical values that are compatible with a certain concept. The degree of compatibility is then represented implicitly by their degree of presence.

The result of fuzzy comparisons is a fuzzy truth value that necessitated an extension of the Arden Syntax data model. Fuzzy truth values are an extended data type of the former Boolean, whose classic values 'true' and 'false' are still usable, yet are only the border cases of gradual truth values in between. Whenever a data value is compared to a fuzzily defined category the resulting fuzzy truth value represents its degree of compatibility to the category.

4.1.2 Use of fuzzy conditional statements

Conditional expressions that use fuzzy truth values execute their dependent operations under a conditional context that depends on the degree of truth of the fuzzy truth value. Such expressions are the 'where' statement, which selects data in dependence of a condition, and statements for algorithmic program flow control ('if-then' statement, 'while' loop, 'conclude' statement).

If a conditional expression of type 'if-then' or 'while' has to execute one of two code blocks and the condition yields neither 'true' nor 'false', then both blocks are executed in parallel, both blocks under their corresponding reduced conditional context. This parallelism follows the concept of the compositional rule of inference that mathematically models uncertain 'if-then-else' decisions by considering both alternatives gradually by a degree of applicability (compare section 1.3.4). The compositional rule of inference uses fuzzy sets to compute the result, whereas Fuzzy Arden 'if-then' statements apply the concept to single values ('while' loops are threatened analogously). If the conditional context is reduced, all results of the operations within that context have a reduced degree of presence and are termed 'fuzzy data'.

If one variable represents in different conditional contexts different values of the same data type, the final result is computed by calculating a weighted average (if the values are numerical) according to the compositional rule of inference. If the results are not numerical or of different types, the one with the higher degree of presence is chosen. This method is comparable to the 'execution with threshold'[76].

[76]Lists are only interpreted as collection of data and not as own data entities. According to the specification it appears to be not clearly defined whether lists have a primary time of their own; the source of the primary time is time information stored in the data bases and delivered together with their corresponding data as a result of queries. The time information is stored with the individual data values but is not

The application of these concepts, which were proposed by Zadeh [Zad73], to selected elements of the Arden Syntax resolved the algorithmic problems that occurred, when the Arden Syntax Boolean truth data type was extended to a fuzzy truth data type.

As truth values are not only used by conditional expressions as described earlier but also for concluding an MLM, the concluding process was adopted to the use of fuzzy truth values. In the sense of fuzzy rules it is now possible to create messages that can cover borderline cases in a smooth way by including the degree of conclusion in their text body. Mainly, the newly defined 'terminate' statement equals in its functionality the old 'conclude' statement (except for executing the action slot even in the case of a 'false' conclusion). To reduce the efforts of learning the differences between recent versions of the Arden Syntax and a new one that supports the extensions proposed in this work it might have been appropriate to keep the 'conclude' statement instead of redefining it and to use a new statement for the extended concluding functionalities. However, from the point of view of intuitive labels, the termination of the logic by 'terminate' seems to be more intuitive.

The influence of the extended truth data type on the entire syntax is rather high. Fuzzy truth values influence conditional expressions, which influence the creation and modification of data which therefore can be "fuzzy" and have to be handled in some way by all other operators. An alternative solution to handle fuzzy truth values in MLMs would have been to define conditional statements only to work on crisp truth values and to generate error messages if fuzzy truth values would have been used. That would mean that the syntax would produce, but not process, fuzzy truth values.

Such a strategy would have avoided the significant functional extensions that were applied to almost every element of the Arden Syntax, as no distinction between 'crisp data' and 'fuzzy data' would have been made. On the other hand, the usability of the fuzzy truth values and the integration of vaguely defined concepts in the decision algorithm would have been significantly decreased, as shown subsequently: Such a separate handling of fuzzy truth values may be compared to the approach of modeling the fuzzy extensions by separate MLMs.

4.1.3 Native extension of the syntax versus use of MLM library

The functionality of the native extension of the Arden Syntax might have been achieved by an alternative approach as well. Inspired by an early presentation of this work at the HL7 working groups meetings, Mike Jones from Thomson Micromedex[77] developed an MLM that provided the mathematics to use fuzzy comparisons of numbers. This MLM can be called with a set of arguments that include the input value, the crisp thresholds and the ranges of acceptable variation. The result is a numerical value from 0 to 1 that exactly represented the fuzzy truth value that would have been the result of the natively extended comparison operator.

This approach could be extended by other MLMs that cover other aspects of the fuzzy concepts presented before and would offer an elegant way to extend the functionality by a MLM library without touching the syntax. However, this approach had certain disadvantages compared to the native extension. Expressions such as

```
if bp_value is within 100 fuzzified by 20 to 140 fuzzified by 10 then
```

associated with a list. Following this method to handle lists as data container, they have neither a primary time nor a degree of presence (or, always a degree of presence of 1.0).

[77] http://www.mdx.com

can be understood more intuitively than expressions that implement the same functionality by an MLM call:

```
result := call fuzzy_set with bp_value, 80, 100, 140, 150;
```

Here the numerical variable 'result' would represent a number that, in turn, represents the fuzzy truth value. In contrast to the extension of the truth data type by fuzzy truth values, this numerical truth value representation cannot be processed by the fuzzy logical operators and cannot be used by other operators or statements that require truth values as arguments. For example, any fuzzy logical expression would have to be implemented as separate MLMs too:

```
result := call fuzzy_or with res_condition1, res_condition2;
```

Complex fuzzy logical expressions may result in hardly understandable sequences of MLM calls. Further, numerical fuzzy truth values could not be mixed with truth values by logical expressions.

Conditional expressions could not be used with the numerical representations of fuzzy truth values when using the "fuzzy set MLM". Instead of writing

```
if age is greater than 18 fuzzified by 2 years then
  ...
endif;
```

one would first have to compute the numerical fuzzy truth value and then define a crisp threshold for the fuzzy truth value:

```
result := call fuzzy_set with age, 18, 20, 0, 0;
if result >= 0.8 then
  ...
endif;
```

Such conditional statements would, however, behave crisply. The gradual degree of presence of variables within the conditional context could then be implemented by a second variable or by two parallel lists (if more than one variable has to be handled) that would represent this additional attribute. The concept of fuzzy data and conditional contexts could hardly be implemented without greater effort.

```
result := call fuzzy_set with age, 18, 20, 0, 0;

/* result >= 0.8 */
dose1 := 15;
dose1_presence := result;

/* else */
dose2 := 25;
dose2_presence := 1-result;

dose := call nu_defuzzification with (dose1, dose1_presence),
                                     (dose2, dose2_presence);
```

4.1.3.1 Linguistic variables

Linguistic variables can be used to define rules by using linguistic concepts rather than numerical values. The use of linguistic variables for conditional statements makes it possible to define expressions that are similar to natural language, and thus fulfill one of the rationales of the Arden Syntax.

Furthermore, the extensions allow fuzzy control rules to be used for more complex tasks such as computer-aided diagnosis or therapy. Fuzzy control systems can be implemented by a set of MLMs which include production rules, linguistic variables, and control MLMs. Such a system has been presented in this work and is discussed later.

If an Arden-Syntax-based knowledge base includes a set of common linguistic variables, they could be used without the need to define separate data base queries to read the needed facts from the data base. However, the curly braces problem arising from non-standardized definitions of data base access expressions is not resolved by the use of linguistic variables. This problem affects the read statements within the linguistic variable input slot as well as those within the data slot of classic Arden Syntax MLMs.

The use of linguistic variables instances differs from the use of regular Arden Syntax data types in two aspects: First, Arden Syntax has no type declaration—every variable can be used with any data type without the need to define them separately. By using linguistic variables, the variable has to be declared within the data slot first, as the term definitions have to be known before any assignments are valid. Secondly, in contrast to a regular variable, multiple assignments to one linguistic variable will not overwrite the old values. Instead each value will be assigned by a certain degree which depends on the fuzzy truth value of the context, for instance the condition of the if-then statement.

Linguistic variables do not make direct use of the first part of extensions (fuzzy comparisons and fuzzy data) but their use by rule MLMs does not provide many benefits if they cannot be used within fuzzy if-then statements, such as:

```
if blood_glucose_level is 'significantly increased' then
```

Therefore, to use linguistic variables, native extensions are highly recommended.

4.2 Implementation of a rules engine

A runtime environment (rules engine) that is able to read, manage, and execute MLMs and can be connected to an information system by flexible interfaces was implemented. The rules engine was tested by a set of test MLMs and used in two projects.

4.2.1 Compilation of Medical Logic Modules

Recent implementations of expert systems based on Arden Syntax used different approaches to execute MLMs by translating them into machine-executable commands of any kind. At the Columbia Presbyterian Medical Center MLMs where translated into pseudo-codes (an intermediate format between programming languages and an executable machine code) which where interpreted by an interpreter [HCJC92]. Other systems translated MLMs into other high-level knowledge representation formats which were used by their execution engine [Pry94, JMB94]. An Arden Syntax compiler of the University of

Giessen directly translated Arden Syntax MLMs into PL-SQL code that could be directly executed within an Oracle data base [Taf99].

A different approach is employed by systems that compile MLMs into a different programming language. Such cross-compilers that generated C++ program code have been realized by the Linköping University [Joh97, Gao93] and by IBM [KR94]. A similar approach that used Java as the target programming language has been realized at the Henri Mondor Hospital in France [KCH+02].

These cross-compilers essentially use the same process. First, an MLM that is to be integrated into the system is translated into the programming language of the goal platform. Then this program code is also compiled, this time by the native compiler that is provided for the particular programming language, into directly executable program code. This code has to be integrated (linked) into the system.

The Linköping approach translates one MLM to one C++ class, the French approach translates it into one Java class. Both approaches partly use pure C++ or Java expressions and partly a library of methods that provide the functionalities of more complex Arden Syntax operators, such as the 'max' operator that selects the greatest element of a list. As Arden Syntax uses, in contrast to C++or Java, dynamic data typing, all MLM variables are encapsuled by objects of a special class that is part of the particular class library of both approaches, and that represents a union of all Arden Syntax data types. To be able to use such objects with common C++ operators, such as '==' (is equal), these operators were overloaded[78]. Java does not allow one to overload operators but only methods; therefore method calls replace such operators.

```
maintenance:
  title:    Prescription without PT test;;
  filename: pt_601;
  ...
knowledge:
  ...
  data:     Prothrombin_percentage := read last
            ({} where it occurred within the past 12 hours);
            new_weekly_dose := read last
            ({} where it occurred within the past 12 hours);
            signature := event {};;
  evoke:    signature;;
  logic:    if not (Prothrombin_percantage is present) and
                   (new_weekly_dose is present) then
               conclude true;
            else
              conclude false;
            endif;;
  action:   write "No new dose must be given without a new PT percentage!";;
end:
```

Figure 4.1: MLM representation by cross-compilers: source MLM

For these approaches the given examples showed a high similarity between the contents of the Arden Syntax logic slot and its representation by the program class. Figures 4.1 and

[78]To overload means that different functionalities are assigned to one operator or method, which are selected depending on the data types of the arguments. For example, when comparing the string constant "foo" with a C++ string variable, the == operator would use its common functionality. If the variable is not a C++ string variable but an Arden Syntax variable representing an Arden Syntax string, the overloaded operator would still know how to handle it by using a different method to compare the values.

4.2 show one example taken from [GJS+93]. The first figure shows the source MLM that is translated into one C++ class shown in the second figure[79]. The result looks structurally similar to the source Medical Logic Module.

```
 0: #include "mlm_to_c++.hxx"
 1: mlm_type &pt_601{
 2: // Declaration of variable
 3: mlm_type Prothrombin_percentage;
 4: mlm_type new_weekly_dose;
 5: mlm_type signature;
 6:
 7: mlm_type NULL_TYPE;
 8: BOOLEAN_TYPE conclude = false;
 9: mlm_type action_message;
10:
11: //data slot part
12: Prothrombin_percentage.read_last(1, "..", "it occur within_the_past( 12 hours )");
13: new_weekly_dose.read_last(1, "..", "it occur within_the_past( 12 hours )");
14:
15: // evoke criteria in <Evoke_table> file
16:
17: // logic slot part
18: while (1) {
19:     if (!(Prothrombin_percentage.present()) && (new_weekly_dose.present())) {
20:     conclude = true;
21:     break;
22:     } else {
23:     conclude = false;
24:     break;
25:     }
26:     break;
27: }
28:
29: // action slot part
30: if (conclude) {
31:     // do action
32:     action_message = "No dose must be given without a new PT percentage! (ref:pt_603)";
33:     write (action_message);
34: }
35: NULL_TYPE = null;
36: return NULL_TYPE;
37: }
```

Figure 4.2: MLM representation by cross-compilers: result

First the required class library has to be included before the current class is defined as MLM data type labeled by the MLM filename[80] (lines 0 and 1). All local variables have to be defined before they can be used (lines 3 to 9). As mentioned earlier, the C++ class uses a special data type 'mlm_type' to encapsule MLM variables that can be of dynamical data type. When comparing both representations one can see where operators were overloaded (for instance the comparison operators in lines 20 and 23) or where Arden Syntax functionalities have been realized by separate methods within the class library (for

[79] A translation into Java classes that is based on the use of a class library probably would differ only in marginal parts. Further, both examples were slightly reformatted.

[80] The MLM example is based on Arden Syntax version 1 where the 'mlmname' slot still was termed 'filename' slot.

instance the read last function in lines 12 and 13 or the 'is present' operator in line 19).

As the next step the resulting C++ class must be compiled into executable code and linked to the system program code. During the pre-compilation, evocation information are also collected and stored externally (line 15). The system then uses the information to evoke the required MLMs [JW92, JB93]. Even if Java does not require to explicitly link the Java code into a program (it can be accessed automatically if it is located in a predefined name space (*classpath*), the process of integrating the translated MLM into the system is similar.

In contrast to such Arden Syntax execution environments that cross-compile MLMs to a high-level programming language, the rules engine presented in this work interprets the MLMs during runtime. A part of the last example is shown as an object tree in figure 4.3. The figure shows the representation of the logic and action slots in the knowledge category. Objects such as 'KnowledgeCategory', 'StructuredSlot', or 'IfThenStatement' are linked by references such as 'logic' or 'first_expr'. Constants include their value, such as the 'true' 'ArdenBoolean' constant. This Java object tree representation of MLMs also required a class library which provides the required object classes.

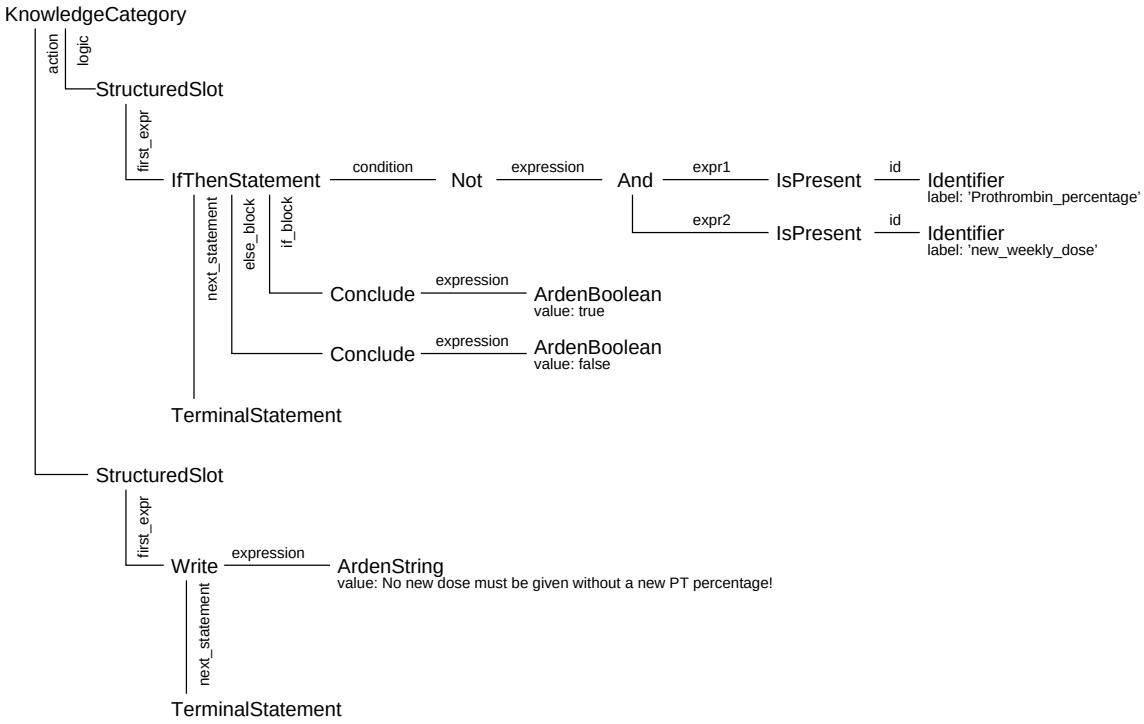

Figure 4.3: Example MLM representation by Java object tree

This representation of the MLM is naturally not designed for a human reader; it is used during the actual runtime by the rules engine[81]. The step of an intermediate representation by another programming language has been avoided in this approach; the user or knowledge engineer only has to deal with Arden Syntax MLMs in their plain text representation.

The cross-compilation approach was discussed as an option for the implementation of the rules engine in an early stage of this work. However, the object tree approach was chosen for reasons of flexibility and maintainability.

[81] However, a semantic-net-like representation of the internal object tree representation seems to be well readable for humans as well.

The flexibility mainly concerns loading MLMs into or unloading them from the running system. As the MLMs are represented by the object tree approach as internal data structures, they can be easily loaded or unloaded during runtime without any major effort. Only the evocation tables that define which MLMs listen to which events have to be updated. No additional tools such as an external compiler have to be run. In contrast, both cross-compilation approaches require the evocation of an external compiler and loading the newly compiled classes either before the engine starts up (direct linking by C++) or during runtime on program code level (by a customized dynamic Java class loader). Therefore, both cross-compilers modify the program code (or class scheme) of the rules engine, while the presented only one modifies internal data structures.

In terms of maintainability it is interesting to note, how easily the Arden-Syntax-relevant parts of the rules engine can be updated or modified. The class scheme of the presented rules engine groups all elements of the Arden Syntax into one package (separated by sub-packages) where every functional element such as a slot, an operator, or a statement is represented by one class. The individual functionality of one such element is basically implemented in the evaluation method. The compilers only use the constructor methods of the classes to build the object tree. The parsing process that was used to read and analyze MLMs was separated into two parts. The basic Arden Syntax parser and lexer built a Java object tree that represented the first and second category of the MLM and included the last one as source text. After evaluating the version of the MLM, the corresponding parser (either version 2, version 2 with fuzzy set extensions, or additionally with linguistic variables) was evoked, which analyzed the last category. As all extensions only affected the knowledge category (and it may be assumed that further extensions will also do so), this method kept the single versions of parsers separately maintainable.

Therefore, changes in the syntactical part of the syntax require alteration of the parsers while changes in functional parts require alteration of the corresponding classes. The cross-compilers also use a class library that provides Arden Syntax specific functionalities; thus the maintainability is comparable.

4.2.2 Java class model

The class model that was used to represent Arden Syntax MLMs as an object tree was modeled according to the structure of the syntax defined by the specification [Hls99]. An MLM is represented by one object that references three category objects, which reference a set of slot objects, and so on. Functionally related elements have been grouped into Java packages that were partially named according to the corresponding sections in the specification document. For instance, the package ‘operators’ includes only operator classes that are grouped into sub-packages such as ‘aggregation’ or ‘logical’.

First a Java class model for classic Arden Syntax version 2 was created. Then the fuzzy extensions were applied to the classes. From the viewpoint of the class model, this only required the addition of some new classes such as the defuzzification operator or a class that provides some fuzzy mathematics, which have been grouped into a separate package, and some new classes for the representation and handling of linguistic variable MLMs.

The extensions in terms of functionality had to be applied to every single class. Default extensions, such as the degree of presence of data or the default handling of the degree of presence, were simply implemented within the abstract root classes of operators or data types. For example, the root class of Arden Syntax data types was the abstract class ‘ArdenData’. This class defined basic properties of data such as the primary time. All

other data types inherited this class and therefore implemented the primary time as well. When the root class had been extended by the degree of presence all other data types were extended simultaneously.

The modular concept of the runtime part of the rules engine should allow easy connection to other information systems without major effort. The MedStage specific classes are limited to one package that includes 22 classes (of about 280 classes in sum) that implement all interfaces. As they are realized using the Java 'interface' technology (that only defines the functionality of the interfaces, not their implementation) they could be easily replaced by other classes that interface to other systems.

4.2.3 Performance

An important aspect of executing MLMs is the time needed for execution. In addition to the performance test of the bigger knowledge bases used in the two projects described in the next section, the performance was measured by three MLMs, running the rules engine on a common personal computer with a Pentium III CPU at 800 MHz and 512 MB memory under Microsoft™ Windows™ 2000.

The time needed for the execution of one MLM was measured by creating a time stamp using a function of the Java-3D class library that provides a method to obtain the actual time. According to the information of the package the resolution is less than 1 ms (279 ns). Time stamps are created when the execution of the knowledge category (the data slot) is started and one, when the last statement of the action (or logic) slot is terminated. Table 4.1 shows some results of the performance tests in milliseconds (ms).

Table 4.1: Performance of the rules engine

mlm	**average [ms]**	**std. dev.**	**maximum [ms]**	**minimum [ms]**
bp_1	559	83	661	450
bp_2	158	26	210	140
bp_3	55	22	91	30
test_1	549	106	671	421
test_2	141	13	161	120
bmi	<1	—	—	—

The first MLM retrieved a set of patient-specific blood pressure values from the data base, evaluated them, and sent a message by email. This MLM is referenced as 'bp_1' in the table. The second entry shows the performance of the same MLM but without sending the result as an e-mail. The third entry 'bp_3' shows the performance of the MLM without sending an e-mail and without data base access (instead the data are constructed as a constant list in the MLM). Sending the e-mail required the maximum time compared to the rest of the execution. On average the transmission of the e-mail took about 400 ms. Additionally the data base access required about 100 ms.

The next MLM that was tested for performance was one of the test MLMs which includes about 150 operations on aggregation operators plus a parallel result list and the comparison of the lists. The first version 'test_1' communicated the result as an e-mail while the second one did not. Again the average time difference is about 400 ms for a plain, short text. The pure execution of the MLM required on average 141 ms.

The last MLM calculated the body mass index (bmi), which is computed according to the gender, height, and weight of a person. This information is entered in a web-based form; the MLM is evoked via the direct evocation servlet. The results are returned as an XML file to the web browser which transforms them into HTML. The time needed for the execution of the MLM could not be measured as the time stamps were identical.

Recent studies identified the data base access as a bottle neck [HCJC92, JB93]. To reduce the communication overhead for retrieving small amounts of data from the data base the rules engine supports, based on the MedStage event system, the transportation of data by events. For example, while monitoring the blood pressure of a patient it is possible to link the data to the event that indicates a new pressure value instead of querying it separately.

As shown, another bottle neck may be the transmission of the results if transport layers with a significant protocol overhead are used. The more complex communication technologies are involved (requiring complex protocols for security reasons), the more negligible is the time needed to execute the pure decision logic, at least when dealing with rather small MLMs.

In summary, the results of MLMs are produced instantly. Considering the fact that it is purely an evaluation platform, the rules engine reacts with high performance. As every MLM is executed by an own thread, parallel running MLMs should benefit from multi processor computers where the load can be balanced on the single processors. Comparisons with MLMs that are directly cross-compiled to Java classes is difficult since the MLMs used for these performance benchmarks are different from those used by the French group. Their results published in [KCH+02] range from less than 5 ms for the execution of one MLM without data base access, to 76 ms for one MLM with read access to 3 data base tables.

Although the test MLMs of their work and the present study cannot be directly compared, the time needed for the execution of cross-compiled MLMs without data base access seems to be similar to the time taken for the "interpretation" of the corresponding Java object tree. The data base access of the MedStage systems seems to require more time, but as MedStage was designed for tele-medicine applications it includes sophisticated security concepts such as the authentication and encryption of data that require additional time.

4.2.4 Event handling

Events can be received by the rules engine in different ways; in the present work communication over JMS was mainly used. Events are defined by their unique name; therefore event mappings simply consist of this identifier. Events can be fired by any system that is connected to the JMS, such as data base triggers, devices, or user interfaces. In addition to the rules engine other systems also could receive such events. This event system was easy to implement as different vendors support JMS and provide communication solutions that are easy to use.

This event model causes all events to be processed asynchronously, as:

- Data base triggers must be asynchronous as current transaction should not be interrupted or held by trigger evaluation and event evaluation.
- System failures during the execution of the engine should never interrupt current data base transactions or crash systems which depend on the current operation.

- More than one software component could listen for events and the evaluation of the events could take different lengths of time. The communication is therefore unidirectional.

Thus all components of the event system and the event evaluation (the execution of rules) are independent threads; especially each individual execution process of an MLM runs as an own thread. In some cases a synchronous execution of rules might be needed, for example to analyze contraindications before ordering drugs or for interactive systems where the system should react instantly to user actions.

One issue that affects both, event handling and the overall execution of an MLM is the identification of the patient who is associated with the current MLM. The examples defined in the appendix of the Arden Syntax specification document avoid direct references to a specific patient in their messages and seem not to include any patient-related identifiers in their data base queries.

The current engine provides, as syntactical extension, a keyword 'patienid' that can be used in the data, logic, and action slot. By doing so it is possible to define data base queries that return only the data related to the current patient. Furthermore, it is possible to include the patient identification in outgoing messages.

4.2.5 MLM authoring

In recent implementations of Arden-Syntax-based systems, the scope included not only the execution of rules but also aspects like authoring and management. In addition to an MLM editor (which we implemented as a syntax highlighting scheme for Ultra Edit[82]), Gao used an MLM pretty-printer which (re-)formats MLMs, as well as an MLM manager for maintaining and linking MLMs to the system [MAG+91]. The present rules engine partly provides such aspects, as single modules can be reloaded or erased from the system during runtime. Furthermore, MLMs can be displayed by an HTML web site.

The Java classes that represent MLMs provide methods for each class to build an XML representation of the current MLM. This representation does not follow the proposals given in the Arden Syntax SIG, as it was not meant to be an exchange format for MLMs but just an output format that would be easily converted to other formats, such as HTML. The XSLT style sheet that was implemented for this specific conversion produced an HTML file that additionally made use of CSS style sheets. Therefore, to alter structural parts of the HTML output, the XSLT file would have to be modified: In order to alter its font shapes or colors, the CSS file would have to be altered. However, changes or extensions of the program code would not be required for these tasks.

Many editors published in recent work were based on forms which concealed the basic structure of an MLM behind a set of text input boxes [BE97, GSA+92]. By providing editors and graphical user interfaces, the representation of medical knowledge could be adapted to the skills and interests of the current user. For those, who are not experienced in writing MLMs (or generally using a formal syntax or programming language) it may be more productive when they are supported by a graphical environment. More skilled users may be faster when working directly on the syntax.

[82] All MLMs which were not created automatically were written with the UltraEdit editor (http://www.ultraedit.com), for which a syntax highlighting scheme was defined (appendix E). The scheme was kept ascetic, but was found to be useful for enhancing the clarity of an MLM.

4.3 CADIAG-II/RHEUMA^{+}Arden

The CADIAG-II/RHEUMA knowledge base was successfully converted to a Fuzzy Arden Syntax representation, modeling an inference process that is close to the one used by the original CADIAG-II system. Two different approaches were adopted to achieve the conversion; the two approaches differ in terms of performance and readability.

The modular approach defined each entity, such as a diagnosis, a symptom, or a combination, by one individual MLM. The inference process is started at the more abstract level of the knowledge base by calling all MLMs that define a diagnosis (compare figure 1.3 on page 17). The MLM calls continue until the lower level of abstraction (the data-to-symbol conversion) is reached. This level rates the symbolic representation of findings, returns the results to the symptom MLMs, which return their ratings to the next higher level of abstraction, until the level of diagnoses is again reached. As entities influence other entities on the same level of abstraction, and changes of one entity can therefore influence another that was computer earlier, the top-level call of the diagnoses is repeated until their ratings are constant.

The compact approach modeled the single inference steps used in CADIAG-II by individual MLMs, aggregating the knowledge nuggets of all entities that are used in the corresponding inference step. For example, a disease definition is defined by relationships to symptoms, to other diseases, and to symptom combinations. In this approach the information was separated according to the inference process into one MLM that defines all symptom-to-diagnosis relationships, one MLM that defined all diagnosis-to-diagnosis relationships, and one MLM that included the diagnosis-to-symptom-combination relationships (figure 4.4).

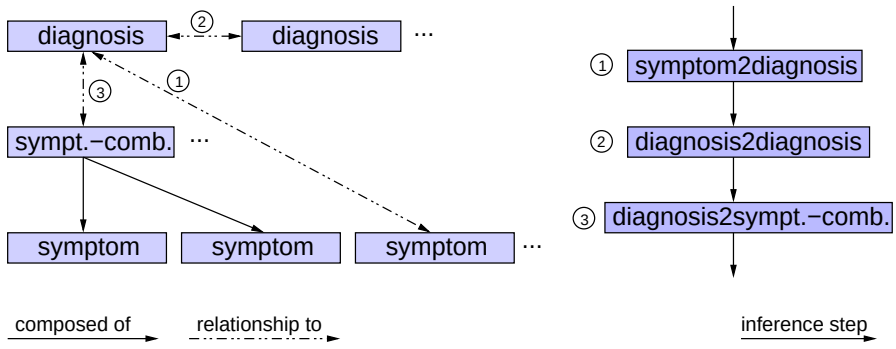

Figure 4.4: Distribution of related knowledge by the modular approach (left) and the compact approach (right)

4.3.1 Performance and readability

Aspects of performance and readability are discussed together as the readability of large Arden Syntax knowledge constructs seems to be reciprocally proportional to the performance of the inference process. All performance tests were run on the Arden Engine described in chapter 3.1.

In terms of readability the modular approach had advantages over the compact one. For example, the definition of a diagnosis aggregated all knowledge about the specific entity, such as relationships to other entities, in one individual MLM. The degree of evidence and the degree of occurrence of each relationship to another entity were defined separately

together with the name of the entity, which has been included as a comment. As every knowledge nugget could be localized easily, a knowledge engineer or physician would have been able to make single corrections and improvements in a convenient, direct way.

On the other hand, the structure of the modular approach had the disadvantage that entities referenced by many entities may be computed redundantly. Redundant computations were avoided by a mechanism that prevented calling those MLMs that had already been evaluated. This mechanism stored for each iteration of the inference process the names of the entities that were already computed (that is, the name of the corresponding MLM). Whenever such an MLM had to be called it checked whether the name was already stored in the list or whether the computation was still required.

The compact knowledge base broke the aggregated knowledge of entities, like symptoms and diagnoses, by separating it into single nuggets that are used by different steps of the inference process and therefore defined within different MLMs. For example, diagnoses are defined by three different MLMs, as the relationships to symptoms are evaluated first, then the relationships between diagnoses, and then additionally between diagnoses and symptom combinations. Besides, the MLMs aggregate the knowledge nuggets of different entities into one MLM.

One advantage of the modular system is its flexibility in terms of maintenance. By using the compact MLMs an entity whose definition needs to be partly changed must first be located within the MLM, then modified, and finally the entire MLM has to be reloaded. When using the modular knowledge base only the MLM that defines the entity is modified without affecting the rest of the system. However, compared to the original representation format by data base tables, both approaches provide a more easy maintainable alternative that could further be easily parsed and converted into other formats.

The amount of MLM calls of the modular approach slowed the system significantly. As the development of this approach was terminated before a complete inference process could be tested, the approaches cannot be directly compared by the time needed for a complete diagnosis. It showed the following: the higher the compactness of the MLMs, the faster is the inference process. On average, one complete inference process by the compact knowledge base needs about 19.9 seconds, including the data-to-symbol conversion. The early modular approaches required more than three minutes to complete approximately one half of the inference process. The original CADIAG-II system needs on average 2 seconds for the data-to-symbol conversion and 3 seconds for the inference process.

4.3.1.1 Use of native Java methods by interface statements

The need to access huge amounts of data base values and to share the ratings of all entities among the individual MLMs affects both the modular and the compact approach.

Before any data-to-symbol rule can apply, all required data have to be retrieved from the data base. The first modular versions of data-to-symbol MLMs implemented every data base query individually. As every individual query transferred only a few bytes and caused a significant overhead of communication (including all security issues such as verification), the entire retrieval of patient data was unacceptably slow. The performance was significantly improved by using a data base cache mechanism that reads all patient data by a few queries without losing the readability of the single data-to-symbol MLMs.

Data that were cached during the beginning of an inference process must then be available for every data-to-symbol conversion rule. Similarly, ratings of entities have to be available for every rule of the knowledge base. A simple way to share data by two or more MLMs

is to pass it along as arguments of an MLM call or to return it as a result of such a call. Sharing an unknown number of values could be realized by passing two lists that can store an arbitrary amount of data, where the first one stores all labels of the variables (the identifiers) and the other, the corresponding values. To look up one specific value, the position of its identifier has to be determined in the first list before the corresponding value in the second list can be accessed.

During each inference process the knowledge base has to handle about 2500 variables (2467 entities such as symbolic findings, symptoms, diagnoses, or scores) and about 1800 data base values (1305 documentation entries, 58 laboratory values, 11 crisp classifications, and 408 further findings); accessing one individual value by searching the label in a list and reading the value from the other would require too much time. In the best case the searched identifier is one of the first elements of the list; in the worst case the entire list has to be processed.

Performance tests showed that searching five labels (selected randomly) from a list of nearly 2500 elements required in the best case less than 1 ms and in the worse case about 500 ms. However, in the beginning of the inference MLMs, more than only five values have to be mapped to local variables. For example, mapping all needed variables for the symptom-to-diagnosis inference step took almost 80 seconds (1023 values being mapped) and looking up values needed by the intermediate combinations took about 24 seconds (327 values being mapped).

Therefore the data base cache and intermediate variables were realized using external Java methods accessed by a set of MLMs which use the Arden Syntax interface statement. This application significantly enhanced the performance of the system. Regardless of the number of queried values, the MLM terminated within a maximum of 5 ms. The MLMs used for comparing both methods are shown in the appendix C.5.3.

As the Java interface is capsuled by a set of MLMs, the readability of the MLMs is not significantly affected. However, the use of the Arden Syntax interface methodology reduces the portability of the knowledge base, as interfaces highly depend on the rules engine used. On the other hand, as long as the destination system supports Arden Syntax interfaces, the data base cache and intermediate variable storage could be implemented by the destination system, and the knowledge base could be transferred to another institution.

Another way to achieve better performance of the intermediate variable implementation would be to extend Arden Syntax in such a way that complex objects can be handled. If Arden Syntax would support objects—and currently the syntax is being developed in this direction—binary trees could be realized to reduce the time needed to localize one data object in a huge mass of objects.

4.3.1.2 Implication and inference operators

A further source of for potential performance problems was the implementation of the implication and inference operators as individual MLMs. Whenever a relationship is defined from one entity to another, the implication operator has to be executed to compute the influence of the implication on the current entity. This is done by calling the implication MLM with the value of the influencing entity and the degree of evidence and occurrence of the relationship. Then the operator returns the rating of the current entity that would be the result if the relationships was the only one defined for it. If the entity has already been rated or if other relationships are defined, these intermediate results are aggregated by the inference operator to a new rating for the current entity.

If an entity defines 335 relationships to other entities, the implication operator has to be called 335 times with the value and the two degrees that characterize the relationship as arguments. To speed up this calculation the operator was extended to accept lists of entities and lists of the degrees of evidence and occurrence as arguments and to compute the individual results internally in a loop. The result is returned also as a list with the results of the single implications. Therefore the entity described before has to call the operator only once.

The construction of the three lists can be implemented in two ways. The list can be constructed by first concatenating all values of the evidences to one list, then all corresponding degrees of evidence to another list, and at last by concatenating all degrees of occurrence to the last list.

Example 49: (Construction of the arguments for the implication operator)

```
impValue      := SF___34, SF___35, SF___36, ...
impEvidence   := fuzzy 0.1, fuzzy 0.05, fuzzy 0.05, ...
impOccurrence := fuzzy 0.07, fuzzy 0.4, fuzzy 0.39, ...
```

When dealing with a small set of related entities, this definition is still somehow readable, but as soon as the structures cannot be represented in one line anymore it becomes very expensive to search for corresponding pairs. In the applied Fuzzy Arden representation the lists are constructed this way, as lists with constants (thus the degrees of evidence and occurrence) can be easily optimized during the compilation of the MLM: Instead of evaluating all list concatenations during the runtime, the result that is constant and does not depend on the individual execution of the MLM can be already computed during the initial compilation. This optimization works as long as no variables are included in the expressions.

Alternatively, constructing the lists in single steps for each entity and including their names, for instance as comments, could result in a more clear definition of the relationships as shown in the next example.

Example 50: (Alternative construction of the arguments for the implication operator)

```
/* ONSET OF DISEASE, BETWEEN AGE 16 AND AGE 29 */
impValue      := SF___34;
impEvidence   := fuzzy 0.1;
impOccurrence := fuzzy 0.07;

/* ONSET OF DISEASE, BETWEEN AGE 30 AND AGE 50 */
impValue      := impValue, SF___35;
impEvidence   := impEvidence, fuzzy 0.05;
impOccurrence := impOccurrence, fuzzy 0.4;

/* ONSET OF DISEASE, AFTER AGE 50 */
impValue      := impValue, SF___36;
impEvidence   := impEvidence, fuzzy 0.05;
impOccurrence := impOccurrence, fuzzy 0.39;
...
```

The code could still be optimized during the compilation of the MLMs even if it would require—considering the pre-construction of lists that contain only constants—a more sophisticated algorithm that is able to detect the use of a list variable within the algorithm. List constructions which are implemented by one sequence of list concatenation operators

can be localized more easily. In the last example the list concatenations that belong together would have to be localized over a discontinuous sequence of statements that use the operators within their arguments.

4.3.2 Inference results

The base for the evaluation of the inference results was a printout of inference results for 116 patients. On approximately 2800 pages, for each patient detailed information and explanations are given about ratings of all positively or negatively rated symptoms, intermediate combinations, symptom combinations, and diagnoses.

As the comparison of the electronic results to these printouts is very time consuming, parts of the results were manually entered into an electronic format. To reduce the huge amount of data that had to be entered, only positively rated symptom combinations, intermediate combinations and diagnoses (both excluded and positively rated) were considered and entered into Microsoft^{TM} Excel^{TM} sheets. This data can be saved in formats that can easily be used by programs to compare the original results with the results of new systems electronically.

4.3.2.1 Conversion of the patient data

To evaluate the results of the Arden-Syntax-based system, more than 3000 cases were imported into the test data base. The export of the data from the WAMIS system data base was the challenging part, as it had been created about 20 years ago and most experts who where involved in the creation and definition of the data base scheme no longer work at the Department of Medical Computer Sciences. The task of exporting the patient data and reprogramming the export routines, which were written in PL-1, was mainly done by Dieter Kopecky.

The result of the export was used for this work and consisted of a set of plain text files in DBF format which were pre-processed and directly imported into an Oracle data base. The pre-processing was based on small Java programs, so that updated versions (which involved some corrections in the export routines) of the exported files could be easily converted and imported again, replacing the old versions. As the CADIAG-II data base scheme was adopted in an identical format, this step was devoid of any complications. Alternatively, as the MedStage data base can be used in an object-oriented manner, it would have been possible to reorganize the patient data in a way that all data of one patient could be represented as collections with one central patient object. This conversion was not done as the data base schema was considered suitable for this specific task.

CADIAG-II used job cards to start its inference process which provided all information to start the inference process and which were not available for the evaluation of the current system. As most job cards included additional information about radiological symptoms, all such symptoms per patient on the printout were initially collected, compared to the existing findings in the patient data base, and added if needed. After this step, the MedStage data base theoretically consisted of the same information as was available during the inference process of CADIAG-II.

4.3.2.2 Comparison of the results with printed original results

First evaluations of the inference results indicated that the inference process of the current CADIAG-II system might behave slightly differently than described in [Fis94]. Of course this assumption is only based on the interpretation of the printout results which were used for the evaluation.

Symptom ratings
As symptoms on the printouts were not entered in the Excel^{TM} sheets, they first had to be compared manually. Stochastic tests of the symbol-to-data conversion layer and the symptom-to-symptom relationships were made by comparing all positive- and negative-rated symptoms on the printouts to the results of the Arden-Syntax-based system. This evaluation comprised approximately 100 positive-rated symptoms and 1000 negative-rated symptoms per patient and was done for 10 patients.

The compared values were identical except for one methodological error which was corrected: The post-processing of radiological findings as described in appendix C.3 seemed not to be included to the inference process which produced the printouts. When including the post-processing MLM in the Arden-Syntax-based inference process, all affected symptoms which are unknown on the print-out are rated 'false'. After the MLM had been removed from the inference process the symptom results were identical.

Further evaluations of the symptoms were deferred until errors on higher levels of abstraction, such as combinations or diagnoses, were detected. The indication that the symptoms are correctly rated is strengthened when one compares the intermediate combinations based almost exclusively on symptoms.

Intermediate combination ratings
In sum, the 116 cases included 2554 ratings of intermediate combinations. Compared to the original CADIAG-II system, 98.51 percent of these ratings were computed identically by the Arden-Syntax-based system (table 4.2).

Table 4.2: Overall differences of intermediate combination ratings per patient

patient id	total ratings	differences
67013021	26	21
67009954	12	7
67008729	17	3
67021741	28	2
67019186	25	2
67002801	24	2
67018058	5	1
other	2417	0
total	2554	38

Seven cases included a total of 38 incorrect intermediate combination ratings. As the main error source, invalid data base entries were identified for 33 of these incorrect ratings. Table C.3 in the appendix on page 174 shows the differences of intermediate combination ratings in detail: Two intermediate combination definitions (PT 18 and PT 34) include diagnosis ratings and were rated incorrectly in five cases (highlighted by a red cell background).

Table 4.3: Overall differences of diagnosis ratings (in sum: 1234 ratings)

error	**num**	**perc**
missing data	23	1.86
filtered relation	15	1.22
correct relation	3	0.24
diagnosis to diagnosis	17	1.38
total	58	4.70

The intermediate combination 'PT 34' ("chronic diarrhea with loss of blood or discharge of mucus") combines the ratings of 'DF 107' ("arthropathy associated with ulcerative colitis") and 'DF 108' ("arthropathy associated with regional enteritis") by using the 'or' operator. In all three cases in which this combination was rated differently, the diagnosis 'DF 107' was rated 0.05 instead of 0.1.

Analogously, the second intermediate combination that includes diagnoses 'PT 18' ("rheumatoid arthritis or arthritis with collagenosis") was rated incorrectly twice, that is 0.8 instead of 0.9. In both cases the rating was based on the rating of 'DF 1' ("rheumatoid arthritis"), which was (wrongly) rated 0.8 instead of 0.9.

In summary, the data to symbol conversion works in a satisfactory manner. All erroneous ratings could been explained either by missing data in the data base or by the influence of more abstract entities. However, the remaining inference process led to further differences in the rating of more abstract entities, such as diagnoses and symptom combinations.

Diagnosis ratings

The first analysis of printouts showed that explanations of diagnosis hypotheses did not include diagnosis-to-diagnosis relationships—with a few exceptions. Therefore, and as first inference tests showed that most diagnosis-to-diagnosis relationships did significantly alter the result, such relationships were not included in the inference process. Only symptom-to-diagnosis and symptom-combination-to-diagnosis relationships were evaluated (similarly, the scoring mechanism does not use diagnosis-to-diagnosis relationships to compute the score of an entity). Furthermore, some sub-types of the symptom-combination-to-diagnosis relationships, where one degree was 1.0 and the other unknown, were also filtered out.

By using this modified inference process, very similar results were achieved. A total of 1234 diagnoses with a rating of at least 0.4 were generated; the detailed results are shown in tables C.4 to C.6 on pages 175 to 177. A total of 95.3 percent of the diagnosis hypotheses were identical. Most of the 58 erroneous ratings were based on incomplete data (table 4.3).

Three diagnosis hypotheses were computed correctly but did not show up on the printout of the corresponding case ("correct relation"). Another source of error was the reduced use of symptom-combination-to-diagnosis relationships ("filtered relation") and the exclusion of diagnosis-to-diagnosis relationships. However, including these relationships leads to new methodological errors, as shown next.

Symptom-combination-to-diagnosis relationships

Including the symptom-combination-to-diagnosis relationships in the inference process

could correct the 15 wrongly ratings described earlier, but would lead to new errors, as explained exemplarily using the relationship between the diagnosis 'DF 86' ("osteoarthritis of the hip") and the symptom combination 'BF 25' ("osteoarthritis of the hip (ON)"). The symptom combination is obligatory for the diagnosis, the degree of occurrence is therefore defined as 1.0; the degree of evidence is not defined.

$$\text{DF 86} \xrightarrow{\text{null}} \text{BF 25}$$
$$\text{DF 86} \xleftarrow{1.0} \text{BF 25}$$

The test data set includes 16 cases, where the symptom combination is excluded correctly (following the printout). As it is obligatory for the diagnosis the Arden-Syntax-based system excludes it, setting it to 'false' whereas the printout defines a positive rating (that however usually is lower than 0.5) for this particular diagnosis.

Diagnosis-to-diagnosis relationships
Similarly, the inclusion of diagnosis-to-diagnosis relationships in the inference process would correct the 17 erroneous ratings described earlier, but would introduce new falsely rated entities. For example, a relationship is defined between 'DF 87' ("osteoarthrosis of the knee") and 'DF 81' ("osteoarthrosis") with a degree of evidence of 1.0 and a unknown degree of occurrence.

$$\text{DF 87} \xrightarrow{1.0} \text{DF 81}$$
$$\text{DF 87} \xleftarrow{\text{null}} \text{DF 81}$$

The data set includes 46 cases, where the correctly rated diagnosis 'DF 87' increases the rating of 'DF 81', whereas on the printout this relationship is not evaluated. In some cases the difference is rather low but as the printout filters all diagnoses rated lower than 0.4, small changes due to this relationship may significantly influence the results displayed to the user.

Implementation of whole inference process
A test with the entire inference process as described in [Fis94] (including all relationships and the radiological symptom post-processing) resulted in 201 differently rated diagnoses compared to the printout.

Table 4.4 shows a brief synopsis of the changes in diagnoses ratings. The table includes only such diagnoses that have been rated on the printout at least with 0.4 and shows how many times the rating of the corresponding diagnosis was increased or how many times the diagnosis was excluded (values in parentheses indicate the number of ratings that changed from 'null' to a positive rating). Furthermore, the average relative change of the ratings is shown as the difference for increased or decreased (excluded) values; those in parentheses refer to ratings that changed from 'null' to a positive rating[83].

Additionally the first column categorizes the reason for the changes of the corresponding entity. This may be diagnosis-to-diagnosis relationships ('df 2 df'), symptom-combination-to-diagnosis relationships ('bf 2 df'), or the post-processing of x-ray symptoms ('post-pro').

As mentioned earlier, the attempt to correct the errors identified when analyzing the results of the reduced inference process directly led to new errors (differences between the electronic results and the printout). For instance, adding the diagnosis-to-diagnosis

[83]In these cases the average value need not be equal to the relative change, as the entities might have been rated with a positive value lower than 0.4 earlier and might have been filtered out in the output

Table 4.4: Selected differences of diagnosis ratings between reduced and complete inference process

diagnosis	source	increased	avg. inc.	excluded	avg. dec.
DF 25	df 2 df	(22)	(0.42)		
DF 36	bf 2 df			7	0.42
DF 62	df 2 df			1 + 1	0.5
DF 81	df 2 df	(22) + 24	(0.5)+0.1		
DF 84	post-pro			34	0.42
DF 86	bf 2 df			16	0.45
DF 93	df 2 df	(1)	(0.5)		
DF 123	df 2 df	(6)	(0.5)		
DF 137	df 2 df	(4)	(0.73)		
DF 161	bf 2 df			6	0.45
DF 165	bf 2 df			6	0.5
DF 168	df 2 df	(9) + 7	(0.9)+ 0.5		
DF 169	df 2 df	5	(0.5)		
DF 193	post-pro			1	0.5
DF 208	df 2 df	(6)	(0.7)		
DF 247	df 2 df	(1)	(0.4)		

relationships to the inference process corrected, on the one hand corrected the 17 erroneous ratings of 'DF 208'. On the other hand, evaluating this type of relationship resulted in 108 new errors, including 6 wrongly rated diagnoses 'DF 208'. Regarding this diagnosis, neither the one method nor the other one yielded a consistent result with no difference compared to the printout.

Similarly, the inclusion of all symptom-combination-to-diagnosis relationships corrected wrong ratings. Five ratings of 'DF 86' were corrected, but 16 new wrong ratings of the same diagnosis have been added. In sum, the inclusion of these relationships resulted in 35 entities that have been rated differently.

The inclusion of the post-processing of the radiological symptoms led to 35 new errors that occurred mainly for diagnosis 'DF 84'.

The results are slightly inconsistent as methodological errors were detected, and could not be corrected, since both alternatives created errors. To compare the results of the original system with the Arden-Syntax-based one (both with a reduced or full inference process) a medical expert should assess which results are "better" or less "wrong".

4.3.2.3 Use of alternative logical operators

The CADIAG-II system provided the possibility to use an alternative set of logical operators that never return 'null'. In such cases where the traditional three-valued operators 'and' and 'or' would return 'null', they would return 'false'. The use of these alternatively defined logical operators caused problems, as logical operators are explicitly defined by the Arden Syntax specification. The option to use alternative operators is not available in Arden Syntax.

As all test results that are present in the form of printouts are based on the alternative logical operators, these operators would have to be implemented by separate MLMs, which

would provide this extended function. However, like the use of other helper MLMs, the use of separate MLMs would have slowed down the system significantly. As work-around, the alternative operators have been implemented hard-coded in the engine.

As this native implementation breaks with the specification of the Arden Engine, a second inference run based on the traditional implementation of the extended Arden Syntax operators (as defined in section 2.3.3.3) was conducted. The results of the inference runs were compared in order to locate those entities that are rated differently by the different operator sets.

Intermediate combination ratings
Logical combinations are affected by two means. Firstly, as the combinations originally used logical operators that never returned 'null', all "unknown" combinations were excluded. Now, those intermediate combinations that were excluded based on missing data are rated as unknown ('null').

Furthermore, the classical Arden Syntax logic operators affected some intermediate combinations that were originally rated as being present. The error rate of 38 differently rated intermediate combinations increased to 90 differences, concerning only the two combinations 'PT 13' (which was rated 'null' instead of 'true' 13 times) and 'PT 50' (which was analogously rated 39 times differently).

Diagnosis ratings
The total number of differences between original and Arden-Syntax-based ratings increases from 58 to 249. Table 4.5 shows the concerned diagnoses which represent definitions of different stages of rheumatoid arthritis. Those diagnoses that are mainly computed with a high score, such as rheumatoid arthritis ('DF 1'), osteoarthrosis ('DF 81'), or osteoarthrosis of the knee ('DF 87'), are not affected and therefore seem to be robust against missing data.

Table 4.5: Differences in diagnosis ratings between alternative logical operators and classical (Fuzzy) Arden Syntax logical operators

diagnosis	**differences**	
	alternative	classic
DF 73	2	28
DF 74	1	22
DF 75	0	9
DF 77	0	16
DF 78	0	10
DF 79	2	14
DF 532	1	44
DF 538	1	33
DF 539	1	23
DF 175	1	0

As the set of entities that are affected by the extended operators is rather small, it might be possible to use the classic Fuzzy Arden logical operators by default and to implement the alternative ones as MLMs, using them only to compute the correct results for the detected

entities. Another option would be to use the classical Fuzzy Arden logical operators only, and to apply the 'is present' operator on entities that have to be rated as 'false' if the data are not available.

Analogous to the assessment of the results of the different inference implementations, this work should be done by a medical expert who is experienced enough to decide whether a missing entity can be interpreted as 'not present' with a high degree of probability (and can be used with the alternative operators, for instance).

4.4 Glaucoma monitoring

To realize a classifier that is based on fuzzy control rules, a Fuzzy Arden-based knowledge base that used fuzzy conditional statements and linguistic variables was created. The Arden-Syntax-based classifier was compared to an alternative representation of the same knowledge base that was based on the Java programming language. The Java-based classifier was created automatically by the software that was used to model the fuzzy control rules ("FuzzyTech").

As medical knowledge was addressed in recent work, the evaluation focuses on technical aspects rather than medical ones.

4.4.1 Medical aspects

The medical evaluation of the Arden Syntax classifier was based on the data sets of 31 patients who were considered suspicious and were selected by a medical expert. The data sets were classified both by the FuzzyTech software and the Arden-Syntax-based one, which computed identical classifications.

If the intraocular pressure was within the range considered normal (up to 21 mmHg) the fuzzy rules correctly classified the data as non-glaucomatous. The test data comprised 13 such cases; three of them strongly indicated a pathological state of the eye, which however is not glaucomatous.

Furthermore, the test data included 11 cases in which the threshold-based classifier would have generated a serious warning message due to a significantly increased IOP value. However, only four of them were rated glaucomatous and three of them were classified as being not very suspicious. Even when the IOP was increased, the perimetry data were rated nearly normal.

The IOP values of the remaining cases were in the range of 21 mmHg to 25 mmHg and were rated suspicious to different degrees. In particular, six cases that were close to the lower threshold of 21mmHg were rated suspicious *and* glaucomatous, and would not have been classified in this manner by a simple IOP threshold classifier.

In summary, perimetry data and CDR parameters may be additionally used to detect glaucoma-related changes in intraocular pressure from other pathological influences on this parameter. Additional pathological states can also be detected and communicated to the patients or physicians.

4.4.2 Technical aspects

The fuzzy control rule sets of the glaucoma classifier have been successfully represented by an extended version of the Arden Syntax. Every linguistic variable and every production

rule set has been represented by its own MLM. Two control MLMs read data from the data base, initialize the input variables, and control the inference process. As a result, a textual message that includes the results for both eyes is generated.

The performance of the Arden-Syntax-based system is good enough to return results instantly (table 4.6).

Table 4.6: Performance of the Fuzzy Arden-based glaucoma classifier

31 runs in total	**msec/MLM**	
	total	classification
average	60	50
maximum	89	68
minimum	44	39
stddev	13	10

The time needed for one classification of the data of one eye takes an average of 25 ms. The pre-processing and creation of the message text require an additional 10 ms. The time required to display the message on the screen or to send it by e-mail is not included in these durations as this additional time expenditure would apply in equal measure to the Arden-Syntax-based classifier and the Java based one.

Compared to the representation of the classifier, which is based on Java, it needs a negligible amount of additional time. However, the performance advantage of the programming language-based classifier has to compete with the flexibility of the Arden-Syntax-based classifier.

Every time the rules are altered they have to be incorporated into the running system. In contrast to the classifier based on a program code, when using Arden Syntax MLMs no programmer was needed to integrate the rules into the system and no program code had to be directly linked into the information system, as the knowledge is strictly separated from the program code of the expert system. Whereas the generated program code had to be externally compiled and linked to the system, which usually requires to stop and to restart it.

Furthermore, both representations had to be modified and individualized manually, as data base queries and evocation procedures had to be implemented separately. A knowledge engineer or the medical expert can modify the MLMs and no programmer of the information system is required. Usually such a programmer would be required if the data base queries and the evocation issues were to be implemented at the programming language level. In other words, the knowledge base of this Arden-Syntax-based telemedicine system can be dynamically modified without stopping, recompiling, or restarting the software.

The reusability of the rules and linguistic variables is improved by the modular representation in Arden Syntax. The use of the linguistic variable MLMs is not limited to this particular classifier or fuzzy control production rules in general. Common linguistic variables, such as body temperature or blood pressure, could be used in fuzzy control rules as well in classical Arden Syntax rules to improve the readability of the code, by encouraging the author to use linguistic expressions.

However, the general overview of complex fuzzy rule sets is easier to comprehend when a graphical user interface, such as the one supplied with the FuzzyTech software, is used. Thus a graphical environment for developing Arden Syntax MLMs could not only improve the development of Medical Logic Modules in general, but also that of complex Arden-Syntax-based fuzzy rules in particular.

Chapter 5

Conclusion

Fuzzy Arden unifies an easily readable syntax with sophisticated concepts of fuzzy set theory and fuzzy logic to provide a means of representing uncertain knowledge by implementing vague concepts. The definition of linguistic variables by independent MLMs allows rule authors to use expressions that are close to their linguistic description and to use the benefits of fuzzy control systems.

Fuzzy Arden was not developed to be "yet another" fuzzy language or a fuzzy-mathematical framework. Thus, only selected concepts of fuzziness theories have been applied to the syntax. Fuzzy sets have been used to calculate fuzzy truth values of conditions that represented the degree of compatibility of one data value to a set, which was defined by the conditional expression. Fuzzy logic is supported in means of logical operators. Because of the functional support of fuzzy truth values and fuzzy data by all operators and statements, Fuzzy Arden is more than a mere MLM library that provides some fuzzy set operations. Almost all elements behave, if used with crisp truth values, like the classic Arden Syntax elements. However, even if the *syntactical* extensions of the basic features are straightforward, the *functional* extensions would require existing runtime systems to be updated and extended in every part.

In the future Arden Syntax knowledge bases might not only consist of a set of procedural rules but also of a set of linguistic variables. Every linguistic variable could then be used by any other MLM to achieve good maintainability. Thus, authors of MLMs could refer to medical concepts by using linguistic variables without having to redefine them every time they are used. However, such usage would imply that the linguistic variables are defined according to a general consensus between the users and authors concerning the interpretation of the terms. The concept of a linguistic variable MLM may be improved in the future by defining context sensitive linguistic variables. The values of a linguistic variable could then be defined, based on a set of different compatibility functions, depending on sex (crisp criterion) or age (fuzzy criterion), for instance. While crisp context criteria would result in selecting one compatibility function, fuzzy ones would result in two dimensional compatibility functions that define the value.

Rules engine
The rules engine served its purpose of being used to implement the extensions and evaluate the concepts. After implementing the basic rules engine that can read and execute common Arden Syntax MLMs, implementing the extensions became a manageable issue. The object-oriented design of the rules engine simplified the extension as some default methods could have been defined centrally. The engine has been tested by a semi-automated

verification process that was based on a set of test MLMs. This process was found useful, an official set of test MLMs by HL7 would be therefore desirable.

The software was not developed as a product; thus some issues still need to be resolved. An important feature could be an interactive debugger that is able to move stepwise through the algorithm and display the values of the local variables. This extension should be easy to realize as the execution-relevant parts within the classes of the operators, commands, and the runtime environment could be extended by such a stepwise execution mode without touching the rest of the engine.

Future work should also address security issues which were less relevant for this work. A clinical expert system must have access to data that usually are restricted to selected clinical stuff. As intruders could write MLMs that search for sensitive data or listen to patient-related events, the access to the knowledge base and altering or adding knowledge should be restricted at the level of the user interface.

CADIAG-II/RHEUMA$^+$Arden

The conversion of the CADIAG-II/RHEUMA knowledge base to Fuzzy Arden has been successful. The results of the inference process were close to the original results. However, certain factors made the new representation difficult. Reconstructing the inference process and the structure of the knowledge base has been a challenging task that has been largely undertaken by members of the Section on Medical Expert and Knowledge-Based Systems at the University of Vienna Medical School. As a by-product of this work, the test results and the corresponding test templates are available electronically and could be used for further projects to verify the inference results.

Compared to the representation of the knowledge base that was the source for the conversion to Fuzzy Arden, the resulting knowledge base representations improves the readability and should therefore improve the maintainability as well. The readability of the knowledge base seemed to be reciprocally proportional to the performance of the system. Therefore Arden Syntax may be a suitable intermediate format for the knowledge base but may not be the optimal one as an executable end format.

Generally the representation of the knowledge base should be driven by the skill level of the user and the desired use of the formalized knowledge. The representation format could be an arbitrarily complex syntax if only a machine has to infer on it. If such an complex and eventually unreadable syntax would result in high performance, it would be perfect for such use. If, on the other hand, humans have to understand the knowledge base, a more simple, possibly abstract representation could be required. Experienced Arden Syntax users might feel most comfortable, when directly coding rules by using a text editor and working with the syntax. Inexperienced users might primarily like to use a graphical knowledge acquisition tool that hides all syntactical constructs behind intuitively comprehensible user interfaces.

In respect of the CADIAG-II knowledge base, a specialized graphical knowledge acquisition tool could increase the maintainability of the knowledge on a high level of abstraction. The next level could be a very modular representation of the knowledge base by Arden Syntax where every single entity, such as a diagnosis or a symptom, is represented as an individual MLM. This representation could still be useful for skilled Arden Syntax engineers who are able to work more productively on the code rather than use the graphical interface. A compiler could then translate the Arden Syntax code to a more compact Arden Syntax code (if the rules engine uses Arden Syntax) or even to another formalism that is finally executed by the system.

Then, however, the output format would not be as important as if the users would have to maintain the knowledge directly in the chosen representation format. However, Arden Syntax would still have the advantage of being a widely accepted standard. If the extensions proposed in this work are included in the standard and implemented by vendors, the entire knowledge base could be transferred and used by other systems.

Future work on the Fuzzy Arden representation of the CADIAG-II knowledge base could include the explanation of given diagnosis hypotheses by clicking on the corresponding hypothesis in the web browser. Another issue is the output of unknown symptoms which, however, would strengthen the level of a given diagnosis hypothesis.

Glaucoma monitoring
The glaucoma classifier is exemplary for projects in which a large number of patients have to be monitored for changes in their vital parameters. When monitoring glaucoma-related changes in the eye status, the fund of data tends to increase rapidly due to the ongoing measurement of IOP values for each patient. The expert system can assist the physician in daily routine by sending alerts in the event of critical or suspicious eye states in their patients.

The fuzzy rule-based classifier has some advantages over a simple classifier that only monitors the IOP and uses crisp thresholds. Compared to such a classifier, the fuzzy rules classify the data sets more precisely and are additionally able to detect non-glaucomatous states with an increased IOP as well as glaucomatous states with IOP values close to the normal range. The use of linguistic variables based on the concept of fuzzy sets and fuzzy production rules avoids unintuitive changes of the classification results of borderline cases.

The explicit representation of knowledge by the Arden Syntax-based rules engine has certain advantages over systems that represent the knowledge implicitly in their program code. It provides high flexibility in terms of adding, removing, or modifying knowledge without having to modify the system.

Finally...
Fuzzy Arden has been worked out with the motivation to carefully enhance an existing format that has a certain support in the medical community as well as in the industry. As these extensions have an impact on borderline cases of decisions, they might be the "icing on the cake" whenever vague medical knowledge has to be represented.

At the recent meetings of the HL7 Arden Syntax SIG it became increasingly clear that the Arden Syntax will gradually become an object-oriented paradigm which could enable the representation of complex data values as data records with different attributes. Thus the extension that defines 'fuzzy data' by a new attribute, 'degree of presence', would perfectly meet this development. As another fundamental aspect of the object-oriented paradigm—the explicit representation of the functionality by methods—will probably not be included in the next versions of Arden Syntax; the functionality of operators and statements will still be implicitly defined by the specification document. To process fuzzy truth values and fuzzy data it will not be sufficient to add the additional attribute 'degree of presence' to the Arden Syntax, but it would require an extension of the syntax as proposed in this work. When adding other attributes that may represent other types of uncertainty, for instance Dempster-Shafer belief values, the functionality of the statements and operators would have to be extended analogous to process and to handle such attributes. Significant functional changes can hardly be avoided if rather complex methods are to be fully included in the syntax.

It has been a pleasure for the author to present and discuss these proposals with members of the Arden Syntax Special Interest Group of HL7 over the past years. Thanks to all of those who supported this work.

Appendix A

Bibliography

[Adl80] K.-P. Adlassnig. A fuzzy logical model of computer-assisted medical diagnosis. *Methods of Information in Medicine*, 19:141–148, 1980.

[AK82] K.-P. Adlassnig and G. Kolarz. Cadiag-2: Computer-assisted medical diagnosis using fuzzy subsets. In M.M. Gupta and E. Sanchez, editors, *Approcimate Reasoning in Decision Analysis*, pages 219–247. North-Holland Publishing Company, 1982.

[AKL+82] K.-P. Adlassnig, G. Kolarz, F. Lipomersky, I. Gröger, and G. Grabner. Cadiag 1: A computer-assisted diagnostic system on the basus of symbolic logic and its application in internal medicine. In *Medical Informatics Europe 82*, Springer, Berlin, 1982.

[AKSG86] K.-P. Adlassnig, G. Kolarz, W. Scheithauer, and H. Grabner. Approach to a hospital-based application of a medical expert system. *Medical Informatics*, 11:205–223, 1986.

[ALK96] K.-P. Adlassnig, H. Leitich, and G. Kolarz. Expert systems in rheumatology. *Rheumatology in Europe*, 25:104–108, 1996.

[AS00] K.-P. Adlassnig and M. Schuertz. Medizinische Expertensysteme. lecture notes, Institut für Medizinische Computerwissenschaften, Bereich für Medizinische Experten- und Wissensbasierte Systeme, University of Vienna Medical School, Spitalgasse 23, 1090 Vienna, Austria, 2000.

[ASG+94] H. Ahlfeld, N. Shahasavar, X. Gao, K. Arkad, B. Johannson, and O. Wigertz. Data driven medical decision support system based on Arden Syntax within the helios environment. *Computer Methods and Programs in Biomedicine*, 45:97–106, 1994.

[Ast92] *Standard Specification for Defining and Sharing Modular Health Knowledge Bases (Arden Syntax for Medical Logic Modules).* Philadelphia, 1992.

[BC90] T.J.M. Bench-Capon. *Knowledge Representation – An Approach to Artificial Intelligence.* Academic Press, London, 1990.

[BE97] M. Bång and H. Eriksson. Generation of development environments for the Arden Syntax. In *Proceedings of the 1997 AMIA Annual Fall Symposium*, pages 313–317, 1997.

[Bib93] W. Bibel. *Wissensrepräsentation und Inferenz*. Vieweg, Braunschweig, 1993.

[Bie97] B. Biewer. *Fuzzy-Methoden – Praxisrelevante Rechenmodelle und Fuzzy-Programmiersprachen*. Springer, Berlin, 1997.

[BS84] B. Buchanan and E.H. Shortliffe. *Rule-based expert systems: The MYCIN experiment at the Stanford heuristic programming project*. Addison-Wesley, 1984.

[Dup94] F. Dupuits. The use of the Arden Syntax for mlms in HIOS+, a decision support system for general practitioners in the netherlands. *Computers in Biology and Medicine*, 24:405–410, 1994.

[Eck00] B. Eckel. *Thinking in Java*. Prentice-Hall, 2nd edition, 2000.

[Fis94] F. Fischler. Die Wissensbasis und der Inferenzprozess des medizinischen Expertensystems CADIAG-II/E. Master's thesis, University of Vienna Medical School, 1994.

[Gao93] X. Gao. *Realizing medical decision support systems using the Arden Syntax as knowledge representation*. PhD thesis, Linköping University, Department of Biomedical Engineering, Medical Informatics, Linköping University, 58183 Linköping, Sweden, 1993.

[Gel94] M. Gelfond. Logic programming and reasoning with incomplete information. *Annals of Mathematics and Artificial Intelligence*, 12:89–116, 1994.

[GJS+93] X. Gao, B. Johansson, N. Shahsavar, K. Arkad, H. Ahlfeld, and O. Wigertz. Pre-compiling medical logic modules into C++ in building medical decision support systems. *Computer Methods and Programs in Biomedicine*, 41:107–119, 1993.

[GM99] A. Geissbuhler and R.A Miller. Distributing knowledge maintainance for clinical decision-support systems: the "knowledge library" model. In *Proceedings of the 1999 AMIA Annual Fall Symposium*, 1999.

[GRS00] G Goerz, C.-R. Rollinger, and J. Schneeberger, editors. *Handbuch der Künstlichen Intelligenz*. Oldenbourg, Munich, 2000.

[GSA+92] X. Gao, N. Shahsavar, K. Arkad, G. Hripcsak, and O. Wigertz. Design and functions of medical knowledge editors for the Arden Syntax. In K.C. Lun et al., editors, *MEDINFO 92*, 1992.

[H+89] G. Herold et al. *Innere Medizin*. Dr. med. Gerd Herold, 1989.

[HBS+01] S. Hoelzer, H. Boettcher, R. Schweiger, J. Konetschny, and J. Dudeck. Presentation of problem-specific, text-based medical knowledge:. In *Proceedings of the 2001 AMIA Annual Fall Symposium*, 2001.

[HCJC92] G. Hripcsak, J.J. Cimino, S.B. Johnson, and P.D. Clayton. The Columbia-Presbyterian Medical Center decision-support system as a model for implementing the Arden Syntax. In *Proceedings of the 1992 AMIA Annual Symposium*, 1992.

[Hel01] H. Helbig. *Die semantische Struktur natürlicher Sprache*. Springer, Berlin, 2001.

[HLP+94] G. Hripcsak, P. Ludemann, T. A. Pryor, O. B. Wighertz, and P. D. Clayton. Rationale for the Arden Syntax. *Computers and Biomedical Research*, 27:291–324, 1994.

[Hls99] Health Level Seven, 3300 Washtenaw Ave, Suite 227, Ann Arbor, MI 48104. *helbig:2001Arden Syntax for Medical Logic Systems*, 1999.

[Hls00a] Health Level Seven, http://www.hl7.org. *Clinical Decision Support and Arden Syntax TC meeting minutes*, Spring Working Group Meeting, May 2000.

[Hls00b] Health Level Seven, http://www.hl7.org. *Clinical Decision Support and Arden Syntax TC meeting minutes*, Fall Working Group Meeting, September 2000.

[Hls01a] Health Level Seven, http://www.hl7.org. *Arden Syntax SIG meeting minutes*, Working Group Meeting, January 2001.

[Hls01b] Health Level Seven, http://www.hl7.org. *Arden Syntax SIG meeting minutes*, Spring Working Group Meeting, May 2001.

[Hls01c] Health Level Seven, http://www.hl7.org. *Arden Syntax SIG meeting minutes*, Autumn Plenary Meeting, October 2001.

[Hls01d] Health Level Seven, http://www.hl7.org. *Clinical Decision Support TC meeting minutes*, Autumn Plenary Meeting, October 2001.

[Hls01e] Health Level Seven, http://www.hl7.org. *Clinical Guidelines SIG meeting minutes*, Spring Working Group Meeting, May 2001.

[Hls01f] Health Level Seven, http://www.hl7.org. *GLIF SIG meeting minutes*, Working Group Meeting, January 2001.

[Hls02a] Health Level Seven, http://www.hl7.org. *Arden Syntax SIG meeting minutes*, Spring Working Group Meeting, April 2002.

[Hls02b] Health Level Seven, http://www.hl7.org. *Arden Syntax SIG meeting minutes*, Winter Working Group Meeting, January 2002.

[Hls02c] Health Level Seven, http://www.hl7.org. *Clinical Decision Support SIG meeting minutes*, Winter Working Group Meeting, January 2002.

[Hls02d] Health Level Seven, http://www.hl7.org. *Clinical Decision Support TC meeting minutes*, Spring Working Group Meeting, April 2002.

[HPW95] G. Hripcsak, T.A. Pryor, and O. Wigertz. Transferring medical knowledge bases between different HIS environments. In H.U. Prokosch and J. Dudek, editors, *Hospital Information Systems: Design and Development Characteristics; Impact and Future Architecture*, pages 241–264, Elsevier, Amsterdam, 1995.

[Hri94a] G. Hripcsak. The Arden Syntax for Medical Logic Modules: Introduction. *Computers in Biology and Medicine*, 24:329–330, 1994.

[Hri94b] G. Hripcsak. Writing Arden Syntax Medical Logic Modules. *Computers in Biology and Medicine*, 24:331–363, 1994.

[JB93] B. Johansson and Y. Bergqvist. Integrating decision support, based on the Arden Syntax, in a clinical laboratory environment. In *Proceedings of the 17th Annual Symposium on Computer Applications in Medical Care*, McGraw-Hill, New York, 394–398 1993.

[JHHC98] R.A. Jenders, H. Huang, G. Hripcsak, and P.D. Clayton. Evolution of a knowledge base for a clinical decision support system encoded in Arden Syntax. In *Proceedings of the 1998 AMIA Annual Fall Symposium*, 1998.

[JJC+01] M.-C. Jaulent, C. Joyaux, I. Colombet, P. Gillois, P. Degoulet, and G. Chatellier. Modelling uncertainty in computerized guidelines using fuzzy logic. In Suzanne Bakken, editor, *Proceedings of the annual symposium of the American Medical Informatics Association*, pages 284–288. Hanley and Belfus, Inc., 2001.

[JMB94] R. Jenders, M. Morgan, and G.O. Barnett. Use of open standards to implement health maintenance guidelines in a clinical workstation. *Computers in Biology and Medicine*, 24:385–390, 1994.

[Joh97] B. Johansson. *Methods, design and development of clinical decision support systems based on the Arden Syntax*. PhD thesis, Department of Biomedical Engineering, Medical Informatics, Linköping University, S-581 85 Linköping, Sweden, 1997.

[JW92] B. Johansson and O. Wigertz. An object oriented approach to interpret medical knowledge based on the Arden Syntax. In *Proceedings of the 16th Annual Symposium on Computer Applications in Medical Care*, McGraw-Hill, New York, 52–56 1992.

[Kan92] A. Kandel. *Fuzzy expert systems*. CRC Press, Boca Raton, 1992.

[KAR01] D. Kopecky, K.-P. Adlassnig, and A. Rappelsberger. An overview of the current state of the MedFrame/CADIAG-IV project. In K.-P. Adlassnig, editor, *Intelligent Systems in Patient Care*, pages 89–96, Österreichische Computergesellschaft, Vienna, 2001.

[KCH+02] H.C. Karadimas, C. Chailloleau, F. Hemery, J. Simonnet, and E. Lepage. Arden/J: An architecture for MLM execution on the Java platform. *JAMIA*, 9:359–368, 2002.

[KR94] R.A. Kuhn and R.S. Reider. A C++ framework for developing medical logic modules and an Arden Syntax compiler. *Computers in Biology and Medicine*, 5:365–370, 1994.

[Kui75] B.J. Kuipers. A frame for frames: representing knowledge for recognition. In D.G. Bobrow and A. Collins, editors, *Representation and understanding*, pages 151–18, Academic Press, New York, 1975.

[Lud94] P. Ludemann. Mid-term report on the Arden Syntax in a clinical event monitor. *Computers in Biology and Medicine*, 24:377–383, 1994.

[Łuk30] J. Łukasiewicz. Philosophische Bemerkungen zu mehrwertigen Systemen des Aussagenkalküls, 1930.

[Łuk51] J. Łukasiewicz. *Aristotle's syllogistic*. Oxford University Press, Glasgow, 1951.

[MAG+91] G. Magyar, K. Arkad, X. Gao, H. Gill, O. Wigertz, and H. Ahlfeld. Strategies for efficient implementation of the Arden Syntax for medical decision support. In *International Congress on Medical Informatics MIE 91*, pages 222–226, Vienna, Austria, 1991.

[Min75] M. Minsky. A framework for representing knowledge. In P.H. Winston, editor, *The psychology of computer vision*, pages 211–277, McGraw-Hill, New York, 1975.

[NKK96] D. Nauck, F. Klawonn, and R. Kruse. *Neuronale Netze und Fuzzy-Systeme*. Vieweg, Braunschweig, 1996.

[PH94] T.A. Pryor and G. Hripcsak. Sharing MLM's: An experiment between Columbia-Presbyterian and LDS hospital. In C Safran, editor, *Proceedings of the Seventeenth Annual Symposium on Computer Applications in Medical Care*, 1994.

[Pry94] T.A. Pryor. The use of medical logic modules at LDS hospital. *Computers in Biology and Medicine*, 24:391–395, 1994.

[Rei91] U. Reimer. *Einführung in die Wissensrepräsentation*. B. G. Teubner, Stuttgart, 1991.

[RM75] E. Rosch and C. Mervis. Family resemblances: Studies in the internal structure of categories. *Cognitive psychology*, 7:573–605, 1975.

[Ros73] E. Rosch. On the internal structure of perceptual and semantic categories. In T.E. Moore, editor, *Acquisition of Language*, Academic Press, New York, 1973.

[Ros78] E. Rosch. Principles of categorzisation. In E. Rosch and B.B. Lloyd, editors, *Cognition and Categorization*, pages 27–48, Erlbaum, Hillsdale, 1978.

[Rus23] B. Russell. Vagueness. *Australian Journal of Psychology and Philosophy*, 1:84–92, 1923.

[San70] E.S. Santos. Fuzzy algorithms. *Information and Control*, 17:326–339, 1970.

[SB75] E.H. Shortliffe and B.G. Buchanan. A model of inexact reasoning in medicine. *Mathematical Biosciences*, 23:351–379, 1975.

[Sin70] A.A. Sinowjew. *Über mehrwertige Logik*. Vieweg, Braunschweig, 1970.

[SKB80] J.R. Searle, F. Kiefer, and M. Bierwisch. *Speech act theory and pragmatics*. Reidel, Dordrecht, 1980.

[Smi85] B. Smith. Prologue to 'reflection and semantics in a procedural language'. In R.J. Brachman, editor, *Readings in Knowledge Representation*, 1985.

[Taf99] A.G. Tafazzoli. *Klinisch einsetzbare wissensverarbeitende Funktionen in einem onkologischen Informationssystem*. PhD thesis, Justus-Liebig-Universität Giessen, Medizinische Informatik, Heinrich-Buff-Ring 44, 35392 Giessen, Germany, 1999.

[TAW+99] A.G. Tafazzoli, U. Altmann, W. Waechter, F.R. Katz, S. Hoelzer, and J. Dudeck. Integrated knowledge-based functions in a hospital cancer registry – specific requirements for routine applicability. In *Proceedings of the 1999 AMIA Annual Fall Symposium*, 1999.

[Who98] World Health Organiation, World Health Organization, Avenue Appia 20, 1211 Geneva 27, Switzerland. *The world health report*, 1998.

[Win92] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, 3rd edition, 1992.

[Wri75] C. Wright. On the coherence of vague predicates. *Synthese*, 30:325–365, 1975.

[Zad65] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[Zad68] L.A. Zadeh. Fuzzy algorithms. *Information and Control*, 12:94–102, 1968.

[Zad71] L.A. Zadeh. Quantitative fuzzy semantics. *Information Sciences*, 3:159–176, 1971.

[Zad73] L.A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Man, and Cybernetics*, 3:28–44, 1973.

[Zad76a] L.A. Zadeh. A fuzzy-algorithmic approach to the definition of complex or imprecise concepts. *International Journal of Man-Machine Studies*, 8:249–291, 1976.

[Zad76b] L.A. Zadeh. The linguistic approach and its application to decision analysis. In Y.C. Ho and S.K. Mitter, editors, *Directions in Largescale Systems*, pages 339–370, Plenum Press, New York, 1976.

[Zad87] L.A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning, part I–III. In R.R. Yager et al., editors, *Fuzzy sets and applications*, pages 219–366, Wiley-Interscience, New York, 1987.

[Zad89] L.A. Zadeh. Knowledge representation in fuzzy logic. *IEEE Transactions On Knowledge And Data Engineering*, 1:89–99, 1989.

[Zad90] L.A. Zadeh. The birth and evolution of fuzzy logic. *General Systems*, pages 95–105, 1990.

[Zad96] L.A. Zadeh. Fuzzy logic = computing with words. *IEEE Transactions on Fuzzy Systems*, 4:103–111, 1996.

[Zad99] L.A. Zadeh. From computing with numbers to computing with words—from manipulation of measurements to manipulation of perceptions. *IEEE Transactions on Circuits and Systems*, 45:105–119, 1999.

[ZSW97] G. Zahlmann, M. Scherf, and A. Wegner. A neuro-fuzzy-classifier for a knowledge-based glaucoma monitor. In *Artificial Intelligence in Medicine*, pages 273–284, 1997.

Appendix B

Fuzzy Arden Syntax BNF

In the official specification, the MLM syntax is defined using Backus-Naur Form (BNF):

> "The following definitions hold:
>
> - `<expression>` — represents the non-terminal expression
> - `IF` — represents the terminal if, iF, If, or IF
> - `:=` — represents the terminal :=
> - `::=` — is defined as
> - `/*...*/` — a comment about the grammar
> - `|` — or
>
> Terminals are listed in upper case, but the language is case insensitive outside of character strings. In structured slots, space, carriage return, line feed, horizontal tab, vertical tab, and form feed are considered white space and are ignored. In addition, the terminal **the** is treated as white space (that is, the word **the** is ignored).
>
> With minor modifications, the following grammar can be processed by an LALR(1) parser generator, except where noted by comments against individual rules."

This appendix includes only those parts that are affected by the extensions. For copyright reasons, the entire BNF is not published here.

B.1 Changes in the BNF for Fuzzy Arden without linguistic variables

In the section "expressions" the expression "fuzzified by" is added for simple comparisons:

```
<expr_comparison> ::=
      <expr_string>
    | <expr_string> <simple_comp_op> <expr_string> <fuzzby_expr>
    | <expr_string> <is> <main_comp_op>
    ...
```

The overall conclude value can be accessed only in the action slot by the keyword “concluding”.

```
<expr_factor_atom> ::=
      <identifier>
    ...
    | "CONCLUDING"          /* Value of "CONCLUDE VALUE" is NULL */
                            /* outside of action slot.           */
```

In the section “operators” the use of the fuzzification on binary comparison operators is defined:

```
<main_comp_op> ::=
      <temporal_comp_op>
    | <range_comp_op>
    | <unary_comp_op>
    | <binary_comp_op_crisp_behavior> <expr_string>
    | <binary_comp_op_fuzzy_behavior> <expr_string> <fuzzby_expr>
                /*  the WITHIN TO operator will accept any ordered parameter,
                    including numbers, strings (single characters), times,
                    Boolean /*
<range_comp_op> ::=
    "WITHIN" <expr_string> <fuzzby_expr> "TO" <expr_string> <fuzzby_expr>

<temporal_comp_op> ::=
      "WITHIN" <expr_string> <fuzzby_expr> "PRECEDING" <expr_string> <fuzzby_expr>
    | "WITHIN" <expr_string> <fuzzby_expr> "FOLLOWING" <expr_string> <fuzzby_expr>
    | "WITHIN" <expr_string> <fuzzby_expr> "SURROUNDING" <expr_string>
    | "WITHIN" "PAST" <expr_string> <fuzzby_expr>
    | "WITHIN" "SAME" "DAY" "AS" <expr_string> <fuzzby_expr>
    | "BEFORE" <expr_string> <fuzzby_expr>
    | "AFTER" <expr_string> <fuzzby_expr>
    | "EQUAL" <expr_string> <fuzzby_expr>
    | "AT" <expr_string> <fuzzby_expr>

<binary_comp_op_crisp_behavior> ::=
    "IN"
<binary_comp_op_fuzzy_behavior> ::=
      "LESS" "THAN"
    | "GREATER" "THAN"
    | "GREATER" "THAN" "OR" "EQUAL"
    | "LESS" "THAN" "OR" "EQUAL"
```

The optional fuzzification is defined by:

```
<fuzzby_expr> ::=
    /* empty */
    | "FUZZIFIED BY" <expr_string>
```

The Boolean value is extended by fuzzy truth constants:

```
<boolean_value> ::=
      "TRUE"
    | "FALSE"
    | "FUZZY" <number>
```

In the section "expressions" an additional keyword for sorting list elements by their degree of presence is added:

```
<sort_option> ::=
      "TIME"
    | "DATA"
    | "PRESENCE"
```

B.2 Further changes in the BNF for Fuzzy Arden including linguistic variables

The first modification affects the knowledge category body. In addition to the common "rule body" an alternative one is defined:

```
<knowledge_body> ::=
    <classic_body>
    | <lv_body>

<classic_body> ::=
    /* the former knowledge_body */
    /* ... */

<lv_body> ::=
    <lv_type_slot>
    <values_slot>
    <input_slot>
    <range_slot>
    <unit_slot>
    <defuzzification_slot>
    <sets_slot>
```

The slots are defined as follows:

```
/***** special slots for linguistic variables ******/
<values_slot> ::=
    "VALUES:" <lv_values> ";;"

<lv_values> ::=
    <term>
    | <term> "," <lv_values>

<input_slot> ::=
    /* empty */
    | "INPUT:" <input_block> ";;"

<input_block> ::=
    /* optional input can be done by one read statement or one or multiple mlm definitions */
    <input_mlm_block>
    | <input_read_statement>

<input_read_statement> ::=
    "READ" <read_phrase> ";"
```

```
<input_mlm_block> ::=
    <input_mlm_block> ";" <input_mlm_definition>
    | <input_mlm_definition>

<input_mlm_definition> ::=
    /* empty */
    | <result_selection> <input_mlm_assign_phrase>

<result_selection> ::=
    /* empty */
    | "RESULT" <number> "FROM"

<input_mlm_assign_phrase> ::=
    "MLM" <term>
    | "MLM" <term> "FROM" "INSTITUTION" <string>

<range_slot> ::=
    "RANGE:" <number> "," <number> ";;"

<unit_slot> ::=
    /* empty */
    | "UNIT:" <term> ";;"

<defuzzification_slot> ::=
    /* empty */
    | <defuzzification_method>

<defuzzification_method> ::=
    "DEFUZZIFICATION:" <id> ";;"

<sets_slot> ::=
    "SETS:" <sets_block> ";;"

<sets_block> ::=
    <sets_block> ";" <set_statement>
    | <set_statement>

<set_statement> ::=
    <term> ":=" <set_type> "(" <set_def> ")"

<set_type> ::=
    "LINEAR"

<set_def> ::=
    <set_point>
    | <set_def> "," <set_point>

<set_point> ::=
    "(" <number> "," <number> ")"
```

To assign linguistic values to a linguistic variable in the logic slot, the assignment expression is extended:

```
<logic_assignment> ::=
    <identifier_becomes> <expr>
    | ...
```

```
    | <linguistic_variable_assignment>

<linguistic_variable_assignment> ::=
    "SET" <id> "TO" <term>
    | "SET" <id> "TO" <term> "WITH" <expr>
```

The main comparison expression is altered to compare linguistic variables to terms:

```
<expr_comparison> ::=
    <expr_string>
    | ...
    | <expr_string> <is> <term>
```

Within the data slot, linguistic variable MLMs can be referenced analogous to common MLMs:

```
<data_assign_phrase> ::=
    "READ" <read_phrase>
    | "MLM" <term>
    | "MLM" <term> "FROM" "INSTITUTION" <string>
    | <init_lv> "LINGUISTIC" "VARIABLE" <term>
    | <init_lv> "LINGUISTIC" "VARIABLE" <term> "FROM" "INSTITUTION" <string>

<init_lv> ::=
    /* empty */
    | "INIT"
```

Finally the ‘defuzzify’ operator is defined:

```
<of_noread_func_op> ::=
    "ANY"
    | ...
    | "DEFUZZIFY"
```

Appendix C

Cadiag-II

C.1 Implication operator

The implication operator is used to compute the influence of an entity e_j to an entity e_i. The influence is controlled by the degree of evidence and the degree of occurrence of the relationship.

$e_j \rightarrow e_i$ with a degree of evidence

$e_j \leftarrow e_i$ with a degree of occurrence

e_j is known, e_i is to be computed

Table C.1: Definition of the CADIAG-II inference operator.

degree of occurrence	**degree of evidence** (b)			
	ϵ	0	$(0 \ldots 1)$	1
ϵ	ϵ	ϵ	ϵ, if $e_j = 0$ ϵ, if $e_j = \epsilon$ ϵ, if $e_j = \omega$ $\min(e_j, b)$, else	ϵ, if $e_j = 0$ ϵ, if $e_j = \epsilon$ ϵ, if $e_j = \omega$ e_j, else
0	ϵ	0, if $e_j = 1$ ϵ, else	ω	ω
$(0 \ldots 1)$	ϵ	ω	ϵ, if $e_j = 0$ ϵ, if $e_j = \epsilon$ ϵ, if $e_j = \omega$ $\min(e_j, b)$	ϵ, if $e_j = 0$ ϵ, if $e_j = \omega$ ϵ, if $e_j = \epsilon$ e_j, else
1	0, if $e_j = 0$ ϵ, else	ω	ϵ, if $e_j = \omega$ ϵ, if $e_j = \epsilon$ $\min(e_j, b)$, else	ϵ, if $e_j = \omega$ ϵ, if $e_j = \epsilon$ e_j, else

The operator is implemented as a separate MLM that is called from the inference MLMs which have to evaluate relationships. To increase the performance of the system, the number of MLM calls should be kept as small as possible; therefore the MLM can either be called with one related entity and the corresponding degree of evidence and occurrence,

or with three lists where the first represents n ratings of related entities and the other two lists represent the corresponding degrees.

Knowledge slot of the CADIAG-II implication operator MLM

```
knowledge:
  type: data-driven;;
  data:
    (valueList, evidenceList, occurrenceList) := argument;
  ;;
  evoke: /* direct call only */;;
  logic:
    if valueList is list then
        resList := ();
        i := 1;
        for value in valueList do
            result := null;

            value      := valueList[i];
            occurrence := occurrenceList[i];
            evidence   := evidenceList[i];

            occ0to1 := occurrence > fuzzy 0 and occurrence < fuzzy 1;
            ev0to1  := evidence > fuzzy 0 and evidence < fuzzy 1;

            if evidence = fuzzy 0 and occurrence > fuzzy 0 then
                result := "omega";
            elseif occurrence = fuzzy 0 and evidence > fuzzy 0 then
                result := "omega";
            elseif evidence is null and occurrence = fuzzy 1
                   and value is not null and value = fuzzy 0 then
                result := fuzzy 0;
            elseif evidence = fuzzy 0 and occurrence = fuzzy 0
                   and value is not null and value = fuzzy 1 then
                result := fuzzy 0;
            elseif ev0to1 = true then
                if value is not null and value > fuzzy 0 then
                    if value < evidence then
                        result := value;
                    else
                        result := evidence;
                    endif;
                elseif value is not null and value = fuzzy 0 then
                    if occurrence = fuzzy 1 then
                        result := fuzzy 0;
                    endif;
                endif;
            elseif evidence = fuzzy 1 then
                if value > fuzzy 0 then
                    result := value;
                endif;
            endif;

            resList := resList, result;
            i := i + 1;
        enddo;
```

```
        result := resList;
    else

        result := null;
        value      := valueList;
        occurrence := occurrenceList;
        evidence   := evidenceList;

        /*****************************/
        /* compute result analogously */
        /*****************************/

    endif;

    conclude true;;
  action:
    return result;;
```

C.2 Inference operator

The CADIAG-II inference operator is used to compute a new rating of an entity e_i by aggregating other related entities and the current rating of e_i. The symbol ω represents a conflicting rating, ϵ represents an unknown rating.

Table C.2: Definition of the CADIAG-II inference operator.

$e_i + e_j$	$e_j = \epsilon$	$0 < e_j$	$e_j = 0$	$e_j = 1$	$e_j = \omega$
$e_i = \epsilon$	ϵ	e_j	0	1	ω
$0 < e_i < 1$	e_i	$\max(e_i, e_j)$	0	1	ω
$e_i = 0$	0	0	0	ω	ω
$e_i = 1$	1	1	ω	1	ω
$e_i = \omega$	ω	ω	ω	ω	ω

It should be noted, that entities which where already rated as conflicting or are related to conflicting entities are automatically rated as conflicting. Further, because of the use of the 'max' operator the rating of an entity can only be increased during the inference iterations or because of higher rated related entities.

The inference operator has been implemented as single MLM. The MLM is called directly by other MLMs; the values which are to be aggregated are passed within a list as an argument of the MLM. The table is implemented as a nested 'if-then' structure.

Knowledge slot of the CADIAG-II inference operator MLM

```
knowledge:
  type: data-driven;;
  data:
    valuelist := argument;
  ;;
  evoke: /* direct call only */;;
```

```
  logic:
    result := null;
    if valuelist is list then
        if count valuelist > 0 then
            /* result is at least first element of list */
            result := valuelist[1];
            /* apply operator to every element */
            for val in valuelist do
                /* if result is conflicting, abort */
                if result = "omega" or val = "omega"
                  or (result = fuzzy 0.0 and val = fuzzy 1.0)
                  or (result = fuzzy 1.0 and val = fuzzy 0.0) then
                    result := "omega";
                    conclude true;
                elseif result = fuzzy 1.0 or val = fuzzy 1.0 then
                    result := fuzzy 1.0;
                elseif result = fuzzy 0.0 or val = fuzzy 0.0 then
                    result := fuzzy 0.0;
                elseif result is null and val is null then
                    result := null;
                elseif result is null then
                    result := val;
                elseif val is null then
                    result := result;
                elseif val > result then
                    result := val;
                endif;
            enddo;
        endif;
    endif;
    conclude true;;
  action:
    return result;;
```

C.3 Post-processing of radiological findings

```
maintenance:
  title: _CADIAG x-ray post-processing;;
  mlmname: X_RAY_POSTPROCESSING;;
  arden: Version 2;;
  version: $Id$;;
  institution: Siemens Medical Solutions, University of Vienna;;
  author: ;;
  specialist: ;;
  date: 2002-08-22;;
  validation: testing;;

library:
  purpose: This MLM sets those symptoms from x-ray group to false that still are null.
           This only concerns symptoms related to joints and the spine;;
  explanation: ;;
  keywords:  ;;
  citations: ;;
  links:  ;;
```

```
knowledge:
  type: data-driven;;
  data:
        valuesId := argument;

        /* storage and database tools */
        valueExtract := mlm 'value_extract';
        valueStore   := mlm 'value_store';

        symptoms_to_process :=
            "SF__494", "SF__747", "SF__753", "SF__754", "SF__755", "SF__756", "SF__757",
            "SF__758", "SF__759", "SF__760", "SF__761", "SF__762", "SF__763", "SF__764",
            "SF__765", "SF__766", "SF__767", "SF__768", "SF__769", "SF__770", "SF__771",
            "SF__772", "SF__773", "SF__774", "SF__775", "SF_1001", "SF_1553", "SF_1642",
            "SF_1643", "SF_1644", "SF_1645", "SF_1675", "SF_1717", "SF_1718", "SF_1741",
            "SF_1753", "SF_1806", "SF_1810", "SF_1816", "SF_1819", "SF_1827", "SF_1828",
            "SF_1829", "SF_1830", "SF_2175", "SF_2176", "SF_2177", "SF_2178", "SF_2538";

        resList := call valueExtract with valuesId, symptoms_to_process;

    ;;
  evoke: /* direct call only */;;
  logic:
        /* set all symptoms to false which are null */
        result := ();
        for aSymptom in resList do
            if aSymptom is null then
                result := result, false;
            else
                result := result, aSymptom;
            endif;
        enddo;

        /* store results */
        foo := call valueStore with valuesId, result, symptoms_to_process;
        conclude true;;
    action:
        ;;
end:
```

C.4 Ratings: differences to printout

Table C.3: Detailed differences of intermediate combination ratings between original CADIAG-II system (O) and Arden Syntax-based system (A)

code	67002801		67008729		67009954		67013021		67018058		67019186		67021741	
	O	A	O	A	O	A	O	A	O	A	O	A	O	A
PT 4							1	false						
PT 5													0.1	false
PT 7							1	false						
PT 8					1	false	1	false						
PT 9							1	false						
PT 10							1	false						
PT 12			1	false	1	false								
PT 13	1	false									0.5	false		
PT 18					0.9	0.8	0.9	0.8						
PT 20							1	false						
PT 26									1	false				
PT 30							1	false						
PT 34	0.1	0.05			0.1	0.05					0.1	0.05		
PT 38							1	false						
PT 42					1	false	1	false						
PT 50							1	false						
PT 56			0.7	false										
PT 61			0.63	false									1	false
PT 62							1	false						
PT 110							1	false						
PT 111							1	false						
PT 113					1	false	1	false						
PT 114							1	false						
PT 115							1	false						
PT 116							1	false						
PT 117							1	false						
PT 120					1	false	1	false						
PT 121							1	false						

This table shows for each patient its intermediate combinations that differ from the results defined by the printout. The rating of each intermediate combination is shown as the original value (column 'O') and the computed one (column 'A'). Most differences are based on missing data in the data base whereas those entries marked with a red background color define differences resulting from differently computed diagnoses.

Table C.4: Detailed differences of diagnosis ratings (1)

arbnr	tot.	diffs	DF 1		DF 4		DF 35		DF 62		DF 73		DF 74		DF 79	
			O	A	O	A	O	A	O	A	O	A	O	A	O	A
67009954	8	6	0.9	0.8											0.9	null
67013021	8	5	0.9	0.8							0.9	0.8				
67018058	3	3											null	0.4		
67008729	10	3													0.9	null
67022098	11	2														
67021741	13	2									0.9	null				
67019186	14	2					0.5	0.2								
67011150	12	2					0.5	0.4								
67004316	10	2														
67003174	13	2			0.8	0.1										
67025526	16	1														
67025313	12	1														
67025178	13	1														
67024635	16	1														
67022357	7	1														
67022047	12	1														
67021172	9	1														
67020907	10	1														
67018317	16	1							0.0	0.5						
67017663	8	1														
67017019	6	1														
67016527	11	1														
67016233	9	1														
67014451	8	1														
67014028	7	1														
67013862	13	1														
67013773	15	1														
67012084	14	1														
67011789	11	1														
67011045	12	1														
67008095	16	1														
67006351	14	1														
67005460	14	1														
67005452	10	1														
67003905	12	1														
67002801	7	1					0.5	0.2								
67002119	10	1														
67001546	14	1														
67000566	11	1														

This table and the following two tables show, for each patient and each diagnosis hypothesis that has been rated differently by the Arden Syntax-based system, the original value (column 'O') and the computed one (column 'A'). Most differences are based on missing data in the data base.

Those entries marked with a red background color show differences that are the result of missing data in the data base. Those with a green background color indicate entities that were rated differently because of ignored symptom-combination-to-diagnosis relationships (compare section 4.3.2). Cells with a yellow background color markup differences that where caused by ignored diagnosis-to-diagnosis relationships. The three entries with a light blue background color show correctly computed values that, however, do not show up on the printout.

The seven patient-ids that are highlighted by blue cells are those which included differences in their intermediate combination ratings as shown earlier in table C.3.

Table C.5: Detailed differences of diagnosis ratings (2)

arbnr	DF 83		DF 84		DF 86		DF 87		DF 119		DF 137		DF 157		DF 161		DF 165		DF 176		DF 208		DF 529	
	O	A	O	A	O	A	O	A	O	A	O	A	O	A	O	A	O	A	O	A	O	A	O	A
67009954			0.41	0.02															0.9	0.01				
67013021	0.4	0.32																					0.9	0.8
67018058																							null	0.4
67008729															0.0	0.46								
67022098																	0.0	0.5			1	null		
67021741																							0.9	null
67019186																					1	null		
67011150															0.4	0.29								
67004316					0.0	0.43															1	null		
67003174											0.4	0.05												
67025526																					1	null		
67025313																					1	null		
67025178																					1	null		
67024635																					1	null		
67022357																					1	null		
67022047							0.0	0.5																
67021172															0.0	0.46								
67020907																					1	null		
67018317																								
67017663							0.0	0.5																
67017019															0.0	0.46								
67016527					0.0	0.43																		
67016233																					1	null		
67014451																					1	null		
67014028					0.0	0.43																		
67013862																	0.0	0.5						
67013773																					1	null		
67012084																					1	null		
67011789																					1	null		
67011045																					1	null		
67008095					0.0	0.43																		
67006351													0.8	0.15										
67005460					0.0	0.4																		
67005452							0.0	0.5																
67003905									0.65	0.05														
67002801																								
67002119															0.0	0.46								
67001546																					1	null		
67000566																					1	null		

Table C.6: Detailed differences of diagnosis ratings (3)

arbnr	DF 530		DF 532		DF 538		DF 539	
	O	A	O	A	O	A	O	A
67009954	0.9	null					0.9	0.8
67013021			0.9	0.8				
67018058					null	0.4		
67008729	0.9	null						
67022098								
67021741								
67019186								
67011150								
67004316								
67003174								
67025526								
67025313								
67025178								
67024635								
67022357								
67022047								
67021172								
67020907								
67018317								
67017663								
67017019								
67016527								
67016233								
67014451								
67014028								
67013862								
67013773								
67012084								
67011789								
67011045								
67008095								
67006351								
67005460								
67005452								
67003905								
67002801								
67002119								
67001546								
67000566								

C.5 Result XML

The result of the Arden Syntax based inference process may either be a plain text file, which is stored into the file system, or may be displayed on the screen as an XML file. The document type definition (DTD) and a sample XSLT stylesheet to transform the XML file into HTML are shown next.

C.5.1 DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT CADIAG (DIAGNOSES, SYMPTOM_COMB, INTERM_COMB, SYMPTOMS)>
<!ATTLIST CADIAG
    id CDATA #REQUIRED
>

<!ELEMENT DIAGNOSES (TRUE, FALSE, NULL, OMEGA)>
<!ELEMENT SYMPTOM_COMB (TRUE, FALSE, NULL, OMEGA)>
<!ELEMENT INTERM_COMB (TRUE, FALSE, NULL, OMEGA)>
<!ELEMENT SYMPTOMS (TRUE, FALSE, NULL, OMEGA)>

<!ELEMENT TRUE (ENTRY+)>
<!ELEMENT FALSE (ENTRY+)>
<!ELEMENT NULL (ENTRY+)>
<!ELEMENT OMEGA (ENTRY+)>

<!ELEMENT ENTRY (LABEL, VALUES)>

<!ELEMENT LABEL (TEXT+)>
<!ATTLIST LABEL
    code   (PCDATA) #REQUIRED
    cadiag (PCDATA) #REQUIRED
>

<!ELEMENT TEXT (#PCDATA)>
<!ATTLIST TEXT
    lang (DE | EN) #REQUIRED
>

<!ELEMENT VALUES (VALUE)>
<!ATTLIST VALUES
    score  (PCDATA) #IMPLIED
>
<!ELEMENT VALUE (#PCDATA)>
```

C.5.2 XSLT stylesheet

Basically, all sections of a result file (diagnoses, symptoms, combinations) are processed identically. Within one section, discrepancies and unknown entries are filtered out (by comments). Rows (entries) are alternately colored by two colors which are defined in a separate CSS file (see template for “ENTRY”).

Additionally, diagnoses are sorted by their value and are displayed with their scores.

```
<?xml version="1.0"?> <xsl:stylesheet version="1.0"
```

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
    <html>
        <head>
            <title>XSLT output</title>
            <link rel="stylesheet" type="text/css"
            href="http://146.254.106.170:8088/arden/arden.css"/>
        </head>
        <body>
            <xsl:apply-templates/>
        </body>
    </html>
</xsl:template>

<xsl:template match="CADIAG">
    <h1>Medizinisches Expertensystem CADIAG-II/ARDEN</h1>
    <h2>Rheumatologische Erkrankungen (RSKA Baden)</h2>
    <h3>Ergebnisse fuer Patient <xsl:value-of select="@id"/></h3>
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="SYMPTOMS">
    <h3>Symptome</h3>
      <xsl:apply-templates/>
    <hr/>
</xsl:template>

<xsl:template match="DIAGNOSES">
    <h3>Diagnosen</h3>
    <xsl:apply-templates>
        <xsl:with-param name="sort">true</xsl:with-param>
    </xsl:apply-templates>
    <hr/>
</xsl:template>

<xsl:template match="INTERM_COMB">
    <h3>Zwischenkombinationen</h3>
      <xsl:apply-templates/>
    <hr/>
</xsl:template>


<xsl:template match="SYMPTOM_COMB">
    <h3>Symptomkombinationen</h3>
      <xsl:apply-templates/>
    <hr/>
</xsl:template>

<xsl:template match="TRUE">
    <xsl:param name="sort"/>
    <h4>Hypothesen:</h4>
    <table>
        <xsl:choose>
            <xsl:when test="$sort='true'">
                <xsl:apply-templates select="ENTRY">
```

```
                    <xsl:sort select="VALUES/@score"
                         order="descending" data-type="number"/>
                </xsl:apply-templates>
            </xsl:when>
            <xsl:otherwise>
                <xsl:apply-templates/>
            </xsl:otherwise>
        </xsl:choose>
    </table>
</xsl:template>

<xsl:template match="FALSE">
    <h4>Ausgeschlossen:</h4>
    <table>
        <xsl:apply-templates/>
    </table>
</xsl:template>

<xsl:template match="NULL">
    <!--<h4>Unbekannt:</h4>
    <!--<table>
        <xsl:apply-templates/>
    </table>-->
</xsl:template>

<xsl:template match="OMEGA">
    <!--<h4>Widersprueche:</h4>
    <table>
        <xsl:apply-templates/>
    </table>-->
</xsl:template>

<xsl:template match="ENTRY">
  <xsl:choose>
    <xsl:when test="VALUES/@score">
       <xsl:if test="number(VALUES/@score) &gt; 0.5 or
                     VALUES/child::node()[position()=1] = 'false' ">
          <xsl:if test="number(VALUES/child::node()[position()=1]) &gt; 0.39 or
                        VALUES/child::node()[position()=1] = 'false'">
            <xsl:if test="position() mod 2 = 0">
                <tr class="BodyTableRow1">
                    <td>
                        <xsl:apply-templates select="VALUES"> </xsl:apply-templates>
                    </td>
                    <xsl:apply-templates select="LABEL">
                        <xsl:with-param name="cssclass">BodyTableRow1Colors1
                        </xsl:with-param>
                    </xsl:apply-templates>
                </tr>
            </xsl:if>
            <xsl:if test="position() mod 2 = 1">
                <tr class="BodyTableRow0">
                    <td>
                        <xsl:apply-templates select="VALUES"/>
                    </td>
                    <xsl:apply-templates select="LABEL">
```

```
                        <xsl:with-param name="cssclass">BodyTableRow0Colors1
                        </xsl:with-param>
                    </xsl:apply-templates>
                </tr>
            </xsl:if>
          </xsl:if>
        </xsl:if>
    </xsl:when>
    <xsl:otherwise>
        <xsl:if test="position() mod 2 = 0">
            <tr class="BodyTableRow1">
                <td>
                    <xsl:apply-templates select="VALUES">
                    </xsl:apply-templates>
                </td>
                <xsl:apply-templates select="LABEL">
                    <xsl:with-param name="cssclass">BodyTableRow1Colors1</xsl:with-param>
                </xsl:apply-templates>
            </tr>
        </xsl:if>
        <xsl:if test="position() mod 2 = 1">
            <tr class="BodyTableRow0">
                <td>
                    <xsl:apply-templates select="VALUES"/>
                </td>
                <xsl:apply-templates select="LABEL">
                    <xsl:with-param name="cssclass">BodyTableRow0Colors1</xsl:with-param>
                </xsl:apply-templates>
            </tr>
        </xsl:if>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="VALUES">
    <xsl:choose>
        <xsl:when test="child::node()[position()=1]='true'">+</xsl:when>
        <xsl:when test="child::node()[position()=1]='false'">-</xsl:when>
        <xsl:when test="child::node()[position()=1]='null'">?</xsl:when>
        <xsl:otherwise><xsl:value-of select="child::node()[position()=1]"/>
        </xsl:otherwise>
    </xsl:choose>
    <xsl:if test="./@score">
        (<xsl:value-of select="round(number(@score)*100)"/>)
    </xsl:if>
</xsl:template>

<xsl:template match="LABEL">
    <xsl:param name="cssclass"/>
    <td>
        <xsl:apply-templates>
            <xsl:with-param name="cssclass"><xsl:value-of select="$cssclass"/>
            </xsl:with-param>
        </xsl:apply-templates>
    </td>
    <td><xsl:value-of select="@code"/></td>
```

```
    <td><xsl:value-of select="@cadiag"/></td>
</xsl:template>

<xsl:template match="TEXT">
    <xsl:param name="cssclass"/>
    <table>
        <xsl:if test="$cssclass = 'BodyTableRow1Colors1'">
            <tr class="BodyTableRow1">
                <xsl:if test="@TOPIC != ''">
                    <td class="BodyTableRow1Colors1" style="white-space:nowrap" >
                        <xsl:value-of select="@TOPIC"/>
                    </td>
                </xsl:if>
                <td>
                    <xsl:value-of select="node()"/>
                </td>
            </tr>
        </xsl:if>
        <xsl:if test="$cssclass = 'BodyTableRow0Colors1'">
            <tr class="BodyTableRow0">
                <xsl:if test="@TOPIC != ''">
                    <td class="BodyTableRow0Colors1" style="white-space:nowrap">
                        <xsl:value-of select="@TOPIC"/>
                    </td>
                 </xsl:if>
                <td>
                    <xsl:value-of select="node()"/>
                </td>
            </tr>
        </xsl:if>
    </table>
</xsl:template> </xsl:stylesheet>
```

C.5.3 Benchmark MLMs

C.5.3.1 Lookup values in list – main MLM

The main MLM uses different settings to benchmark the performance of intermediate variables. Two MLMs that provide the lookup method are referenced and used for every setting.

The first MLM that is implemented in pure Arden Syntax accepts a list of intermediate variable labels, a list of the corresponding values, and a list of labels of those variables that have to be looked up. The second one that uses a Java class to implement the value storage accepts the id as explained in section 3.2.6 and a list of labels of those variables that have to be looked up.

```
maintenance:
  title: CADIAG test lookup data in lists;;
  mlmname: LookupTest;;
  arden: Version 2f;;
  version: $Id: lookuptest.mlm,v 1.1 2002/09/10 07:27:16 tiffsvgt Exp $;;
  institution: Siemens Medical Solutions, University of Vienna;;
  author: ;;
  specialist: ;;
```

```
  date: 2001-08-22;;
  validation: testing;;

library:
  purpose: ;;
  explanation: ;;
  keywords:  ;;
  citations: ;;
  links:  ;;

knowledge:
  type: goal-directed;;
  data:
    event1       := event {cadiagLookupTest};
    message_send := mlm 'message_send';
    dest1        := destination {console:log};

    /* storage and database tools */
    valueStorageInit := mlm 'storage_init';
    valueStore       := mlm 'value_store';

    testit  := mlm 'LookupTest2';
    testit2 := mlm 'LookupTest3';
  ;;
  evoke:
    event1;;
  logic:
    /* list of all intermediate variable identifiers
       (about 2500 string constants) */
    labels := "AA07AACA", "AA07AACT", ...;

    /* list of integer numbers that stand for the intermediate variable values */
    values := (); i := 1;
    for aLabel in labels do
        values := values, i;
        i:=i+1;
    enddo;

    /* initialize database storage for DB performance test*/
    valuesId := call valueStorageInit;
    foo := call valueStore with valuesId, values, labels;


    /* lookup five values by linear search algorithm (Arden Syntax) */
    /* first five values */
    testlist := "AA07AACA", "AA07AACT", "AA07AAFAC", "AA07AAFAE", "AA07AAFAF";
    results := call testit with labels, values, testlist;

    /* last five values */
    testlist := "SF____5", "SF____6", "SF____7", "SF____8", "SF____9";
    results := call testit with labels, values, testlist;

    /* lookup five values by interface (Java class) */
    /* first five values */
    testlist := "AA07AACA", "AA07AACT", "AA07AAFAC", "AA07AAFAE", "AA07AAFAF";
    results  := call testit2 with valuesId, testlist;
```

```
    /* last five values */
    testlist := "SF____5", "SF____6", "SF____7", "SF____8", "SF____9";
    results  := call testit2 with valuesId, testlist;


    /* prepare list of entities used by intermediate combinations
       (327 values) */
    testlist := "DF____1", "DF___52", "DF__107", ...;

    /* lookup values by linear search algorithm (Arden Syntax) */
    results := call testit with labels, values, testlist;

    /* lookup values by interface (Java class) */
    results := call testit2 with valuesId, testlist;


    /* prepare list of entities used by rules_s2d_compact
       (1023 values) */
    testlist := "SF____1", "SF____2", "SF____3", ...;

    /* lookup values by linear search algorithm (Arden Syntax) */
    results := call testit with labels, values, testlist;

    /* lookup values by interface (Java class) */
    results := call testit2 with valuesId, testlist;

    conclude true;;
  action:
   ;;
end:
```

C.5.3.2 Lookup values in list – linear search MLM

```
maintenance:
  title: _CADIAG test lookup data in lists;;
  mlmname: LookupTest2;;
  arden: Version 2f;;
  version: $Id: lookuptest2.mlm,v 1.1 2002/09/10 07:27:16 tiffsvgt Exp $;;
  institution: Siemens Medical Solutions, University of Vienna;;
  author: ;;
  specialist: ;;
  date: 2001-08-22;;
  validation: testing;;

library:
  purpose: ;;
  explanation: ;;
  keywords:  ;;
  citations: ;;
  links:  ;;

knowledge:
  type: goal-directed;;
  data:
    (labels, values, lookup) := argument;
```

```
  ;;
  evoke: ;;
  logic:
  result := ();
    for aLabel in lookup do
        notfound := true;
        elements := count labels;
        i := 1;

        while (notfound = true and i<= elements) do
            if aLabel = labels[i] then
                notfound := false;
            else
                i := i + 1;
            endif;
        enddo;

        if (notfound = false) then
            result := result, values[i];
        else
            result := result, null;
        endif;

    enddo;

    conclude true;;
  action:
    return result;
   ;;
end:
```

The algorithm could be improved by a binary search, if the list of labels would be ordered. As the list is generally not ordered (and the option to use the sort operator is not provided, since the corresponding values in the 'values list' cannot be re-sorted analogously) a linear search algorithm has been used.

C.5.3.3 Lookup values in list – Java interface MLM

```
maintenance:
  title: _CADIAG test lookup data in lists;;
  mlmname: LookupTest3;;
  arden: Version 2f;;
  version: $Id: lookuptest2.mlm,v 1.1 2002/09/10 07:27:16 tiffsvgt Exp $;;
  institution: Siemens Medical Solutions, University of Vienna;;
  author: ;;
  specialist: ;;
  date: 2001-08-22;;
  validation: testing;;

library:
  purpose: ;;
  explanation: ;;
  keywords:  ;;
  citations: ;;
  links:  ;;
```

```
knowledge:
  type: goal-directed;;
  data:
    valueStorage_get := interface
        { class;de.medstage.projects.cadiag.ardenifc.ValueStorage;get };

    (valuesId, testlist) := argument;
  ;;
  evoke: ;;
  logic:
    result := call valueStorage_get with valuesId, testlist;
    conclude true;;
  action:
    return result;
   ;;
end:
```

Appendix D

MLM XML representation (DTD)

This is the document-type definition used for the XML representation by the rules engine. It is transformed by an XSLT stylesheet into HTML. The XSLT file is not shown, as...

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- doctype mlm ArdenSyntaxMlm.dtd   -->
<!ELEMENT MLM (CATEGORY+)>
<!ELEMENT CATEGORY (SLOT)+>
<!ATTLIST CATEGORY
    name CDATA #REQUIRED
>
<!ELEMENT SLOT (#PCDATA | CODEBLOCK)*>
<!ATTLIST SLOT
    type (coded | textual | structured) #REQUIRED
    name CDATA #REQUIRED
>
<!ELEMENT CODEBLOCK (STATEMENT)*>
<!ELEMENT STATEMENT (IDENTIFIER | DATA | OPERATOR | CODEBLOCK)*>
<!ATTLIST STATEMENT
    name (assignment | ifthen | while | for | conclude | write | call
                     | evoke | return) #REQUIRED
>
<!ELEMENT OPERATOR (OPNAME?, PARAMETER, OPNAME?, PARAMETER?, OPNAME?)+>
<!ELEMENT PARAMETER (IDENTIFIER | DATA | OPERATOR)*>
<!ELEMENT OPNAME (#PCDATA)>
<!ELEMENT IDENTIFIER (#PCDATA | IDENTIFIER)*>
<!ATTLIST IDENTIFIER
    type (group | single) #REQUIRED
>
<!ELEMENT DATA (ORDERED | TRUTHVALUE | LIST | INTERNAL | REFERENCE)>
<!ELEMENT LIST (ORDERED | TRUTHVALUE)*>
<!ELEMENT ORDERED (#PCDATA)>
<!ATTLIST ORDERED
    type (number | time | duration | string | null) #REQUIRED
>
<!ELEMENT TRUTHVALUE (#PCDATA)>
<!ATTLIST TRUTHVALUE
    type (crisp | fuzzy) "crisp"
>
<!ELEMENT INTERNAL (#PCDATA)>
<!ATTLIST INTERNAL
```

```
    type (readmapping | destination | term | argument | null | now |
          eventtime | triggertime | patientid) #REQUIRED
>
<!ELEMENT REFERENCE (MLMNAME, INSTNAME?)>
<!ATTLIST REFERENCE
    type (mlm | lv) "mlm"
>
<!ELEMENT MLMNAME (#PCDATA)>
<!ELEMENT INSTNAME (#PCDATA)>
```

Appendix E

UltraEdit syntax highlighting scheme

To edit Medical Logic Modules the 'UltaEdit' editor has been used. This text editor supports syntax highlighting schemes that can be customized by editing a word-list file.

```
/L6"Medical Logic Module"
Line Comment = //
Block Comment On = /* Block Comment Off = */
File Extensions = MLM

/Delimiters = ~!@$%^&*()+=|\/{}[];"'<>  ,.?/
/Indent Strings = "{"
/Unindent Strings = "}"
/Function String = "%[ ^t]++sub[ ^t{]"

/C1 end: knowledge: library: maintenance:
/C2 action: author: citations: data: date: evoke: explanation:
filename: institution: keywords: links: logic: purpose:
specialist: title: type: validation: version: mlmname:
arden: author: action:
```