Artificial Intelligence in Medicine 49 (2010) 1–10

Contents lists available at ScienceDirect

Artificial Intelligence in Medicine

journal homepage: www.elsevier.com/locate/aiim

Fuzzy Arden Syntax: A fuzzy programming language for medicine

Thomas Vetterlein [a,*], Harald Mandl [b], Klaus-Peter Adlassnig [a,b]

[a] Section for Medical Expert and Knowledge-Based Systems, Medical University of Vienna, Spitalgasse 23, 1090 Vienna, Austria
[b] Medexter Healthcare GmbH, Borschkegasse 7/5, 1090 Vienna, Austria

ARTICLE INFO

Article history:
Received 6 March 2009
Received in revised form 15 December 2009
Accepted 8 January 2010

Keywords:
Clinical decision support system
Arden Syntax
Fuzzy Arden Syntax
Fuzzy set theory

ABSTRACT

Objective: The programming language Arden Syntax has been optimised for use in clinical decision support systems. We describe an extension of this language named Fuzzy Arden Syntax, whose original version was introduced in S. Tiffe's dissertation on "Fuzzy Arden Syntax: Representation and Interpretation of Vague Medical Knowledge by Fuzzified Arden Syntax" (Vienna University of Technology, 2003). The primary aim is to provide an easy means of processing vague or uncertain data, which frequently appears in medicine.
Methods: For both propositional and number data types, fuzzy equivalents have been added to Arden Syntax. The Boolean data type was generalised to represent any truth degree between the two extremes 0 (falsity) and 1 (truth); fuzzy data types were introduced to represent fuzzy sets. The operations on truth values and real numbers were generalised accordingly. As the conditions to decide whether a certain programme unit is executed or not may be indeterminate, a Fuzzy Arden Syntax programme may split. The data in the different branches may be optionally aggregated subsequently.
Results: Fuzzy Arden Syntax offers the possibility to formulate conveniently Medical Logic Modules (MLMs) based on the principle of a continuously graded applicability of statements. Furthermore, ad hoc decisions about sharp value boundaries can be avoided. As an illustrative example shows, an MLM making use of the features of Fuzzy Arden Syntax is not significantly more complex than its Arden Syntax equivalent; in the ideal case, a programme handling crisp data remains practically unchanged when compared to its fuzzified version. In the latter case, the output data, which can be a set of weighted alternatives, typically depends continuously from the input data.
Conclusion: In typical applications an Arden Syntax MLM can produce a different output after only slight changes of the input; discontinuities are in fact unavoidable when the input varies continuously but the output is taken from a discrete set of possibilities. This inconvenience can, however, be attenuated by means of certain mechanisms on which the programme flow under Fuzzy Arden Syntax is based. To write a programme making use of these possibilities is not significantly more difficult than to write a programme according to the usual practice.

1. Introduction

In medical computer science, intelligent management of clinical data is the objective of what is known as clinical decision support, or CDS for short. The performance of a CDS system (CDSS) clearly extends the mere storage and user-specified manipulation of patient data. Its essential feature is the ability to provide consequences of the digitally handled medical information, derived on the basis of a medical knowledge base. A logical inference module provides methods of deducing consequences from the available data in accordance with coded interrelations between medical facts.

We deal here with an established programming language designed especially for computerised analysis of medical data. The language is named Arden Syntax and has been used for fifteen years now; it was first published by the American Society for Testing and Materials in 1992. The language has been developed since. The present paper refers to Arden Syntax 2.5, which was published by Health Level Seven, Inc., in 2005. Complete specifications can be found in [1], and further information on the subject is available on the Web, e.g., at http://www.hl7.org/implement/standards/ardensyntax.cfm (accessed on 10 October 2008).

Arden Syntax is a convenient tool for the implementation of a CDSS. The aim of an Arden-Syntax-based CDSS is to derive, from a patient's electronic record, information suitable to facilitate or improve clinical decision-making. The user might employ the CDSS by specifically calling some of its functions. However, in order to achieve optimal performance the system should run permanently in the background and notice in real time whether the available data imply the necessity of a consequence. In this case, a proposal or if necessary a warning is communicated to the clinical personnel. A

* Corresponding author. Tel.: +43 1 40400 6665; fax: +43 1 40400 6667.
E-mail address: Thomas.Vetterlein@meduniwien.ac.at (T. Vetterlein).

0933-3657/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.artmed.2010.01.003

CDSS should not merely perform routine tasks, but give discrete hints as to what should or should not be done. One of the primary objectives is to avoid the mistakes that become evident from the available data, provided the appropriate knowledge has been entered previously.

In principle an inference mechanism based on medical knowledge may be coded in any programming language. However, Arden Syntax is equipped with several features which were specifically chosen for the intended application: (i) a syntax close to natural language, which renders a programme to a large extent readable by the non-expert in computer science; (ii) thorough choice of just a few data types typically needed for medical documentation, including time and duration; (iii) the possibility to process and to communicate real-time events, determined according to the host system specifications; (iv) easy handling of temporal relations, supported in particular by a time component attached to all simple data types; (v) high error tolerance of Arden Syntax programmes, selecting, in case of doubt, the option most likely meant by the user, rather than interrupting the work flow every time by an error message.

The function of an Arden-Syntax-based system is to check a patient's accessible digital data with regard to specific conditions and react appropriately if the conditions apply. Needless to say, medical data do not always permit a clear distinction between whether a specific condition is fulfilled or not; borderline cases defy this stipulation. When the assumptions of an implication are approximately but not entirely fulfilled, it is, from a strict point of view, not possible to draw any conclusion. However, a conclusion can well be desirable. This situation suggests the implementation of methods provided by fuzzy logic. In fuzzy logic, a conclusion is permitted, provided that appropriate modifications are applied to the result: the outgoing statement is weakened with regards to content. The simplest way to realise this idea is to assign a weight to the outcome. In case of more than one possible outcome, several distinct outputs might result, all weighted with a value strictly smaller than 1.

Based on these considerations, a conservative extension of Arden Syntax was proposed by S. Tiffe in his Ph.D. thesis [2]. The principle is to generalise classical two-valued logic to a many-valued logic; the extension is named Fuzzy Arden Syntax. The work presented here has taken up this line and led to an alternative version of Fuzzy Arden Syntax, whose full specification can be found in [3]. A comparison to Tiffe's work is contained in Section 11.

2. Fuzzy logic to process vague information

The features which have been added to Arden Syntax aim at simplifying programmes which process indeterminate data by means of fuzzy logic. Let us review very shortly the significance of this methodology.

As a starting point for the development of fuzzy logic, often Zadeh's article in 1965 is seen [4]. The field has since then developed enormously and into several different directions. Among the numerous general monographs on the topic we may recommend [5].

The main idea is to generalise the notion of a truth value. Rather than using two truth values 0 and 1, modelling "false" and "true", respectively, typically the elements of the entire real unit interval $[0,1]$ are used. When a truth value is associated with a vague statement, it expresses the degree to which a proposition is true. In particular, 0 expresses the clear falsity while 1 expresses the clear truth of a proposition; anything between the two just expresses a tendency.

Fuzzy logic is at present widely used for controlling tasks and in automated decision support. Its use is implied in all cases in which the precise specification of a problem is not possible or too complex, where, however, a specification in vague terms is available and assumed to be fully sufficient for the practical needs. In controlling applications, for example, instruction sets are often available in natural language. A standard method to use these instructions directly, dealing with the involved vagueness in an amazingly efficient way, is due to Mamdani and Assilian [6]. Variants of the Mamdani–Assilian controller exist, according to one of which, e.g., the system FuzzyKBWean [7] is based, which provides recommendations to the clinical personnel of an optimal choice of instrument parameters when weaning off an intensive care patient from intubation. In CDS, a problem may be to specify a pathological state on base of the lower and upper boundaries of laboratory values. The use of fuzzy sets rather than sharp intervals prevents the otherwise inevitable discontinuities of such specifications. Boundary cases are treated appropriately and jumps of the output by arbitrary small change of the input are avoided. The medical expert systems CADIAG are an example; see, e.g., [8].

By default, fuzzy logic is rather specific about the type of truth degrees dealt with. A truth value in fuzzy logic does not represent uncertainty in the sense of lack of knowledge. It rather represents to what degree some well-specified fact, represented, e.g., by a real number, is in accordance with a vague notion, like, e.g., a natural language expression. For instance, a body temperature of 38.5° is in partial, but not full accordance with the concept of "high fever", and could be described by a value, say, 0.3. The statement "high fever" associated with a truth value 0.3 is sharply to be distinguished from the statement that the patient has high fever with the probability 0.3; in the former case, uncertainty is not involved.

In the present context, however, we will adopt a broader viewpoint about the nature of truth degrees. In medicine, values of the real unit interval are frequently used not only to express degrees of applicability of a vague concept, but also of uncertainty, or more specifically the probability of some event. In Fuzzy Arden Syntax, the meaning of truth values is not predefined; we do not intend to prevent the user from using a probabilistic interpretation or an interpretation according to some more general framework of uncertainty management. We will actually in the sequel explicitly include this possibility. We note, however, that Fuzzy Arden Syntax does not contain special functions supporting reasoning under uncertainty; after all its design aims at supporting the procession of vague rather than uncertain information. To include probability theory or some kind of plausibility logic could motivate a further extension of the language.

We note that fuzzy analoga have been proposed for several programming languages. The aim has been to simplify inferences based on methods of fuzzy logic. In particular, the language HALO [9] has been described as "fuzzy Pascal". The language FRIL [10] extends Prolog, so as to deal with uncertainty of data and rules; similar aims are followed in [11]. The programming language **L**[12] does not refer to an existing language, but takes up the concept of *W*-recursiveness, which is an adaption of the notion of recursiveness for fuzzy functions. Active research is still being conducted in this field; a recent paper [13] could well be added to the list.

On the very basic level, Fuzzy Arden Syntax is comparable to these programming languages. Apart from that, however, the intentions are not comparable. Recall that Arden Syntax provides programmes which read, in the ideal case, like a specification of a medical procedure proposed under certain circumstances. Our aim was not only to provide fuzzy equivalents for the standard data types found in any language; we were primarily interested in rendering more flexible those components which were introduced into Arden Syntax in order to adapt it for use in medicine.

3. Arden Syntax

Fuzzy Arden Syntax is based on Arden Syntax and every programme written in Arden Syntax runs under Fuzzy Arden Syntax without any alteration of its effect. That is, Fuzzy Arden

Syntax is backward compatible with Arden Syntax. We tried to keep all changes to the necessary minimum. The official Arden Syntax specifications can be found in [1]. It would be impossible to enumerate them here. Nevertheless, we intend to make the main features of Fuzzy Arden Syntax comprehensible even to those without a profound knowledge of HL7's Arden Syntax. To this end we will explain a few basic principles of Arden Syntax and provide a sample programme. Moreover, when presenting Fuzzy Arden Syntax we will provide integrated specifications.

As mentioned earlier, Arden Syntax is typically used to realise a CDSS. An Arden-Syntax-based CDSS is highly modular; it is composed of a possibly high number of programming units named Medical Logic Modules, or MLMs for short. The set of MLMs is not hierarchically structured; each MLM may be considered and can be read and understood independently. However, MLMs may call each other, such that some MLMs may be considered subroutines of other MLMs. An MLM may take over data and return data. The execution of an MLM may be dependent on a specific event communicated from the outside.

A host system provides the user interface, access to the patient data base, and control over the execution of MLMs. All requests originating from the user or an MLM are processed by the host. The host specifies a set of events and communicates the occurrence of any event to the MLMs concerned. An event may reflect, for instance, a specific change in the patient database.

A typical MLM refers to a specific event which may take place at any time in the database. Whenever this event occurs, the MLM is triggered and causes, or reminds the users to take, the appropriate action.

This phenomenon is demonstrated by one of the sample MLMs from [1]. The MLM `Penicillin_Allergy` is executed whenever a penicillin is prescribed at the hospital where the system is in use or, more precisely, when the corresponding change is made in the database. A warning is issued when an allergy against penicillin is registered for the respective patient. What follows is a brief version of the MLM which can be found in [1, X3.3].

```
maintenance:
  mlmname:  Penicillin_Allergy;;
  arden:  version 2.5;;
  institution:
    Columbia-Presbyterian Medical Center;;
  [...]
library:
  [...]
knowledge:
  [...]
  data:
    Penicillin_Order := event
      {Medication_Order (Penicillin)};
    Penicillin_Allergy :=
      read last {Allergy (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;
    ;;
  [...]
end:
```

An MLM consists of three main parts, named categories, each of which consists of several entries, named slots. The first two categories, namely `maintenance` and `library`, contain metadata, of which only the content of three slots are of interest to the programmer. These are shown in the above example and are presumably self-explanatory.

The programme itself is located in the last category named `knowledge`. In the `knowledge` category, there are always a data, a logic, and an action slot, which are executed in this order, as well as an evoke slot. In the data slot of our example, we first find the event `Penicillin_Order` defined, which occurs whenever penicillin is ordered. The definition of an event is given within curly brackets; this part is interpreted by the host system and the syntax is specific for the medical institution mentioned in the maintenance category. The next line contains a database query; the `read` command causes a list to be read containing the notifications of an allergy of the patient to penicillin. The result is stored in the variable `Penicillin_Allergy`. The effect of the key word `last` is that all entries but the last one are deleted from the list. Note that in the present case, only one entry is needed.

The events evoking this MLM are specified in the evoke slot. In this example, it is the event `Penicillin_Order`, which has been defined in the data slot.

The logic slot contains the part of the programme whose purpose is to decide whether the subsequent action slot will be executed or not. A command of the form `conclude ActionCondition`, where `ActionCondition` is some Boolean variable, contains the decision: if `ActionCondition` is true, the action slot is performed; otherwise the MLM is terminated at this point.

In the above example, the programme first checks whether the list `Penicillin_Allergy` is empty. If this is not the case the command `conclude true` is executed, meaning that the programme jumps to the beginning of the action slot.

The action slot defines the action to be taken, provided that the need for an action has been identified in the logic slot. In the above example, a warning message is sent to the host, which will be displayed on the user's interface.

4. The concept underlying Fuzzy Arden Syntax

Fuzzy Arden Syntax incorporates new concepts into Arden Syntax in order to assist in processing information that may not be completely determinate. It is based on the observation that, in medicine, we typically draw conclusions from real parameters in a way that the exact values do not really matter. Besides, we frequently infer information from facts which are not seen to be clearly true, but are considered true to a certain degree which high enough to allow a conclusion.

In contrast, a typical MLM written in Arden Syntax does depend on exact values and only deals with the two truth values "true" and "false". Consequently, a small change of a parameter may cause a sharp change in an outgoing recommendation; for, if a recommendation depends on a real value, a limit value must be chosen and this limit is typically chosen ad hoc because a precise limit point is rarely available.

A solution would be to specify a limit point in a rough manner rather than precisely, and issue differentiated recommendations in borderline cases. Similarly, statements which merely express a tendency rather than a simple "true" or "false" should also be permitted. The new features of Fuzzy Arden Syntax simplify the integration of such elements.

To illustrate the problem, let us consider a further example of an MLM written in Arden Syntax. The MLM `UTI_SUTI` is based on the specification of nosocomial infections by the US-American Centers for Disease Control and Prevention [14]; it determines whether a hospitalised patient has a symptomatic urinary tract infection, or

SUTI for short. This MLM, which is part of the MONI system [15], is designed as a subroutine of another MLM; it expects two input parameters, namely a code for the patient's hospital stay, which is stored in the variable `Stay`, and a date, stored in the variable `Day`. It yields a Boolean truth value indicating whether a SUTI is present or not.

```
maintenance:
  mlmname:  UTI_SUTI;;
  arden:  version 2.5;;
  [...]
knowledge:
  [...]
  data:
    (Stay, Date) := argument;
    Temperature :=
      read {temp (Stay, Date)};
      /* Body temperature. */
    if Temperature >= 38
      then Fever := true;
    endif;
    Urgency :=
      read {urge_urinate (Stay, Date)};
      /* Urge to urinate? */
    Micturition :=
      read {mict (Stay, Date)};
      /* Increased frequency of urination? */
    Dysuria := read {dys (Stay, Date)};
        /* Painful urination? */
    Suprapubic_tenderness :=
      read {suprtend (Stay, Date)};
      /* Suprapubic tenderness? */
    Organ_urine_culture :=
      read {org_urine_cult (Stay, Date)};
      /* Number of microorganisms of ≤ 2 species. */
    if Organ_urine_culture >= 1e5


      /* if number of species is ≥ 10^5/cm^3 */
then Urine_culture := true;
    endif;
    ;;
  evoke:
    ;;
  logic:
    UTI_SUTI :=   (Fever OR Urgency OR
          Micturition OR Dysuria OR
          Suprapubic_tenderness)
          AND Urine_culture;
    conclude true;
    ;;
  action:
    return UTI_SUTI;
    ;;
end:
```

The result, which indicates whether a UTI-SUTI infection is present, depends on two real parameters and four yes–no questions. Vagueness is involved in all cases. For instance, in order to check whether `Fever` is true, the patient's body temperature must be compared with a sharp value simply because this variable is two-valued; the limit value depends on an ad hoc choice. With regard to the variable `Dysuria`, the patient is required to specify whether or not the condition is present and this may be difficult in the borderline case. The final result may be negative but would well have been positive in case of a slight change in the manner of determining the input parameters.

The aim underlying the conception of Fuzzy Arden Syntax is to process, in a convenient way, statements that are not necessarily entirely false or entirely true, and thus render it unnecessary for an MLM's author or user to make odd decisions about borderline cases. The following principle is essential. To write or to understand a Fuzzy Arden Syntax programme should not cause essential difficulties for those already familiar with Arden Syntax. Certainly, the user must be aware of certain modifications. The program text should, however, remain practically unchanged when based on fuzzy logic rather than classical logic.

The main decisions underlying Fuzzy Arden Syntax have been made in accordance with fuzzy set theory and fuzzy logics. In particular, as a set of truth values, all (representable) reals between 0 and 1 are used. The MLM UTI_SUTI illustrates the usefulness of extending the set of truth values. We argue that for input parameters like `Dysuria` which are subjective in nature, it is more appropriate to use generalised truth values rather than two-valued ones. Similarly, for those input parameters which result from a measurement, such as `Temperature`, derived propositions like `Fever` should be considered fuzzy and assigned non-sharp truth values in borderline cases. To what extent does the MLM needs to be modified under Fuzzy Arden Syntax in order to incorporate these demands? As a matter of fact the programme can remain practically unchanged. All that needs to be specified is the degree of impreciseness, which is easily done by formulating the comparison involving body temperature as follows: "`Temperature >= 38.5 fuzzified by 0.5`". In Section 9 we will show what this example, when fuzzified, looks like in Arden Syntax and in Fuzzy Arden Syntax.

Let us now indicate the contexts in which fuzziness appears in Fuzzy Arden Syntax. The former data type `boolean` now takes values within the real unit interval [0,1]. If an if-then-else command depends on a variable containing, say, 0.7, the programme splits. Both the then-block and the else-block are executed: the former weighted 0.7, the latter weighted 0.3. The two branches may be reunified if the programmer wishes to do so. The variables are then aggregated, in a way that the weight of each branch is taken into account.

For every data type involving real values – number, time, duration – a fuzzy counterpart is added. Piecewise linear fuzzy sets over the respective base set can be represented. Distinctions like "the patient is young" or "middle-aged" or "old" can be made in a convenient way, namely without the necessity to specify sharp borders. Moreover, a fuzzy set can be defuzzified afterwards by means of a single command. For example, the behaviour of a Mamdani–Assilian controller can be achieved without noteworthy effort.

Finally, all non-compound data types are endowed with an additional component called the degree of applicability. This component stores a truth value expressing, for instance, the degree to which it would be reasonable to use the value in the variable's main component. It is 1 by default, and whenever the programme branches it is reduced automatically according to the weight assigned to the branch. The programmer may decide to make explicit use of this component but is not required to do so.

When processing vague data, one general problem inevitably encountered in fuzzy logic is the fact that there is no canonical choice for the connectives on the extended set of truth values. As we also cannot solve this problem, the programmer may choose between several options in Fuzzy Arden Syntax. Non-default options must be specified within a newly introduced category, named `fuzzy options`. The available options will be explained in the relevant context.

Let us finally consider the question how well a Fuzzy-Arden-Syntax-based CDSS might be accepted by the user. As a matter of fact, clinicians supporting the formulation of MLMs or using a fuzzy-logic-based environment will be asked for a slightly different point of view than before. In the process of writing an MLM, clinicians are asked to provide data together with an explicit

indeterminacy, like, e.g., fuzzy boundaries of laboratory parameters. Furthermore, the user of an MLM will be faced with the fact that in general more differentiated information is provided in the output than before. In practice this can mean that a set of weighted alternatives is presented as result, a possibly uncommon situation.

Note that both mentioned points actually represent a progress. Ad hoc decisions about exact boundaries of the normal range of some parameters are no longer necessary. Furthermore, complex results just occur in borderline situations; to provide a simplified output in such cases would hide the actual complexity of the situation.

In both cases, however, a minimal understanding of the nature of continuous truth degrees must be assumed. We are decidedly optimistic in this respect. We can rely on a rich experience in the cooperation with clinicians; the concept of a graded applicability is usually quite easily understood and the acceptance is high.

5. Data types in Fuzzy Arden Syntax

We will now begin describing the features of Fuzzy Arden Syntax in a systematic way. For additional information, we refer to [3].

Our first issue is to explain the data types. To maintain the clarity of the presentation, the data types incorporated already in Arden Syntax will *not* be separately marked.

There are simple and compound data types. In general, a variable is bound to a data type by assignment and a declaration is not necessary; thus Fuzzy Arden Syntax is loosely typed as is Arden Syntax. Furthermore, a variable may contain the value `null`. This is to reflect that the variable is undefined, as for instance when the defining expression is erroneous.

Every simple data type contains three components: the main value, the primary time, and the degree of applicability. The following explanations refer to the main component; the meaning of the other two will be explained later. Neither the second nor the third component needs to be explicitly addressed at any time; both can safely be ignored if not needed.

In Fuzzy Arden Syntax, the data type of propositional variables is denoted by `truth value` or – for reasons of backwards compatibility – equivalently `boolean`. A variable of this type stores real numbers between 0 and 1. We may write:

```
Var := truth value 0;
   or, equivalently,   Var := false;
Var := truth value 0.667;
Var := truth value 1;
   or, equivalently,   Var := true;
```

A variable of type `number` or, equivalently, `crisp number` stores real numbers which can be represented in the available floating point format. A variable of type `time` or `crisp time` stores a date, a time, and optionally the referred time zone. A variable of type `duration` or `crisp duration` stores a length in time. For example:

```
NumberVar := 3.4e-11;
TimeVar := 2007.11.15t10:43:02.347;
DurationVar := 3 months 4 days 4 minutes
                0.51 seconds;
```

We next turn to the fuzzy data types. The data type `fuzzy number` is dedicated to fuzzy sets over the reals. Needless to say, fuzzy sets cannot be allowed to have an arbitrarily complicated structure. This is no essential restriction. In medicine, as in most other applications, specific truth values do not have a specific meaning. Consequently, we may confine ourselves to fuzzy sets of a simple form. We will assume that we may partition the reals into a finite number of (possibly unbounded) intervals on each of which the fuzzy set is linear and continuous.

Formally, a fuzzy set $u : \mathbb{R} \to [0,1]$ can be stored in a variable of type `fuzzy number` if the following condition is met: There are $a_1 < a_2 < \ldots < a_k$, $k \geq 1$, in $\mathbb{R}$ such that u is linear on each open interval $(a_1, a_2), \ldots, (a_{k-1}, a_k)$, u is constant on $(-\infty, a_1)$ and (a_k, ∞), and for each $x \in \mathbb{R}$, $u(x)$ coincides either with the left limit or the right limit of u at x. If u is continuous, we then define

`Fuzzyset_u:= fuzzy set` $(a_1, t_1), (a_2, t_2), \ldots, (a_k, t_k)$;

where $t_i = u(a_i)$ for $i = 1, \ldots, k$. Thus, for instance, the fuzzy set

$$\mathbb{R} \to [0,1], \quad s \mapsto \begin{cases} 1 & \text{if} \quad s < 150 \\ \frac{1}{10}(160 - s) & \text{if} \quad 150 \leq s < 160 \\ 0 & \text{if} \quad s \geq 160, \end{cases}$$

expressing the predicate "small" for body size in centimetres, is defined by:

```
Small:= fuzzy set (150, 1), (160, 0);
```

Discontinuities are, as usual, allowed to include the characteristic functions, which are not likely to be required in applications, but should at least be definable. (Recall that characteristic functions correspond to subsets; it maps all elements of a set A to 1 and all elements of the complement of A to 0.)

At discontinuity points we denote the left as well as the right limit. The first assignment is taken to be the value at that point, unless the second one appears twice. For instance,

```
TwotoThree := fuzzy set (2, 0), (2, 1),
              (2, 1), (3, 1), (3,0);
```

gives the characteristic function of the set $[2,3] \subseteq \mathbb{R}$.

Furthermore, the triangular normal fuzzy sets are likely to appear frequently. This refers to those fuzzy sets whose graph forms a symmetrical triangle around one point, which is mapped to 1. A simplified notation is permitted for these: an expression of the form `fuzzy set` $(a-b, 0), (a, 1), (a+b, 0)$, where $a, b \in \mathbb{R}$ and $b > 0$, may also be written as:

a `fuzzified by` b

All definitions concerning the data type `fuzzy number` apply mutatis mutandis to the data types `fuzzy time` and `fuzzy duration` as well. The `fuzzified by` operator, for instance, may be used for pairs of a time and a duration; e.g., when referring to the time period approximately three days before the current time we may use the expression:

```
3 days ago fuzzified by 12 h
```

For the sake of completeness we mention that one more simple data type named `string` exists. A variable of this type stores text.

As mentioned earlier, two additional components are included in each simple data type. The so-called primary time is stored in the second component. This is typically the time at which the value emerged, which could be the time the value was measured. When reading data from the patient database this component may be filled automatically with a provided value; no explicit command is needed. Details are implementation-dependent, and the primary time is `null` if not specified.

The third component contains what we call the degree of applicability, or applicability for short. This is a truth value expressing the degree to which the main value may be considered applicable. Typical examples would be:

(i) We wish to derive a statement about the development of a quantity within the last 24 h. The limit of 24 h is chosen ad hoc and should be considered fuzzy. We may collect the values from the last, say, 28 h, and assign to each value a degree of

applicability of 1 if it originates from the preceding 24 h, else successively smaller values. The applicabilities may then be used to weight the values accordingly in the subsequent calculation.

(ii) In Fuzzy Arden Syntax a programme arriving at a point where the continuation depends on a condition which is neither clearly false nor clearly true, will split. Before executing one of the branches the set of variables is duplicated and the applicabilities are reduced according to the weights of the branches.

(iii) The degree of applicability is also at the user's free disposal. For instance, it may be attached to a variable in order to express that this value is not fully reliable.

The third component of a variable `Var` can be read and changed through the expression `applicability of Var`. By default it is set to 1 and it is never undefined.

Two compound data types exist: `list` and `object`. A list variable stores a sequence of n values of a simple data type, where $n \geq 0$ is dynamic. An object variable stores a fixed number of values of a simple data type; the components are denoted by specific identifiers, which must be declared in advance. For reasons which will become apparent in Section 10 below, the key word `object` may be replaced by `linguistic variable`.

6. Operations in Fuzzy Arden Syntax

The question as to how one should interpret the logical connectives for generalised truth values, has always been a delicate one. A generally valid recommendation cannot be made. In fact, it would even not be permissible to say that a specific type of application requires a specific set of connectives.

Only a few basic properties are usually required, namely those which are found to be natural for the respective connective. The conjunction is commonly interpreted by a function $\odot : [0,1]^2 \to [0,1]$ which is associative, commutative, neutral with respect to 1, and in both arguments isotone. Such a function is called a t-norm [16]. The disjunction is usually taken as the corresponding t-conorm, and the negation as the subtraction from 1. This standard is assumed in Fuzzy Arden Syntax as well; however, subroutines to interpret the connectives may also be defined by the programmer.

The three basic operations to combine truth values, are `and`, `or`, and `not`. To interpret `and`, the user may opt for the Łukasiewicz t-norm $\odot_{\mathrm{L}}$, the product t-norm $\odot_{\mathrm{P}}$, or the Gödel t-norm $\odot_{\mathrm{G}}$, where

$$a \odot_{\mathrm{L}} b = \max\{a + b - 1,\, 0\},$$
$$a \odot_{\mathrm{P}} b = a \cdot b,$$
$$a \odot_{\mathrm{G}} b = \min\{a, b\}$$

for $a, b \in [0,1]$. The choice is realised by an option in the fuzzy options category. By default, the Gödel t-norm is used. Furthermore, the `and` connective may be specified by the user, in which case the command

```
conjunction by
MLM 'UserspecifiedConjunction' ;
```

must appear. Here, `UserspecifiedConjunction` must be the name of an MLM which accepts two truth values as its input and returns one truth value as output. It is presumed that this MLM encodes a t-norm. As usual in Arden Syntax, however, it is not checked whether this MLM fulfils any of the special properties which are expected from a conjunction.

The interpretation of the disjunction and the negation may be chosen in a similar fashion, independent of the conjunction. However, we presume the user will make use of this freedom only in the exceptional case. By default the conjunction is the t-conorm $\oplus$ associated to the t-norm $\odot$ in use, that is

$$\oplus : [0,1]^2 \to [0,1], \quad (a,b) \mapsto 1 - ((1-a) \odot (1-b)).$$

Thus, in case that the default t-norm is used the maximum of the two truth values is taken. Moreover, the negation is, by default, the standard negation

$$[0,1] \to [0,1], \quad t \mapsto 1 - t.$$

Note that if one of the three standard t-norms, its corresponding t-conorm and the standard negation is chosen, then the sharp truth values 0 and 1 are connected as in classical two-valued logic. Thus, the compatibility with Arden Syntax is ensured in this case, the special treatment of `null` included.

In the present context two further connectives, denoted `at least` and `at most`, are important. In medical literature, when listing symptom combinations specifying a situation in which the presence of a certain disease is assumed we frequently encounter phrases like "at least two of the following conditions must be met: ...". By default these connectives are defined by the basic ones. For instance, let `List` be a list of truth values; then the expression

`at least` n `of List`

is the disjunction of all conjunctions of exactly n entries in `List`. We easily check that if the default connectives are used, the displayed expression returns the n-th largest value in `List`. The connective `at most` is specified similarly.

However, there are further reasonable possibilities to interpret `at least`. In Fuzzy Arden Syntax, a user-defined definition can be given, which need not be related to the chosen t-norm. For instance, the following interpretation has been proposed to interpret `at least` n `of List`:

$$\frac{\min\{v_1 + \cdots + v_k,\, n\}}{n},$$

where $v_1, \ldots, v_k$ are the truth values contained in `List`.

A few other logical operations are provided. For instance, `any of List` is the disjunction of the truth values contained in the list `List`. These operations depend on the three main connectives mentioned, and we will not enumerate them.

We next turn to the operations with numbers. Number variables may be connected by the basic arithmetic operations $+, -, \star, /$. A few other common functions are available. For time and duration variables these operations are also defined whenever it makes sense.

For variables storing fuzzy sets, addition and subtraction as well as multiplication with, and division by, positive crisp reals are defined according to Zadeh's extension principle, provided that the respective operation is well-defined. Zadeh's extension principle [4] is the canonical way to extend operations on real numbers to fuzzy sets. For instance, the sum $u + v$ of fuzzy sets u and v over $\mathbb{R}$ is defined by

$$(u+v)(x) = \sup\{u(y) \wedge v(z) : \; y + z = x\}$$

for $x \in \mathbb{R}$.

Finally, numbers as well as times and durations may be compared with respect to their natural order. The comparison of two crisp numbers yields a crisp truth value.

Furthermore, fuzzy sets being available, we need a way to query the compatibility of crisp values with properties modelled by fuzzy sets. A crisp number r, contained, say, in `Var`, may be correlated to a fuzzy number u contained, say, in `FuzzyVar`, by the expression

```
Var is FuzzyVar
```

which simply gives the value of u at r, that is, $u(r)$. Moreover, the expression `Var <= FuzzyVar` returns $\sup\{u(x) : r \leq x\}$, and similarly, `Var >= FuzzyVar` returns $\sup\{u(x) : r \geq x\}$. Analogous definitions apply for fuzzy times and fuzzy durations.

Finally, if `FuzzyVar` stores a fuzzy set, we may call a defuzzification function by

```
defuzzified Fuzzyset
```

returning an element of the respective domain. Centre-of-gravity defuzzification and mean-of-maximum defuzzification are predefined (for these methods, see, e.g., [5]). A user-defined function may be chosen in the fuzzy options category as well.

One operation is especially important in Arden Syntax and has been modified in Fuzzy Arden Syntax. Given a list, we may form a sublist by means of the operator `where`. We may actually think of the `where` operator as an operator forming a subset. In Fuzzy Arden Syntax this operator should serve to form a fuzzy subset. Let a list

```
List := Value1, ..., Valuen;
```

be given, and let `Condition`$(\cdot)$ be an expression of type truth value with one free variable. Then

```
List where Condition(it)
```

arises from `List` as follows: For each i, the degree of applicability of `Value`i is connected with `Condition(Value`i`)` by the `and`-operator. The result, if defined and >0, is stored as the new applicability of `Value`i; otherwise, the entry `Value`i is removed from the list.

To see how this command works, let us reconsider the above example. We may produce a list with a patient's body temperature from the last approximately 24 h using the following:

```
TempatureList := read{ temperature} where
it occurred within the past 24 hours
fuzzified by 4 hours;
```

This leads, just as proposed above, to a list of values from the last 28 h where the applicability of the values from the time period between 28 h ago and 24 h ago is reduced: The older the value is, the smaller is its applicability.

7. Conditional statements and branching in Fuzzy Arden Syntax

Conditions in Fuzzy Arden Syntax may be indeterminate. The truth values vary continuously from 0 to 1. Therefore, a command directing the programme flow into one of two or more blocks must be carefully interpreted, depending on the content of the respective propositional variable.

In full accordance with what we are familiar with, an if-then-else statement in Fuzzy Arden Syntax would be as follows:

```
if Condition
  then /block 1/
  else /block 2/
endif;
```

where `Condition` is an expression of type truth value. However, its manner of execution is one of the main differences between Arden and Fuzzy Arden Syntax.

The command is executed as follows. If `Condition` is 1, *block 1* is executed; if `Condition` is 0 or `null`, *block 2* is executed. If, however, `Condition` is $t \in (0, 1)$, the programme splits: *block 1* and *block 2*, named programme branches in the sequel, will be executed in parallel. To this end, each branch is provided with its own set of variables which, accordingly, are duplicated. Moreover, the degree of applicability of each variable is in case of *block 1* multiplied by t, in case of *block 2* multiplied by $1 - t$. t and $1 - t$ are called the relative weights of *block 1* and *block 2*, respectively.

The programme may branch several times. Each command executed during the run of the programme is assigned a weight in the straightforward manner. The weight is 1 as long as the programme does not split; when the weight is w and the programme enters a branch with relative weight t, the weight will be reduced to $w \cdot t$.

In a branch of weight w, the range of the degree of applicability of any variable is $[0, w]$. Whenever the content of a variable is changed its applicability will be reduced to w if necessary.

The number of branches into which the programme may split at a time is not limited to two. Branching into $n + 1$ blocks is coded as follows:

```
if Condition1 then /block 1/
  elseif Condition2 then /block 2/
  .  .  .
  elseif Conditionn then /block n/
  else /block n + 1/
endif;
```

In this case the relative weight t_i of the i-th branch is given by `Condition`i, where $i = 1, \ldots, n$. The case that `Condition`i is undefined is treated like $t_i = 0$, in which case the branch is not executed. Moreover, if the sum of the t_i is strictly smaller than 1, the relative weight of *block* $n + 1$ will be $1 - t_1 - \ldots - t_n$, else this block is skipped.

The possibility of letting the programme branch into more than two programme blocks is one of the significant features of Fuzzy Arden Syntax. Quite often we have to distinguish between conditions of the form `Var is FuzzySet1`, ..., `Var is FuzzySet`n, where `Var` is a crisp value and `FuzzySet1`, ... are fuzzy values. We allow an abbreviating syntax for this case, namely,

```
switch Var
  case Value1 /block 1/
  .  .  .
  case Valuen /block n/
  default /block n + 1/
endswitch;
```

The same is possible if the conditions are of the form `Var = Value1`, ... for any data type of the involved variables.

An example will follow in Section 10. In the following we will describe how one proceeds after completion of an if-then-else command.

8. Conditional statements and aggregation in Fuzzy Arden Syntax

Once all branches of a programme have completed their execution in parallel because of an unsharp condition, it is difficult to issue a general recommendation as to how one should proceed. Two possibilities exist:

(A) The programme remains split, that is, all subsequent commands are executed in parallel as well, the action slot included.
(B) The programme reunifies. The multiplied variables are merged into single ones.

Both options are available in Fuzzy Arden Syntax; possibility (A) is the default. The more appropriate option in the individual situation should be decided on the basis of the specific application.

If (A) is selected the MLM's results will be provided by each branch separately. The unit to which the results are sent – the host

system or the calling MLM – must be prepared to deal with the situation.

If the MLM sends information to the host, the host system has to process possibly divergent information. The key tool to be used is the third component of the data: the degree of applicability must then be interpreted by the host in order to conclude the relevance of the received data. For instance, when displaying the information to the user a clear statement as to the applicability must be added or the information must be modified in another appropriate way. The user in turn has to understand that the data, in case of low applicability, is to be interpreted as one of several possibilities.

If the MLM is called by another MLM and returns data the calling MLM splits accordingly as well.

The possibility (B) implies that the task of combining divergent pieces of information is executed within the MLM itself. To opt for (B), the final line of an if-then-else statement is modified: after the key word `endif` or `endswitch`, respectively, the key word `aggregate` is added. Thus, when writing

```
if Condition then /block 1/
  else /block 2/
endif aggregate;
```

the two branches unify after their execution. The programme weight is then set to the sum of the weight of the branches, i.e., to the same value as before.

Moreover, corresponding variables are aggregated. Let `Var` be a variable defined in at least one branch. As far as the main component is concerned, the procedure is as follows. If the content of `Var` is the same in each branch, the content is taken over. Otherwise, if `Var` is defined in all branches and of the same simple data type except `string`, the contents are aggregated according to a predefined method. If `Var` is of the same compound type in all branches, we proceed successively with the components in the same manner.

In the remaining cases `Var` is set to `null`. For example, this is applicable when a string variable is assigned a different text in two branches. No method is currently able to aggregate text automatically or make a canonical choice.

The aggregation method may be specified in the fuzzy options slot, separately for crisp and fuzzy data. Numerous methods are available to aggregate data and the implementation of a user-defined function is possible. By default the weighted mean is calculated. Thus, e.g., the values $r_1, \ldots, r_n$ with degrees of applicability $t_1, \ldots, t_n$, respectively, are aggregated to

$$\frac{t_1 r_1 + \cdots + t_n r_n}{t_1 + \cdots + t_n};$$

the same formula is used for crisp and fuzzy data. In the fuzzy case another predefined method is available, namely, the supremum of the r_i, cut off at height t_i, may be taken:

$$(r_1 \wedge \bar{t}_1) \vee \ldots \vee (r_n \wedge \bar{t}_n); \tag{1}$$

here, $\bar{t}_i$ is the constant t_i function, $\wedge$ connects two fuzzy sets by the pointwise minimum, and $\vee$ connects two fuzzy sets by the pointwise maximum.

To aggregate the contents of variables with respect to the remaining two components is straightforward. The primary time of `Var` is taken over if coincident in all branches. If distinct times appear we can no longer assume that these times are related to the time at which the value emerged; the primary time will be set to `null` in this case.

Furthermore, as might be expected, the degrees of applicability are added. Thus, if left unchanged during the execution of all branches, the applicabilities prior to the execution of the if-then-else statement, will be restored.

The `conclude` command is treated exactly in the same manner as the if-then statement. Thus, the command

```
conclude Condition;
```

is interpreted as "if `Condition` applies, then jump to the action part, else quit the programme". In other words, if `Condition` yields a value > 0, the applicabilities of all variables are multiplied by this value, and the action slot is executed. If `Condition` is 0 or undefined, the programme or this branch of the programme is terminated.

9. Fuzzy Arden Syntax versus Arden Syntax

This section will deal with the example from Section 4: the MLM `UTI_SUTI`. Let us see how a "fuzzified" version of this MLM looks in Arden Syntax on the one hand and in Fuzzy Arden Syntax on the other hand.

In Arden Syntax the result would look as follows, provided the logical connectives are chosen as the Fuzzy Arden Syntax default connectives.

```
maintenance:
mlmname:  UTI_SUTI_FuzzyEmulation;;
arden:  version 2.5;;
  [...]
knowledge:
[...]
data:
  (Stay, Date) := argument;
  Temperature := read {temp (Stay, Date)}
  if Temperature <= 38
      then Fever := 0;
    elseif Temperature <= 39
      then Fever := 39 - Temperature;
    else Fever := 1;


  endif;
  Urgency := read {urg (Stay, Date)};
  Micturition :=
    read {mict (Stay, Date)};
  Dysuria := read {dysuria (Stay, Date)};
  Suprapubic_tenderness :=
    read {suprtend (Stay, Date)};
  Organ_urine_culture :=
    read {org_urine_cult (Stay, Date)};
  if Organ_urine_culture >= 1e5
      then Urine_culture := 1;
    elseif Organ_urine_culture >= 5e4
      then Urine_culture :=
      (Organ_urine_culture - 5e4) / 5e4;
    else Urine_culture := 0;
  endif;
  ;;
evoke:
  ;;
logic:
  UTI_SUTI_FuzzyEmulation :=
    minimum (maximum(Fever, Urgency,
    Micturition, Dysuria,
    Suprapubic_tenderness),
    Urine_culture);
  conclude true;
  ;;
  action:
  return UTI_SUTI_FuzzyEmulation;
  ;;
end:
```

It is not necessary to print the MLM `UTI_SUTI` from Section 4 again in order to show what the fuzzy version looks alike under

Fuzzy Arden Syntax. The following changes must be made: (i) replace `Temperature > 39` by `Temperature >= 39 fuzzified by 1`; (ii) replace `Organ_urine_culture >= 1e5` by `Organ_urine_culture >= 1e5 fuzzified by 5e4`.

Fuzzy logical calculations have to be stated explicitly in Arden Syntax. This makes the programme more cumbersome. The only advantage might be transparency. In Fuzzy Arden Syntax, practically nothing changes in the programme text. The user who is familiar with Arden Syntax, however, must have some knowledge of the different ways in which the programme is interpreted.

Note that in case of emulation under Arden Syntax as well as in case of Fuzzy Arden Syntax, further processing of the result is not included in the MLMs. However, as the input values are provided in a more differentiated way, the result has to be treated in a more differentiated way as well. Considerably more effort is needed to communicate a result to the user if vagueness or uncertainty are involved. In the example, there will be more than the possibilities of "a SUTI applies" and "there is no evidence of a SUTI". The result is a continuous truth degree, to which one message from a larger set of messages should be associated, the two mentioned ones representing the extreme cases.

10. Fuzzy inference with Fuzzy Arden Syntax

This section illustrates how fuzzy inference may be realised in Fuzzy Arden Syntax.

In applications, fuzzy sets usually appear when partitioning a domain of values into subdomains without sharp borders and typically a conditioning according to this partition follows. A fuzzy partition over a domain M is a partition of unity over M, that is, a finite set $u_1, \ldots, u_n$ of fuzzy sets such that $u_1(x) + \cdots + u_n(x) = 1$ for each $x \in M$. In Fuzzy Arden Syntax, a fuzzy partition is conveniently stored in an object variable; we recall that instead of the key word `object`, we may equivalently use the key word `linguistic variable`.

Let us consider the following declaration:

```
RangeofAge := linguistic variable
  [Young, Middle_Aged, Old];
Age := new RangeofAge;
```

Consequently, we have created three variables, namely `Age.Young`, `Age.Middle_Aged`, and `Age.Old`. Each of these variables is ready to be used like any variable of a simple data type. We proceed by assigning these three variable fuzzy sets which form a partition of unity over the positive durations:

```
Age.Young :=   fuzzy set (0 years, 1),
    (25 years, 1), (35 years, 0);
Age.Middle_Aged :=   fuzzy set
    (25 years, 0), (35 years, 1),
    (65 years, 1), (75 years, 0);
Age.Old :=   fuzzy set
    (65 years, 0), (75 years, 1);
```

We may now use the blurred age ranges for distinctions as follows:

```
switch AgeofPatient
  case Age.Young
    DoseforPatient := LowDose;
  case Age.Middle_Aged
    DoseforPatient := MiddleDose;
  default
    DoseforPatient := HighDose;
endswitch aggregate;
```

Then the variable `DoseforPatient` will contain a weighted mean of the three values `LowDose`, `MiddleDose`, and `HighDose`, where the weights reflect the, possibly partial, compatibility of `AgeofPatient` with the three age ranges.

The example may be modified to include the case that the doses are fuzzy as well. If `Dose.Low`, `Dose.Middle`, `Dose.High` is a further partition of unity we may write as follows:

```
switch AgeofPatient
  case Age.Young
    DoseforPatient := Dose.Low;
  case Age.Middle_Aged
    DoseforPatient := Dose.Middle;
  default
    DoseforPatient := Dose.High;
endswitch aggregate;
DoseTobetaken :=
    defuzzified DoseforPatient;
```

In this case `DoseforPatient` contains the result of the aggregation of three fuzzy sets rather than the aggregation of three crisp values. The outcoming fuzzy set is defuzzified to the crisp value `DoseTobetaken`.

Note that, when choosing the supremum as the aggregation method for fuzzy sets, our last sample programme imitates a Mamdani–Assilian controller. Namely, it first determines to which extend the input value, i.e., the age, is compatible with each of the three conditions; let t_1, t_2, t_3 be the resulting truth values. Then, the fuzzy sets modelling a low, middle, and high dose, say r_1, r_2, r_3, respectively, are aggregated to

$$(r_1 \wedge \bar{t}_1) \vee (r_2 \wedge \bar{t}_2) \vee (r_3 \wedge \bar{t}_3).$$

Finally, the last command defuzzifies this fuzzy set and provides a sharp output value. For an explanation of a Mamdani–Assilian controller, see, e.g., [5]. Note that the text of the programme is not more complicated than the text of if-then rules.

11. Fuzzy Arden Syntax: a view back and forward

Our work relies on ideas which were developed in the Ph.D. Thesis of S. Tiffe [2] in 2003. When comparing our proposal to Tiffe's, we see that most elements were modified. Some of the changes are due to the fact that [2] is based on the older Arden Syntax version 2.1 and several improvements in Arden Syntax have been effected since that time. To explain all differences would exceed the present framework; the interested reader is referred directly to [2]. However, we shall list some facts.

- Tiffe's *degree of presence* and *degree of applicability* were replaced by our degree of applicability and the program weight, respectively. The function of these values and their mutual relationship between these two concepts has been newly defined. Furthermore, a truth value describing the applicability is associated with any data type and can be manually modified.
- In [2], a canonical way how to proceed after programme branching is not proposed; instead, several different possibilities are presented. In contrast, we have decided to let always all branches of a split programme be executed in parallel and we require the user to specify how the results are recombined.

  The difference is most evident in case of different texts sent to the host by different branches. If, for instance, there are results "Give medicament M immediately" and "To give medicament M is not recommended" from two program branches, it seems inappropriate to select one of these texts on the basis of their applicabilities. The user should rather specify how to deal with contradictory recommendations.
- We do not fuzzify while- or for-loops. Actually, in [2] a warning against while-loops is already contained.

- To process continuous truth values, we have generalised the existing boolean data type rather than introducing a new data type.

  Furthermore, we allow to associate the value `null` to a variable storing a truth value. It is a difference not to specify a truth degree or to use the value 0.
- We have strictly simplified the usage of linguistic variables. In particular, the use of external MLMs to define fuzzy sets are not necessary. The new features of Arden Syntax version 2.5 compared to 2.1 made these improvements possible.

An implementation of Fuzzy Arden Syntax according to the specification [3] is in progress. Whereas for the user, the transition from Arden to Fuzzy Arden Syntax might not be serious, the effort needed for an implementation of the extended language is quite high. The implementation of a Fuzzy Arden Syntax compiler will presumably be finished in the last quarter of 2009. Its subsequent use within the new data management system of the Vienna General Hospital is scheduled. Afterwards, a systematic test of the fuzzy extension will be undertaken.

12. Conclusion

We have outlined the specification of an extension of the programming language Arden Syntax, which is designed for CDS applications in medicine. Our primary reason for elaborating Tiffe's Fuzzy Arden Syntax is to incorporate the possibility to deal with indeterminate data, i.e., when the processed information is vague or uncertain.

We focused on keeping Fuzzy Arden as simple as Arden Syntax. The syntax is kept as close to natural language as is the case with Arden Syntax and ensures easy comprehension of an MLM's contents. Indeed a programme text, when fuzzy rather than crisp data is processed, remains practically the same in typical applications. When a Mamdani–Assilian inference is realised the programme text is closely correlated to the text of the if-then rules on which the inference is based.

The advantage of Fuzzy Arden Syntax when compared to Arden Syntax is evident. The actual benefits of Fuzzy Arden Syntax will have to be proven in clinical practice.

References

[1] Arden Syntax for Medical Logic Systems, Version 2.5, Health Level Seven; 2005.
[2] Tiffe S. Fuzzy Arden Syntax: representation and interpretation of vague medical knowledge by Fuzzified Arden Syntax. Ph.D. thesis. Vienna: Technical University Vienna; 2003.
[3] Vetterlein T, Mandl H, Adlassnig KP. Vorschläge zur Spezifikation der Programmiersprache Fuzzy Arden Syntax (Proposal of a specification of the programming language Fuzzy Arden Syntax—in German), technical report. Vienna: Vienna University of Technology; 2008. Available at http://www.meduniwien.ac.at/user/thomas.vetterlein/articles/FuzzyArdenSpezif.pdf (last accessed: 20 October, 2009).
[4] Zadeh LA. Fuzzy sets. Inf Control 1965;8:338–53.
[5] Nguyen HT, Walker EA. A first course in fuzzy logic. Boca Raton: Chapman & Hall/CRC; 2006.
[6] Mamdani EH, Assilian S. An experiment in linguistic synthesis of fuzzy controllers. Int J Man-Mach Stud 1975;7:1–13.
[7] Schuh C, Hiesmayr M, Ehrengruber T, Katz E, Neugebauer T, Adlassnig KP, et al. Fuzzy knowledge-based weaning from artificial ventilation (FuzzyKBWean). In: Jamshidi M, Fahti M, Pierrot F, editors. Proceedings of the world automation congress 1996. 1996. p. 583–8.
[8] Adlassnig K-P, Kolarz G. CADIAG-2: computer-assisted medical diagnosis using fuzzy subsets. In: Gupta MM, Sanchez E, editors. Approximate reasoning in decision analysis. Amsterdam: North-Holland Publ. Comp.; 1982 p. 219–47.
[9] Clark DF, Kandel A. HALO—a fuzzy programming language. Fuzzy Sets Syst 1991;44:199–208.
[10] Baldwin JF, Martin TP, Pilsworth BW. Fril: fuzzy and evidential reasoning in artificial intelligence. New York: John Wiley & Sons; 1995.
[11] Munakata T. Notes on implementing fuzzy sets in Prolog. Fuzzy Sets Syst 1996;98:311–7.
[12] Morales-Bueno R, Conejo R, Pérez de la Cruz JL, Clares B. An elementary fuzzy programming language. Fuzzy Sets Syst 1993;57:55–73.
[13] Zhao X, Li F. Denotational semantics of dynamic fuzzy logic programming language. In: Zhang YQ, Lin YT, editors. Proceedings of the IEEE international conference 2006 on granular computing. 2007. p. 409–12.
[14] Garner JS, Jarvis WR, Emori TG, Horan TC, Hughes JM. CDC definitions for nosocomial infections. In: Olmsted RN, editor. APIC infection control and applied epidemiology: principles and practice. Mosby: St. Louis; 1996. p. A1–20.
[15] Adlassnig KP, Blacky A, Koller W. Artificial-intelligence-based hospital-acquired infection control. Stud Health Technol Inf 2009;149:103–10.
[16] Klement EP, Mesiar R, Pap E. Triangular norms. Dordrecht: Kluwer Acad. Publ.; 2000.