ELSEVIER

Artificial Intelligence 116 (2000) 67–85

Artificial Intelligence

www.elsevier.com/locate/artint

Think!: A unified numerical–symbolic knowledge representation scheme and reasoning system

Christian Vilhelm [a,*], Pierre Ravaux [a], Daniel Calvelo [b], Alexandre Jaborska [b], Marie-Christine Chambrin [b], Michel Boniface [a]

[a] *Lab. Biomathématiques, Faculté de Pharmacie de Lille II, 3, rue du Pr. Laguesse, BP83, 59006 Lille Cedex, France*

[b] *UPRES EA n 2689, ITM, Batiment Vancostenobel, CH&U Lille, 59037 Lille Cedex, France*

Received 27 November 1997; received in revised form 7 May 1999

Abstract

More and more applications of artificial intelligence technologies are made in biomedical software and equipment. These applications are multiple: intelligent alarms, intelligent monitoring, diagnosis support, ... Several different knowledge representation schemes are already in use: decision trees, first-order logic expert systems, calculations (mathematical modeling), trained neural network simulations, ... All these techniques have their own preferred field of application, and they do not overlap. Building a complete diagnosis support tool would require the use of several of these techniques. The problem is therefore communication between these very different systems and the complexity of the composite result. This paper describes the Think! formalism: a unified symbolic-connectionist representation scheme which tries to subsume some of the precited formalisms. Being able to integrate these knowledge representation schemes in a single model enables us to use existing knowledge bases and existing knowledge extraction techniques to make them communicate and work together.

Keywords: Knowledge representation; Symbolic reasoning; Numeric reasoning; Truth propagation

1. Introduction

One of the main problems in an intensive care unit (ICU) is the multiplicity of the sources of information in the room and the service itself (pieces of biomedical equipment, papers,

* Corresponding author. Email: cvilhelm@phare.univ-lille2.fr. This work was funded by the Centre Hospitalier Régional Universitaire de Lille through the Institut de Technologie Médicale.

0004-3702/00/$ – see front matter
PII: S0004-3702(99)00095-8

wall blackboard, the medical staff, the patient himself). All this information has to be read, integrated, interpreted and synthesized by the clinician before he can take therapeutic action. The amount of information available renders its interpretation increasingly difficult.

To assist the staff, ICU rooms have been equipped with computers running monitoring software. These monitoring systems can have several objectives, requiring different technologies:

- Basic display centralisation stations with alarm monitoring, currently available from equipment manufacturers (HP, SpaceLabs, ...): typically these do not use knowledge-based techniques to interpret data but rather advanced display techniques along with signal analysis.
- Building more advanced monitoring systems with data interpretation requires knowledge-based techniques to be used [6]. But, systems using a single knowledge representation scheme tend to be limited to a single monitoring task, such as forecasting, alarm analysis or therapeutic planning. Specialization stems from adapting the representation formalism used to the type of acquired data and the objectives of the task. Thus, the global monitoring systems proposed [4,9] generally make use of several knowledge representation techniques to complete these different analysis tasks requiring different abstraction levels. Knowledge representation formalisms are described in [12]. Problems when dealing with multiple knowledge sources mainly concern communication between different representation formalisms, the choice of each subsystem competence, and the scheduling of all these sources. These lead to very complex systems that need high performance hardware to run, and are difficult to maintain.

As part of the Aiddiag project [11], we are trying to build a central low-cost workstation, placed at the patient's bedside, acting as a unique information display and interpretation system, to help the clinician.

- The system will have to acquire and interpret different types of data, thus requiring the simultaneous use of several different knowledge representation techniques. The amount of work done in the ICU monitoring field should be taken into account, and existing knowledge bases should be used when available. But we should be able to add new parts to the knowledge base when developed, without having to stop the whole system for testing and modifications.
- All these different knowledge-based subsystems using different formalisms should be capable of interoperating easily. The resulting system must be kept as simple as possible so that it can be integrated into a low cost platform or embedded hardware, and be easily maintained.
- The knowledge base should be easily understandable, so that the clinician can judge the relevance of rules, and possibly develop his own rules and integrate them into the system to test their accuracy.

In this paper, we have tried to explore a way of dealing with the multiplicity of knowledge representation schemes, data types and learning strategies by proposing a unified representation model [5] subsuming several formalisms currently used in ICU monitoring systems. By integrating all these representation techniques into a unique formalism, we can make them communicate and cooperate effectively, without the need for run-time translations or scheduling.

Think! is based on a connectionist structure, but loosely connected to have explicit paths. We have introduced symbolic representation objects into this network, together with the concept of propagating truth values associated with these symbols. An early model was described in [8]. What we describe here is a more complex model, but with a wider range of representation and reasoning possibilities, together with an implementation of the model.

We shall first describe this formalism, beginning with a global overview of the base elements. We shall see how these are generalizations of existing concepts used in existing formalisms (neurons, links), with added capabilities. We shall then examine more advanced concepts and the dynamics of the system, how forward and backward chaining are implemented, as well as learning. We shall explain how existing formalisms are implemented using the Think! system and how they can be made to interoperate.

2. Structural and operational definition of concepts

The model is based on three structure elements: containers, processes and tubes. The structure elements define a network representing the knowledge base. Reasoning is achieved by propagating excitations through the network from one element to another and by making calculations on these excitations. Excitations, the basic data of the network, are a pair consisting of a numerical value and an associated truth value. Fig. 1 shows an example of a Think! network. Containers hold the information (excitations), processes make calculations, tubes transport the information. Groups can be defined as sub-networks, representing functional units of the global network (we can see four groups in Fig. 1 defining two production rules, a totally connected neural network and a decision tree). These sub-networks exchange information through tubes.

We shall now describe the elements constituting the network, the dynamics of the system, and some advanced capabilities of the model.

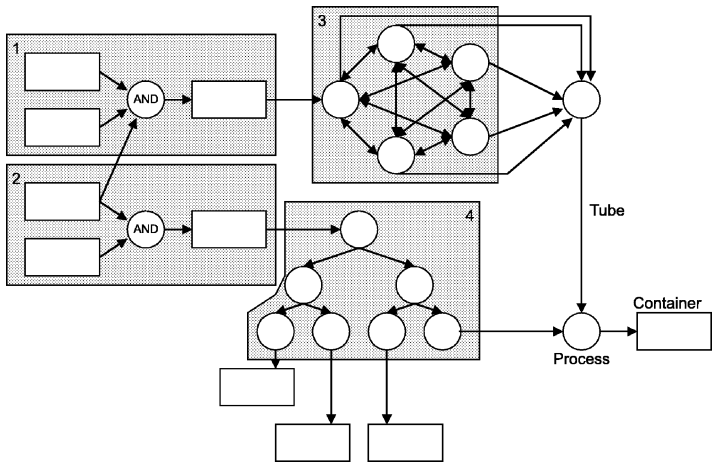

Fig. 1. A sample Think! network.

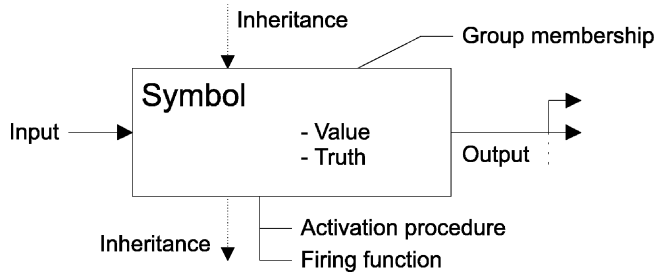

Fig. 2. Container definition and graphical representation.

2.1. Structure elements

2.1.1. Containers as holders and input/output slots of the network

Containers are symbol holders with a persistent internal state. They are the memory of the system, and the elements by which an external system can communicate with the network.

A container is defined by:

- A symbol, naming the container.
- A numerical value and a truth value, representing the state of the symbol (container), its *excitation*.
- One or more output tubes, transmitting the container's state (excitation) to other elements of the network.
- A maximum of one input tube, receiving new values that change the internal state of the container.
- An activation procedure, executed whenever the internal state of the container changes.
- A firing function, executed when a question arrives in the container. The firing function is responsible for communicating the internal state of the container to the element requiring it, and, if this state is unknown (truth value = Unknown), for establishing it.
- Inheritance relations, to organize containers with father–child 'is-a' relations.
- Group membership, to organize structure elements in functional groups.
- The capacity to pass or block. Blocking containers filter input and propagate it only if it is different from the current internal state of the container, this input replacing the internal stored value. Passing containers just propagate whatever input they receive.

Containers do not provide means of implementing learning. Input/output from an external program is carried out by modifying container excitation, by querying it, or by using the activation function, which can be an external program.

2.1.2. Processes as calculation elements

Processes are the active elements of the network, performing calculations and tests on the excitations.

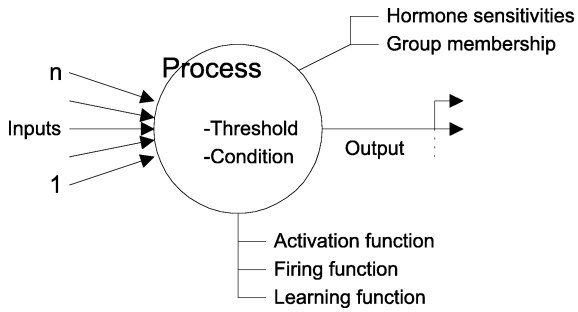

Fig. 3. Process definition and graphical representation.

A process is defined by:

- One or more input excitations θ_i $(i = 1, \ldots, n)$ coming from input tubes. Each input is *active* by default (triggering execution of the activation procedure upon arrival of an excitation) but can be specified as *inactive* (it will not trigger the activation procedure upon arrival of an excitation). Input tubes are labeled and input excitations are dated upon arrival, enabling calculations that rely on delays between incoming propagations, or on the age of an input value (e.g., to determine outdated excitations).
- One output excitation σ, transmitted through one or more output tubes; output tubes are not labeled.
- A threshold value.
- A condition test.
- An activation function f_{a}, computing σ from θ_i which decides if this output excitation should be propagated to the output tubes, using the condition and threshold as parameters.
- A firing function f_{f}, responsible for answering questions directed at the process about the status of its output excitation.
- A learning function f_l, responsible for making modifications to the network or to the process characteristics to adapt its calculated output to the preferred output received from one of the output tubes.
- Group membership and hormone sensitivities, to organize structure elements in functional groups and to apply global modifiers to elements in these groups.

In fact, processes are a generalization of formal neurons as found in neural network simulations. This enables us to implement neural network architectures and behaviour directly. The difference from classic neural networks is that the activation, firing and learning functions can vary from process to process, and can be specific (whatever their complexity) to processes or subnetworks.

2.1.3. Tubes as special links

Tubes are oriented links, propagating excitations from one element (container or process) to other elements of the network (containers, process or tubes), and attenuating or amplifying the excitations propagated. A tube has a *length*, and propagations travel into the tube at a given *speed*, after respecting a given *delay* before beginning.

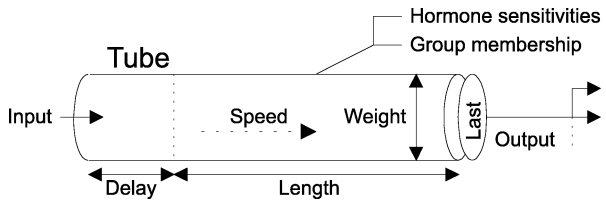

Fig. 4. Tube definition and graphical representation.

Tubes are a generalization of synapses used in neural network simulators. The introduction of delay, length and speed (thus a certain 'travel schedule' of excitations going through the tube) enables us to introduce temporal notions and dependencies into the knowledge base. Length alone makes it possible to model networks that have a certain physical coherence, with notions of proximity and distance.

2.2. Dynamic elements

2.2.1. Excitations

An *excitation* is the association of a numerical value with its truth value.

Truth value is a continuous value between True and False, with two additional special values: Unknown and Mu. Unknown means that there is no information about the truth value, which is different from the median value between True and False, meaning neither true nor false. The Mu value means that the information carried has no sense, so there can be no truth value. These special truth values must be taken into account in the implementation of logical operators used in the process functions.

Due to implementation constraints, all numerical values used in Think! must respect the same format. If we choose to implement the value contained in the excitations as 4-point fuzzy intervals, all other numbers will have to be 4-point fuzzy numbers: truth values, weights, thresholds, group memberships, hormone concentrations.

When an excitation is in transit in a tube, it is called a *propagation*.

2.2.2. Ticks and intrinsic time base

The network is updated in three steps:

(1) all propagations are moved further down the tube they are in (according to their speed and the length of the tube),
(2) all required functions (and only those required) are carried out,
(3) new excitations are given to tubes.

These three basic steps constitute a *tick*. Ticks do not have a fixed duration, because the system waits for these three steps to finish before proceeding to the next tick.

We evaluated the possibility of fixed-duration ticks, but this would require taking deferred propagations into account, in case the program, or computer, does not have enough time to perform all the necessary propagations during the time allocated to the tick. We determined that the complexity of managing desynchronized propagations (deferred one or more clock steps) is much greater than the added complexity of taking an external source for a real-time clock. This real-time clock can be transferred to where it is needed—from a

passing container propagating the value of the clock to the inactivated inputs of processes that need it.

2.3. Advanced concepts

2.3.1. Groups determine subnetworks

Structure elements (containers, processes, tubes) can be grouped together. Each of the elements in the group has a degree of membership, thus defining a fuzzy group, or set. This leads to several functionalities:

- By defining functional groups, one can isolate different separated sub-networks each performing a specific task in the global system. This way, complex networks become more understandable. It is possible to copy whole functional groups to make them repeat the same functionality for other data, or to make minor modifications to the structure or characteristics of elements and compare the behaviour of the two groups. Network design becomes modular.
- With the combination of hormones, one can make global modifications in the behaviour of elements included in the group.

2.3.2. Hormones introduce global parametrizations

Hormones behave like global variables for all elements contained in a specific group where the hormone is present. It is possible for some objects (processes and tubes) to define hormone sensitivities. When a certain hormone concentration is specified for a certain group, the hormone concentration applied to an object is the concentration of the hormone in the group multiplied by the object's degree of membership in the group.

The hormone concentration can serve as a parameter in any of the functions of the process—activation, firing or learning. They can serve as modifiers to certain characteristics of the processes (threshold), or tubes (propagated truth value, propagated numerical value, weight, speed, delay).

The modification induced is always calculated from the base value of the modified parameter. When an object has several hormone sensitivities for the same parameter, all the modifications are calculated separately from the base value and added.

$$\begin{aligned} \textit{modification} = \textit{base value} &* \textit{group membership} \\ &* \textit{hormone concentration in the group} \\ &* \textit{hormone sensitivity} \end{aligned}$$

$$\textit{final value} = \textit{base value} + \textit{modification}$$

2.3.3. Inheritance leads to immediate 'is-a' relationships

Containers may be organized in father–child relationships, which can be considered as 'is-a' relations. A child container will then inherit all inheritable output tubes from its fathers (inherited tubes are inheritable, allowing for multi-level inheritance). These inherited tubes will propagate the new excitations of the container as well as the output tubes directly connected to the container.

When several containers try to propagate an excitation through the same inherited tube, a conflict resolution must occur. The algorithm used is implementation-dependent, and currently uses a 'last-win' strategy.

3. Dynamics of the system

3.1. Forward chaining

The main operating mode of a Think! network is a feed-forward system, with added time-related possibilities given by delays and speeds, and with advantages from the differentiation of all processes thanks to a potentially wide variety of functions, from complex calculus to logical functions (operators).

Inputs are introduced into the system by exciting the input containers, i.e., changing their excitation and making them propagate and initiate execution of the activation function of processes.

When a *blocking* container gets an excitation, whether from its input tube or from an external program, it tests if this new excitation is different from the currently stored one. If it is different, the new excitation replaces the old one, the activation procedure is carried out and the new excitation is given to the output tubes. When a passing container receives an excitation, it performs the activation procedure and propagates the excitation without testing. The activation procedure can be used to display the new value, or to activate some external module (program), making it possible for a Think! network to control its environment.

Excitation arrives in a tube as input and then becomes propagation.

The tube has a certain delay, defined in ticks. Propagation waits for 'delay' ticks before beginning to travel. After this delay, propagation advances 'speed' distance units at each tick.

When the propagation arrives at the end of the tube (after length/speed ticks), either the numerical value or the truth value is multiplied by the weight of the tube, depending on the action defined for the tube. Weight, transmitted numerical value or transmitted truth value can be modified by various hormone actions when the tube is included in one or more groups. The resulting excitation is then transmitted to the output objects of the tube.

When the output object of a tube is another tube (for example, tube1 going from container A to tube2, tube2 going from process B to process C), any element of the output excitation of the tube (numerical value, truth value of tube1) can modify or replace any element of the output tube (numerical value, truth value, weight, speed, delay of tube2), as specified when creating the source tube (tube1). This is in fact similar to hormone actions, but on a more local scale, hormones being global modifiers.

When an excitation arrives on an active input of a process, the activation function of the process is performed. The process activation function, f_a, computes

$$(\sigma, \tau) = f_a(\theta_1, \ldots, \theta_n, \mathit{condition}, \mathit{threshold})$$

where θ_i = (value, truth, tick of arrival) from tube i and n = number of input tubes. σ is the output excitation of the process, τ is the decision (boolean) whether to propagate σ or not.

f_a can be any function. This activation function is executed no more than once per tick, if at least one new excitation has come from an active input. Inputs being labeled, f_a can compute nonsymmetrical functions, such as subtractions, divisions or comparisons.

When an excitation arrives on an inactive input of a process, the activation function of the process is not performed. But, when it occurs at a later tick, the input value from the inactive tube will be the most recent excitation received from this tube. In fact, excitations arriving at processes are retained at the interface between the tube and the process, so that functions performed have a value for each input and this value is the latest excitation to have arrived at the input.

Activation functions of processes are performed only if needed (if propagation comes from an active input). This is interesting from a performance point of view. Furthermore, it allows for the detection of active reasoning paths, where propagations often travel, and less active zones of the network, only activated when needed. This is very useful to debug the system, by detecting falsely active processes or tubes, according to the current inputs of the system.

3.2. Backward chaining

Another mode of reasoning used in knowledge based systems is backward chaining, either for deduction systems (goal driven), or to obtain learning capabilities (neural network simulations). Think! integrates both operating modes through the firing and learning functions of its elements and by transmitting information in the reverse direction of tubes, by propagating what we call reverse propagations. These reverse propagations are of two types:

- *Questions*: meaning that the destination of the tube transmitting the reverse propagation requires the source's value and truth to establish its own value and truth.
- *Learning*: learning reverse propagation contains a numerical value and truth value, representing the preferred output of the element. This means that the destination of the tube (origin of the reverse propagation) specifies to the process that its output is incorrect, and gives it the right values.

Reverse propagations when arriving at processes are handled by specific functions.

- The firing function f_f, is used when a question arrives from one of the output tubes. If the activation function decides to propagate σ, it is propagated to the output tube from which the question came. If f_a decides not to propagate σ, the question is considered pending and questions are reverse propagated to one or more input tubes of the process. The question is pending until an excitation is propagated to the output tube. This extends backward chaining to production systems, with the ability to modify the solution search strategy from process to process, adapting it to the semantics of the inputs.
- The learning function f_l, is used when a learning reverse propagation arrives from one of the output tubes. The learning function should reduce the error between the output of the activation function σ and the excitation transmitted by the learning reverse propagation θ'. The long term goal of the learning function is to have

$$\sigma = \theta' = f_a(\theta_1, \ldots, \theta_n, \mathit{condition}, \mathit{threshold}).$$

To achieve this goal, it has several possibilities:

- By modifying the weights of the input tubes, which is similar to the gradient backpropagation algorithm applied in neural network simulations [12], with the added possibility of modifying weight differently on each input tube, or ignoring some input tubes, depending on their meaning for the activation function. The rate parameter used in neural network simulations can be implemented as a hormone concentration or by a specific input of the process. This offers the advantage of changing the rate parameter during learning, and specifying different rates for different processes.
- By sending learning reverse propagations to one or several input tubes. This will be used when correcting only a small part of the error, letting elements situated further away the inputs correct the rest.
- By adjusting the hormone sensitivities of the process, or the sensitivities of input tubes. Or by modifying hormone concentrations.
- By changing other characteristics of the elements, such as the delays or speeds of tubes, or by altering the structure of the network.

All these possibilities do not have to be used in a single function: standard learning functions will use only one or two simultaneously to avoid learning functions that are too complex and CPU-hungry. It is interesting however that subsystems can be made to use different learning techniques.

4. Examples and translating existing formalisms

We are now going to show how existing knowledge representation schemes can be 'translated' into our formalism and retain all their capabilities.

4.1. Feed-forward

A forward-chaining production rules system can be implemented easily. Fig. 5 shows a very simple production rule.

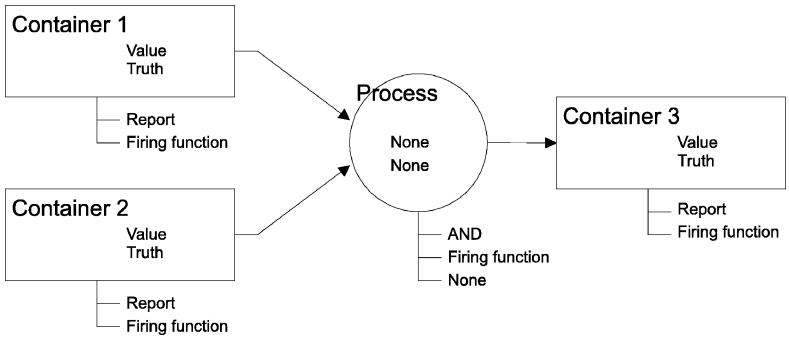

Fig. 5. Simple production rule: IF Container1 AND Container2 THEN Container3.

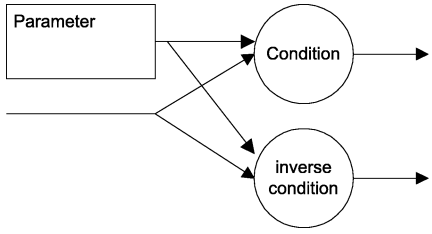

Fig. 6. Decision tree generic node.

There is no pattern-matching since applicable rules (tubes) are directly connected to the facts (containers). Of course, more complex rules are possible. Because the inputs of a process are labeled, it is possible to have very complex rules with any number of premises and complex relationships between them (multiple and, or, . . .), but this makes the network much less readable, and thus less explicit.

Rules can be inherited from upper containers, making it possible to establish general rules.

Another example of a feed-forward production system is the decision tree generated by induction algorithms, like C4.5 [7]. The translation between C4.5 output files and the Think! formalism has been implemented in an automatic translator directly reading the output files from C4.5 software. The tree node implementation is shown in Fig. 6.

Results obtained by the Think! network are exactly the same as the results obtained by C4.5 software, with the same test set, showing the adequation of the formalism to represent production rules.

Implementation of trained neural network simulations is fairly straightforward. Neural nets are usually organized in layers and we have three types of neurons: input neurons, output neurons and neurons hidden in several layers.

Input neurons just transmit their values to the first layer. These neurons are connected to several or all neurons in the next layer. Fig. 7 shows the implementation of the three neuron types: input neurons just transmit their values without modification (containers), layer and output neurons modify their outputs (processes). The activation function computes the sum of the inputs and propagates if the numerical value is positive. An output neuron gives its value to a container. The container is visible from the outside of the system, displaying the result, or activates some external program.

This was tested by setting-up an automatic translator program that takes a save file from an existing network simulation application [10] and creates the corresponding network in

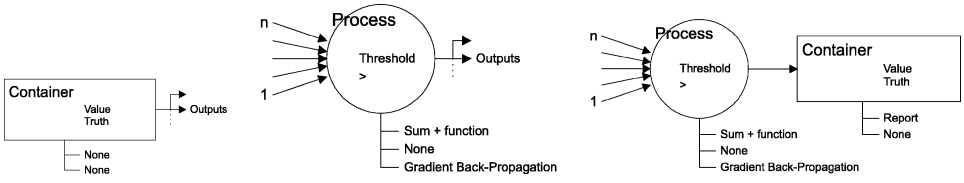

Fig. 7. Input, layer, output neurons expressed in the Think! formalism.

our formalism. The functions used for activation and learning functions are the same as in the original neural network. Of course, neural networks with other architectures can be implemented as well, the only constraint being the containers needed for input and output.

We have also implemented rules from a fuzzy expert system described in [13]. The values contained in the excitations were 4-point fuzzy numbers. The rule implementation in Think! was again straightforward and the results obtained are the same as those described by those authors.

4.2. Backward chaining

Backward chaining is obtained using the firing function, when an output container is asked a question. The firing function of the process is responsible for choosing the right solution-search strategy (depth first, breadth first, or any other type of search); it is possible to have a mixed-search method by having different firing functions in the same network for different processes. The firing function of the container determines if the question should be sent upwards through tubes or should be sent to an external program (to ask the user through a graphical interface for example).

For the Fig. 5 example: if Container3 is asked a question, the firing function sends a questioning reverse propagation to the Process. The firing function of the process then decides to send questions simultaneously to Container1 and Container2, thus making a breadth-first search. As Container1 and Container2 have no inputs, each of them will ask the user (or an external program) a question.

When a container's excitation is computed by a combination of subsystems (as in Fig. 1), a question addressed to one of the output containers can pass through many subsystems before arriving at the input containers, where the question is asked outside the network.

4.3. Learning

The implementation of untrained neural networks is the same as the implementation of trained neural networks, but all tube weights are set at 1.0 at creation time and the learning function is an implementation of the gradient back-propagation algorithm (that modifies tube weights). The rate parameter can be specified with a hormone concentration in the group containing the whole neural network. The activation function can be the standard sigmoid function or any other. Training is carried out by exciting the input containers with the input pattern of the training set and, when the network has computed its output, by reverse propagating the response pattern from output containers.

Another learning possibility is to adapt the weights of tubes used in production rules (representing confidence or certainty). By hiding the conclusions of the network and introducing (reverse-propagating) the conclusions found by the clinicians (entered through an advanced graphical interface) [2], the learning functions of the processes used in the rules will adapt the weights to progressively better the answers of the network (reinforcing the rules leading to correct conclusions by increasing the weights of the tubes used to find these conclusions, or weakening the rules leading to false conclusions). In this way, we could implement rules given by the clinicians and let the system refine them. After a certain learning time, it would be possible to remove inadequate rules.

4.4. Composite system

To show the integration capabilities of Think! and its ability to implement all the steps of a system, part of the reimplementation of our respiratory monitoring system RespAid [1] is shown, as it is now integrated into the Aiddiag system. The first stage of RespAid includes the filtering and symbolising of a numerical parameter acquired from a monitoring device (through a serial interface). The first part of the network (Fig. 8) takes the raw numbers given by the monitoring device and filters them by computing a weighted average of 3 successive measure points. The weights are directly implemented as tube weights and

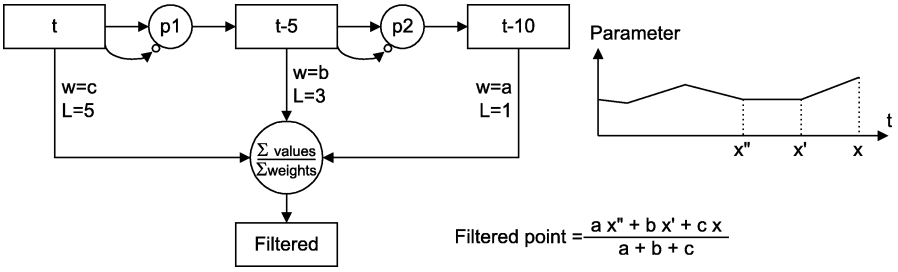

Fig. 8. Signal filtering.

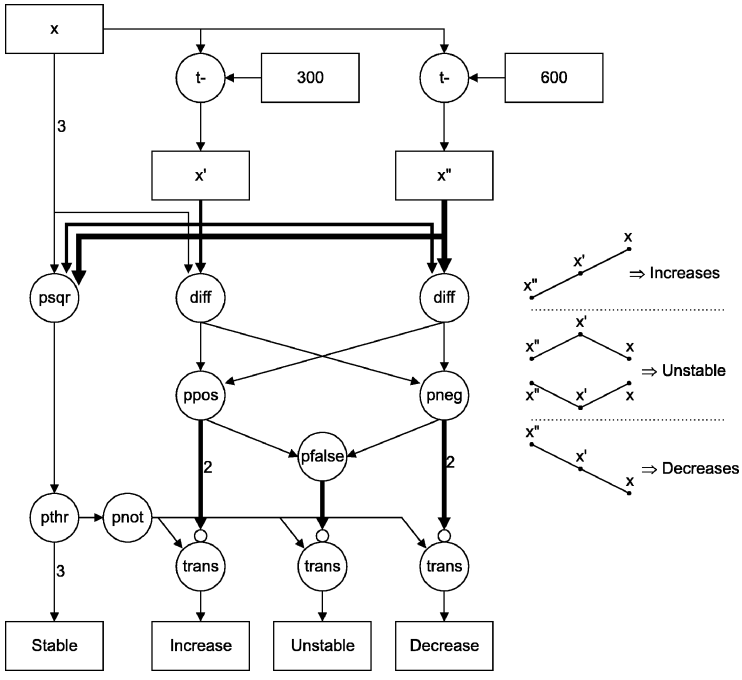

Fig. 9. Parameter symbolisation.

the length of the tubes is chosen so that all weighted measures arrive at the same tick in the averaging process. These filtered points are then sent back to the Aiddiag system and displayed.

A second stage (Fig. 9) evaluates the trend of the parameter's temporal evolution as a symbol ("stable", "unstable", "increasing" or "decreasing").

First, the network takes three values of the filtered parameter, at time t (now), $t-5$ minutes (300 seconds) and $t-10$ minutes (600 seconds): $x = p(t\text{m})$, $x' = p(t-5\text{m})$, $x'' = p(t-10\text{m})$. The "t-" processes query the Aiddiag system to collect the values of the parameter. The parameter is considered stable if the standard deviation of the three points is below a certain value, which is computed by processes "psqr" (standard deviation) and "pthr" (threshold). The networks computes $x-x'$ and $x'-x''$ (processes diff) and then tests if these differences are all negative (process pneg) or all positive (ppos). If ppos outputs the truth value 'true', then the parameter is considered to be increasing. If pneg outputs the truth value 'true', the parameter is considered to be decreasing. If ppos and pneg both output the truth value 'false' (tested by the process pfalse), the parameter is considered unstable. The lengths of the tubes are set so that all conclusions arrive simultaneously in the final containers. Weights of the tubes are all neutral (1.0).

These trends are then transmitted to a decision tree to detect potential medical complications. The trends are also used by the display mechanism.

We have shown here experimentally the ability of the model to integrate several subsystems using different formalisms in a single network, and yet retaining their special capabilities. By getting all these different subsystems to use a single formalism and data type, it is simple to connect them with tubes to let them interchange data, conclusions or facts. We have shown that different types of subsystems could be constructed, ranging from mathematical calculus networks to expert systems and learning neural network simulations.

We will now examine how this formalism has been implemented.

5. Implementation

The definition of the formalism was made parallel to its implementation, with stress on high performance and low memory usage, offering a program able to run on low cost hardware.

The current implementation is based on a client-server model. A *knowledge server* looks after all the reasoning and the management of the knowledge network. Clients then access the server to create, delete or modify structure elements, modify or read the excitation of containers concurrently. The server is constantly running, performing activation, learning and firing functions as needed. Functions are executed during a tick only when one of their inputs has been modified. When loops exist in the Think! network, the server is always putting forward some propagations and executing some functions, but due to the nature of the reasoning mechanism, these loops do not have adverse effects on the performance of the program, in fact, loops are necessary for closed systems where the final conclusion is re-injected at one of the inputs, or for two subsystems controlling each other.

The client is linked to a C library providing the API (Application Programming Interface) to access the server through a network socket. A special C variable type has been

created to be able to use special types for the numerical and truth values of an excitation, such as 4-point fuzzy intervals.

The client also has the possibility of providing the server with special process functions (activation, firing or learning). When the function needs to be executed, the server sends a message to the client along with all the function parameters. The client executes the function and sends back a message to the server containing the result. It is useful to have some very specific functions performed by clients to test them before their integration in the server. In a networked environment, it helps to distribute the calculation load.

The application is written in C language and is available for the Linux operating system (Free Unix compatible, see http://www.linux.org). This implementation has been placed under the GPL (GNU Public License).

Any program or language able to link to a standard Unix library is able to access a knowledge base system using our formalism. The current implementation has been extensively tested on PC compatible computers equipped with a simple Pentium processor running at 133 MHz, with 32 Mbyte memory. The program performs about 15,000 propagations per second and 15,000 activations par second with activation functions integrated into the server. Performance does not vary with the size of the network and scales well with processor performance. The steps of a tick are highly parallelizable and according to the locality of the processing, performance should scale well on multiprocessor computers, but this has not been tested. The program has a memory footprint of 1400 kbyte and storage of the knowledge base is also memory efficient (84 bytes/propagation, 468 bytes/container, 360 bytes/tube, 580 bytes/process).

6. Discussion and conclusion

We have shown experimentally that several knowledge representation schemes can be translated into our formalism and retain all their capabilities. We have also shown by our implementation of the Respaid system that several of these representation schemes could interoperate together and with numerical calculus parts in the same system.

- The graphical representation is easily understandable by non-AI specialists. They can also easily express their knowledge on paper using this graphical representation, but a C coder is necessary to implement the knowledge base into the program. In fact, one of the main drawbacks of our system is the lack of a graphical user interface. Conceiving and implementing such a graphical knowledge modeler is one of our urgent objectives.
- Adding delays, speeds and lengths to the tubes makes it possible to enforce timing dependencies (order) between the activation of conclusions, or temporal reasoning. Of course, Think! does not have hard real-time capabilities, due to the nonconstant duration of a tick. But, with the introduction of an external clock or time source, a Think! network can take into account clock (wall) time and can be made to output conclusions at fixed frequencies.
- Adding weights helps to strengthen or to weaken conclusions found.

The static nature of existing systems renders the evolution of the knowledge base difficult. It is not rare to be forced to stop a program to add, delete or modify the knowledge base used. The implementation of the Think system has been achieved in a totally dynamic

manner, any object or characteristic of object or even the structure of the network can be modified at run-time:

- It is possible to attach or detach parts of a Think! network dynamically. This is used in our Aiddiag system when we add or remove the knowledge base concerning a certain piece of biomedical equipment as it is required. For example, when Aiddiag is monitoring a patient and the clinical staff enters the room to connect a ventilator, Aiddiag dynamically adds the respiratory knowledge base to the pool of knowledge currently in use, and this without stopping or restarting the system. The new rules are immediately available and used. When the respirator is disconnected, the respiratory knowledge base is removed from the system without restarting the program.
- By changing functions dynamically we can test several solutions one after the other. Even if the tested function is not included in the server, we can implement it in the client and change it at run time without interrupting the server. It is possible to test several different structures by switching the connections between groups to make the system use one or the other.
- By temporarily relocating the carrying-out of functions from the server to the client one can trace what is happening in the network (content of propagations arriving, output of the function, ...), meaning that run-time debugging of the knowledge base and of the functions can take place.
- It is possible to remove unused knowledge or to dynamically remove rules leading to false conclusions, or just to redirect their output to another part of the network.
- The dynamic nature of the implementation does not hamper its performance. Altering the structure or modifying characteristics of elements is done without any performance loss.

Currently used systems or formalisms mainly use only a fixed, predetermined set of operators (disjunction and conjunction, sigmoid function and its inverse function...). Since we want to be able to run all these formalisms in one unique system, we need to integrate all possible operators and functions. In the definition of the Think! formalism and in its implementation we decided not to put any restriction on the functions or operators used throughout the network.

- Because there is no limitation on the activation functions, use can be made of only simple operators (AND, OR), or only simple calculations, like the sum-threshold function in neural networks. But it is also possible to have other operators (min, max, ...) or more complex calculations (the membership function in a fuzzy interval). It is also possible to have very complex calculations made in a single process, like a polynomial equation solver where the coefficients are given by the excitation transmitted by the input tubes.
- The same possibility exists for the learning functions, no limits are put on them. It is then possible to implement any learning algorithm by just changing the learning function, and to have different learning strategies in different sub-networks (groups). Due to the dynamic nature of the implementation, it is also possible to change the learning strategy during run-time to adapt it to the current situation. It is the responsibility of the learning function to choose between the several possibilities it has to change its inputs (tube weights, hormones, structure modification, ...). It should be noted that standard learning functions such as the neural network learning

function only uses a few. A Think! learning function is not forced to use all techniques available.
- Differentiating the firing functions makes a mixed-search backtracking strategy possible by changing the firing function from process to process. Firing questions into all input tubes at the same time is a breadth-first search, firing questions into the tubes one at a time is a depth-first search. It is possible to fire questions into several tubes at once but not into all tubes.
- Of course, this freedom can be a drawback. By differentiating the process functions too much or inadequately, the network can be rendered difficult to understand. It is the responsibility of the conceiver of knowledge bases to limit the use of complex functions, or to choose a fixed set of operators and functions. Also, this freedom for operators prevent us from proving the global completeness or consistency of a Think! network (it should be possible on subnetworks using only certain operators).

7. Perspectives

As we have shown in the examples, the Think! formalism enables us to use existing knowledge bases by translating them. It enables us to reuse existing knowledge extraction techniques, like induction algorithms, pasting their results into our global system after translation. The development of automatic translators enables people to use the Think! program to test their existing knowledge bases and exchange them with other people using the Think! program to compare their systems and results. When there is no translator, the C API provided by the implementation is very easy to use, and it is then possible to integrate Think! capabilities into existing programs.

Expressing all these distinct knowledge bases in a single formalism and implementing them in a single system means they are tightly integrated. Communication is carried out by simple connections with tubes; conclusions of one system can be directly used by another. This integration makes it possible to compare results from different subsystems using different formalisms. This is a first and important step in creating a centralized monitoring workstation.

The graphical nature of this formalism and the simplicity of the dynamics (feed-forward) means non-AI specialists can express knowledge directly into simple Think! networks and understand existing rules expressed in the formalism (implementation of these networks still requires a C programmer). By providing a catalogue of existing processes implementing certain functions (operators, tests, simple calculus, display, interaction), we hope to provide the final users with a toolbox that will enable them to conceive whole programs by just connecting containers (representing variables) to chains of processes expressing the required algorithm. This will require a graphical user interface and debugger.

The freedom given to the available functions has enabled us to implement signal processing sub-networks directly connected to the symbolic interpretation system, without having separate programs. The client–server architecture and the inherently parallelizable reasoning mechanism allows for the use of multiprocessor computers or networks of computers to handle very large knowledge bases or very complex and processor hungry

activation functions without hampering the performance of the global system. As the implementation is made in standard C language, it should be easily portable on other computer architectures.

The flexibility of the implementation means functions are tested before being integrated into the server; also, several different algorithms can be tested by modifying dynamically the functions or the structure of the network during run-time. This is important for debugging the network and for testing new solutions. We are now investigating how to change network structures dynamically to achieve advanced learning abilities and the blending of knowledge bases.

The implementation has been integrated into the Aiddiag medical data acquisition, report and interpretation system, in the RespAid module, to monitor the respiratory function of the patient. The system is constantly fed with data acquired from biomedical equipment. We are now implementing more complex knowledge bases which take into account additional acquired parameters to make better interpretations of signals. We are also investigating the possibility of incorporating Think! networks into the Aiddiag system to fully control the user interface, by deciding which items to display, and to determine the screen layout from the medical status of the individual patient and the individual user in front of the computer (physician, nurse, ...).

The Think! formalism and its implementation have been designed in respect of the requirements of the biomedical field. But we believe the resulting system is usable in a wide variety of fields, due to the great flexibility of the model and its implementation.

References

[1] M.C. Chambrin, P. Ravaux, C. Chopin, J. Mangalaboyi, P. Lestavel, F. Fourrier, RESPAID: Computer-aided evaluation of respiratory data in ventilated critically ill patients, Internat. J. Clinical Monitoring and Computing 6 (1989) 211–215.

[2] M.C. Chambrin, P. Ravaux, A. Jaborska, C. Beugnet, P. Lestavel, C. Chopin, M. Boniface, Introduction of knowledge bases in patient's data management system: Role of the user interface, Internat. J. Clinical Monitoring and Computing 12 (1995) 11–16.

[3] M. Dojat, F. Pachet, Z. Guessoum, D. Touchard, A. Harf, L. Brochard, NéoGanesh: A working system for the automated control of assisted ventilation in ICUs, Artificial Intelligence in Medicine 11 (2) (1997).

[4] B. Hayes-Roth, R. Washington, D. Ash, R. Hewett, A Collinot, A. Viña, A. Seiver, Guardian: A prototype intelligent agent for intensive-care monitoring, Artificial Intelligence in Medicine 4 (2) (1992) 165–185.

[5] M. Minsky, Logical vs. analogical or symbolic vs. connectionist or neat vs. scruffy, Artificial Intelligence at MIT, Expanding Frontiers, Vol. 1, MIT Press, Cambridge, MA, 1990.

[6] F.A. Mora, G. Passariello, G. Carrault, J.-P. Le Pichon, Intelligent patient monitoring and management systems: A review, IEEE Engineering in Medicine and Biology (1993) 23–32.

[7] J.R. Quinlan, Improved use of continuous attributes in C4.5, J. Artificial Intelligence Res. 4 (1996).

[8] P. Ravaux, C. Vilhelm, M. Boniface, M.-C. Chambrin, A neural approach to knowledge base systems, in: Proc. 14th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Paris, France, 1992, pp. 928–930.

[9] D.F. Sittig, N.L. Pace, R.M. Gardner, E. Beck, A.H. Morris, Implementation of a computerized patient advice system using the HELP clinical information system, Computers and Biomedical Research 22 (1989) 474–487.

[10] SNNS Stuttgart Neural Network Simulator: Users guide.

[11] C. Vilhelm, A. Jaborska, M.-C. Chambrin, P. Ravaux, A software architecture for a medical data acquisition, report and interpretation system in intensive care unit, in: Proc. 18th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Amsterdam, The Netherlands, 1996.

[12] P.H. Winston, Artificial Intelligence, Addison-Wesley, Reading, MA, 1992.
[13] Z. Zalila, P. Lezy, Contrôle longitudinal d'un véhicule autonome par régulateur hybride flou / classique, in: Proc. 5ème Conférence Internationale IPMU, Information Processing and Management of Uncertainty in Knowledge-Based Systems, Vol. 1, 1994, pp. 478–484.