AMIA: an environment for knowledge-based discrete-time systems simulation

Michel Page **Jérôme Gensel** **Mahfoud Boudis**

Université Pierre Mendès France, Grenoble
INRIA Rhône-Alpes
655, avenue de l'Europe
38330 Montbonnot Saint-Martin, France
e-mail: {Michel.Page,Jerome.Gensel,Mahfoud.Boudis}@upmf-grenoble.fr

Abstract

In this paper, we present AMIA, a workbench for developing knowledge-based discrete-time simulation systems. AMIA is original in two respects. First it uses an algebraic modeling language for combining discrete-time models (difference equations) and symbolic knowledge. Second, it uses a new simulation algorithm able to exploit this combination of numerical and symbolic knowledge. AMIA also includes a model management system for supporting the modeling and simulation process.

Introduction

Numerical simulation models embody quantitative knowledge about a specific system in the form of numerical variables and mathematical relations (equations). An important problem with numerical modeling is that much information and knowledge concerning the system described by the model and the context in which it operates are not numerical in nature. This includes: the entities that compose the system, the properties of these entities as well as the relations between them; the context of validity (hypotheses) of the equations of the model; the variants of the model and the domain knowledge required to select one of them, adequate in a specific context. Because of its symbolic nature, this knowledge is not taken into account in present simulation tools.

To overcome this problem, researchers in artificial intelligence (AI) and simulation have spent much effort during the last decade in studying knowledge-based simulation systems; see (Widman, Loparo and Nielsen 1989) and (Kowalik 1986) for an overview. In these systems, symbolic knowledge, stored in a knowledge base and exploited by a reasoning system, is used for various tasks of the modeling-simulation process: assistance in formulating models or in performing simulation, explanation of simulation results, ...

Most researchers working on knowledge-based simulation attempt either to couple a numerical simulation tool with an AI knowledge representation language (Holsapple and Whinston 1988), (Klein and Methlie 1995) or to integrate numerical simulation facilities in an AI knowledge representation language (Reboh and Risch 1986), (Gelman et al. 1988). However, the result is in both cases affected by a well-known bottleneck of knowledge-based systems: knowledge acquisition, because using a knowledge representation language usually requires to be an AI specialist.

In this paper, we propose a new framework for combining discrete-time models (expressed as sets of difference equations) and knowledge-based systems. This framework makes it possible to explicitly describe:

- the entities that compose a system, the properties of these entities as well as the relations between them. It thus reduces the conceptual gap existing between a model and the system it describes. It also increases the model intelligibility and facilitates the automatic explanation of simulation results.
- the context of validity (hypotheses) of the equations of a model. It reduces the danger of misusing a model and increases its reusability.
- different variants of a model and the domain knowledge required to select the adequate one in a specific context. It enhances model reusability.

We combine difference equations models and symbolic knowledge into an algebraic modeling language (AML). An AML is a computer-readable language similar to the algebraic notations used in mathematics. AMLs have been primarily designed for numerical modeling. However, their expressive power is very high and, as we will show, they can also be used for representing symbolic knowledge. Furthermore, because algebraic formalism is familiar to most modelers, they are able to build a model involving symbolic knowledge on their own, avoiding by this way the knowledge acquisition bottleneck.

In order to exploit the combination of numerical and symbolic knowledge, we have developed a new simulation/inference algorithm for AMLs. This algorithm combines graph-theoretic and numerical methods.

These ideas are implemented in a computer program for building and exploiting knowledge-based discrete-time simulation systems called AMIA.

The paper is organized as follows. We first explain what AMLs are and why they are appropriate for representing numerical as well as symbolic knowledge. The simulation/inference algorithm adapted to the AML of AMIA and the model management system of AMIA are then presented. Conclusions are finally drawn.

Representing numerical and symbolic knowledge in AMLs

AMLs are based on algebraic notations. Algebraic notations are widely used in scientific textbooks and publications for describing mathematical models consisting of equations and/or constraints. Allowing indexed expressions, sets, variables and iterated operators like Σ and Π, they provide a convenient way to form expressions such as:

$$\forall i \in I \quad a_i = \sum_{j \in \{j\}} x_{ij} \leftrightarrow y_j$$

AMLs have become very popular in the Operations Research community through languages like AMPL (Fourer, Gay and Kernighan 1990) and GAMS (Brooke, Kendrick and Meeraus 1988). More recently, AMLs have also been used in AI for constraint programming (Michel and van Hentenryck 1996).

The popularity of AMLs for numerical modeling comes from different factors. First, it is not necessary to be a computer scientist in order to use these languages: the effort to implement a model using an AML is small once the mathematical equations are available. Second, AMLs are declarative: each mathematical equation or constraint in a model forms an independent corpus of knowledge, and the order in which the equations and/or constraints are written is unimportant. These features make AMLs very suitable for building numerical models.

The claim we make with AMIA is that AMLs are also adequate for representing symbolic knowledge. This claim may seem surprising because AMLs have been designed for numerical modeling. However, we now show that one can also use them for expressing symbolic knowledge.

Knowledge encoded using an AI knowledge representation language is usually of two kinds: factual and deductive. Let us introduce how both kinds of knowledge are expressed in AMIA.

Representing factual knowledge in AMIA

In AMIA, factual knowledge is encoded at three levels: atoms, sets and variables. We introduce these concepts through the example below referred as the "market example" in the sequel of the paper.

Let us consider a market on which are shipped three products: P1, P2 and P3. This market is divided in two segments: A and B. Segment A corresponds to products with a high price, say 10% more than the average price of the products, and a high quality. Other products are in segment B. Assuming that:

- the annual demand for segment A is approximated to represent 20 % of the total annual demand in the forthcoming years;
- the annual demand for products in segment A is equally distributed over each product of this segment;
- the annual growth of the demand for each product in segment B is 10%.

We want to forecast the demand for each product and the total demand for the forthcoming years.

Data about the products: quality (assumed time-independent), price and demand for the initial year (1998) are presented in Table 1.

Product	Quality	demand 1998	price 1998
P1	Medium	100	300
P2	high	20	600
P3	high	25	600

Table 1: data for the market example

Atoms are the most basic elements in AMIA. One can think about them as distinguishable entities in the real world, denoted by a symbolic constant. Examples of atoms in the market model are product P1 and segment A. Modeling in AMIA consists in defining properties of atoms and the relations between them.

Sets are used to group atoms having common meaning, properties and relations. PRODUCTS = {P1,P2,P3} is an example of set in the market model. A set can be used in two different ways in AMIA: first, as a domain for the value of variables and expressions, i.e., a set in which a variable or an expression takes its value and, second, as a way for indexing variables and expressions (see below). Predefined sets are provided in AMIA for booleans, integers and reals. Time is also treated as a set whose atoms are the relevant time points for simulation.

A *variable* corresponds either to a property shared by the atoms of a set or to a relation between atoms from two or more (possibly equal) sets. From a mathematical point of view, a variable V is a total or partial function:

$$V: S_1 \times S_2 \times ... \times S_n \rightarrow S$$
$$(x_1, x_2, ..., x_n) \mapsto V(x_1, x_2, ..., x_n)$$

where S and each S_i ($i \in \{1,...,n\}$) are sets. For instance, the quality of the products is defined as a variable QUALITY indexed by the set PRODUCTS and taking its value in the set LEVELS = {LOW,MEDIUM,HIGH}. Relations between sets are also represented by variables. For instance, the price of the products at a particular time is a relation between PRODUCTS and T (here, T denotes time, i.e., the set of simulation time points) modeled by a real-valued variable: PRICE.

We note $V(S_1,S_2,...,S_n)$ the variable V indexed by the sets $S_1,S_2,...,S_n$. For instance, the two above mentioned variables are noted QUALITY(PRODUCTS) and PRICE(PRODUCTS,T). For a particular tuple of atoms $(A_1,A_2,...,A_n) \in S_1 \times S_2 \times ... \times S_n$, the variable V applied to $(A_1,A_2,...,A_n)$ is called a *scalar variable* and is noted $V(A_1,A_2,...,A_n)$. For instance, the scalar variable denoting the quality of product P1 is written QUALITY(P1).

Representing deductive knowledge in AMIA

In AMIA deductive knowledge is represented by equations. In a purely numerical simulation system, equations express numerical relations between numerical variables; they allow the computation of unknown variables from known ones. In AMIA, equations can also express symbolic relations between variables (see for instance equations (5) and (6) below). Unary variables

represent properties of entities, while n-ary (n>1) variables represent relations between entities.

In AMIA, equations are expressed in a particular form called *explicit form*. The left-hand side of an equation in explicit form only contains one variable; the right-hand side is an expression indicating how the left-hand side variable is computed. An AMIA equation defines the value of a variable $V(S_1,S_2,...,S_n)$ on a subset of $S_1 \times S_2 \times ... \times S_n$ with the following format:

x_1 in $\sigma_1(S_1)$, x_2 in $\sigma_2(S_2)$, ..., x_n in $\sigma_n(S_n)$:
$V(x_1,x_2,...,x_n) = expr$

where:

- $\sigma_i(S_i)$ ($i \in \{1,...,n\}$) is a subset of set S_i;
- x_i ($i \in \{1,...,n\}$) are called *indices*; they have the same meaning as in standard algebraic notations. They are dummy identifiers (written in lower-case) used as subscripts of variables and expressions, and denoting an atom in a set.
- *expr* is an AMIA expression formed with numerical constants and atoms, variables, sets, indices, operators and functions.

Each equation has two parts: the *extent* (x_1 in $\sigma_1(S_1)$, x_2 in $\sigma_2(S_2)$, ..., x_n in $\sigma_n(S_n)$) which delimits the domain of validity of the equation and the *defining expression* ($V(x_1,x_2,...,x_n) = expr$) which states how the variable is defined in this extent. A variable can be defined by several equations as long as the extent of these equations are disjoint.

In AMIA, models are made of linear and/or non linear simultaneous equations. These equations can be algebraic equations and difference equations (see for instance equation (3) below). Equations can be piece-wise defined, i.e., the defining expression of a variable can be dependent on one or several condition(s). For this reason, we call these equations *piece-wise defined equations* (PWDEs). The kind of symbolic knowledge discussed in the introduction: description of the entities that compose a system, their properties and relations, hypotheses of validity of equations and variants of a model can be conveniently expressed with PWDEs written in explicit form. This is due to the fact that much of this knowledge is in the form "this property or relation is defined this way in this context". This point is illustrated below, on the market example.

Sets:
T = {1998,1999,2000,2001}
PRODUCTS = {P1,P2,P3}
SEGMENTS = {A,B}
LEVELS = {LOW,MEDIUM,HIGH}

Variables:
TOTAL_DEMAND(T) → REAL
DEMAND(PRODUCTS,T) → REAL
SEGMENT(PRODUCTS,T) → SEGMENTS
PRICE(PRODUCTS,T) → REAL
AVERAGE_PRICE(T) → REAL
PRICE_LEVEL(PRODUCTS,T) → LEVELS
QUALITY(PRODUCTS) → LEVELS

Equations:
(1) t in T:
TOTAL_DEMAND(t)=
sum(p in PRODUCTS: DEMAND(p,t))
(2) p in PRODUCTS, t in T-{1998}:
DEMAND(p,t) =
if SEGMENT(p,t) = A
then 0.2 * TOTAL_DEMAND(t) /
card(p1 in PRODUCTS:
SEGMENT(p1,t) = A)
else 1.1 * DEMAND(p,t-1)
(3) p in PRODUCTS, t in T-{1998}:
PRICE(p,t) = 1.1 * PRICE(p,t-1)
(4) t in T:
AVERAGE_PRICE(t) =
average(p in PRODUCTS:PRICE(p,t))
(5) p in PRODUCTS, t in T:
PRICE_LEVEL(p,t) =
if 0.9*AVERAGE_PRICE(t) > PRICE(p,t)
then LOW
else
if 1.1*AVERAGE_PRICE(t) > PRICE(p,t)
then HIGH
else MEDIUM
(6) p in PRODUCTS, t in T:
SEGMENT(p,t) =
if QUALITY(p) = HIGH and
PRICE_LEVEL(p,t) = HIGH
then A
else B

AMIA simulation algorithm

Solving an AMIA model amounts to solving a system of simultaneous piece-wise defined difference and/or algebraic equations. In classical discrete-time simulation systems, equations contain no condition, variables are only indexed by time and every variable is numerical. In such systems, simulation is generally performed in four steps. First, an oriented graph is associated with the system of equations. This graph, called *dependency graph*, describes the variables dependencies. An ordered pair (v,w) of vertices (variables) in this graph expresses that the variable v at time t appears in the right-hand side of the equation defining w at time t. Second, the *strongly connected components* (SCCs) of this graph are computed. Each SCC represents a subsystem of simultaneous equations which can be solved independently from the others. Third, a topological sort is performed on the SCCs for determining the order in which the associated sub-systems are to be solved. Fourth, for each simulation time point, the sub-systems of equations are solved in the order determined in the previous step, using an adequate solving algorithm.

The previous algorithm would not work in AMIA, because of the expressive power of its AML. First, in AMIA, one writes PWDEs, i.e., equations which contain conditions. The dependency graph cannot be computed once for all, because the expression defining a variable (the right-hand side of the equation) is known only when the condition can be evaluated. Second, variables can be indexed by several indices. It means that PWDEs can be recurrent on any index, not only on those denoting time. Hence, SCCs must be computed, not from the variables themselves, but from each of their associated scalar variables. Third, variables which have a symbolical value cannot be handled by numerical equation solving algorithms.

For these reasons, we have devised a more powerful simulation algorithm for AMIA. This algorithm is made of two components (Figure 1): a *simulation engine* and an *equation solver*. The simulation engine dynamically builds and explores the dependency graph in order to find sub-systems of simultaneous equations. When a

sub-system is discovered, it is sent to the equation solver which attempts to numerically solve it and sends the results (be they successful or not) back to the simulation engine. This one then integrates the results obtained and proceeds with the exploration of the graph. We now detail these two components.

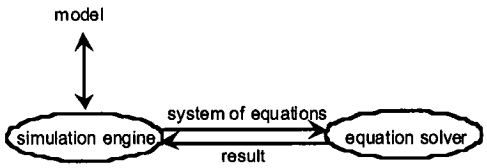

Figure 1: architecture of AMIA simulation algorithm

The simulation engine

The simulation engine, described in details in (Boudis 1997), is based on Tarjan's SCCs detection algorithm (Tarjan 1972).

To determine the SCCs of a directed graph, Tarjan's algorithm explores it in a depth-first manner. A depth-first search (DFS) from a vertex u in a graph induces a tree rooted at u called *DFS tree*. During DFS, when going from a vertex v to a vertex w, one of the statements below must hold:

- w is unexplored: (v,w) is a *tree edge*;
- w is already explored and w is an ancestor of v in the DFS tree: (v,w) is a *back edge*;
- w is already explored and w is a descendant of v in the DFS tree: (v,w) is a *forward edge*;
- w is already explored and neither v is a descendant of w in the DFS tree, nor w a descendant of v: (v,w) is a *cross edge*.

Tarjan has demonstrated that the vertices of an SCC form a subtree in the DFS tree. He named the root of this subtree the *root* of the SCC. His algorithm determines the SCCs by identifying their roots. This is done using an index, *lowlink*(v), corresponding to the vertex with the smallest number in the same SCC as v and the above edges classification which helps in maintaining this index. Edge classification is handled using two parameters:

- *number*(v), order in which vertex v is visited in DFS.
- *stack*, the stack of the vertices traversed in DFS.

When an edge (v,w) is traversed, it is first classified and the *lowlink* parameter of v is then maintained accordingly. A tree edge is characterized by the fact that w is not yet numbered. A tree edge does not affect the *lowlink* parameter but indicates that the search must proceed deeper on. Forward edges are characterized by the fact that w is already numbered and *number*(v) ≤ *number*(w); they do not affect SCCs. Back edges and cross edges in the same SCC are characterized by *number*(v) > *number*(w) and $w \in$ *stack*. They affect the *lowlink* parameter in the following way: if *number*(w) is smaller than *lowlink*(v), then *lowlink*(v) becomes *number*(w). Tarjan has demonstrated that v is the root of an SCC if and only if *lowlink*(v) = *number*(v).

The simulation engine uses Tarjan's algorithm to detect sets of simultaneous equations. It dynamically builds and explores a graph, called *scalar graph*. The scalar graph is similar to the dependency graph used by classical simulation algorithms, but it differs in three ways. First, vertices of the scalar graph are scalar variables (and not variables). This is because the scalar variables of the same variable do not have necessarily the same defining expression. Second, the edges of the scalar graph are determined dynamically because of the presence of conditions in the PWDEs. Third, edges are in the reverse order: successors of a scalar are the scalar variables which appear in the expression that defines it. This is because in Tarjan's algorithm, the SCC of a vertex is determined after all the SCCs of its successors have been discovered. This way, the set of equations associated with an SCC are solved after the scalar variables appearing in the right-hand side of these equations have been computed.

The algorithm of the simulation engine is presented below.

```
variables
  i: integer
  v: scalar_variable
  stack: stack of scalar_variable

procedure simulation_engine(m: model)
begin
  i ← 0;
  stack ← {};
  foreach v in unknown_scalar_variables(m) do
    begin
      set_value(v,UNKNOWN);
      set_number(v,0);
      set_lowlink(v,0);
    end;
  foreach v in unknown_scalar_variables(m) do
    if number(v) = 0 then compute_scalar_variable(v)
end

procedure compute_scalar_variable(v: scalar_variable)
variables
  w: scalar_variable
  scc: list of scalar_variable
  p: pwde
begin
  if value(v) = UNKNOWN then
  begin
    i ← i + 1;
    set_number(v,i);
    set_lowlink(v,i);
    push(v, stack);
    p ← scalar_pwde(v);
    foreach w in scalar_variables_of_pwde(p) do
    begin
      if value(w)=UNKNOWN and number(w)=0 then
      begin
        compute_scalar_variable(w);
        set_lowlink(v,min(lowlink(v), lowlink(w)))
      end
      else
        if number(w) < number(v) and w ∈ stack then
          set_lowlink(v,min(lowlink(v), number(w)))
    end;
    if lowlink(v) = number(v) then
    begin
      scc ← {};
      while stack≠{} and number(head(stack)) ≥ number(v) do
        insert(pop(stack), scc);
      equation_solver(scc)
    end
  end
end
```

Below is given a description of the functions and procedures which are used in the algorithm but not defined:

- functions value, number, and lowlink (resp. set_value, set_number, and set_lowlink) respectively return (resp. set) the value, the *number* and *lowlink* parameters of a scalar variable.
- function unknown_scalar_variables(m: model) → **set of** variable returns the set of unknown scalar variables of model m.
- function scalar_pwde(v:scalar_variable) → pwde returns the scalar PWDE defining scalar variable v.
- function scalar_variables_of_pwde(p: pwde) → **set of** scalar_variables returns the set of scalar variables contained in the right hand side of PWDE p. If this one contains a condition, this one is recursively evaluated using compute_scalar_variable.
- procedure equation_solver(scc:**set of** vertex) passes the set of equations associated with scc to the equation solver. As a side effect, it assigns to the scalar variables of the scc the solution determined or UNKNOWN if no solution is determined.

The termination of the simulation algorithm depends on those of equation_solver. Regarding complexity, since Tarjan's algorithm is linear in time relatively to the number of vertices, the simulation engine is also linear in time relatively to the number of scalar variables.

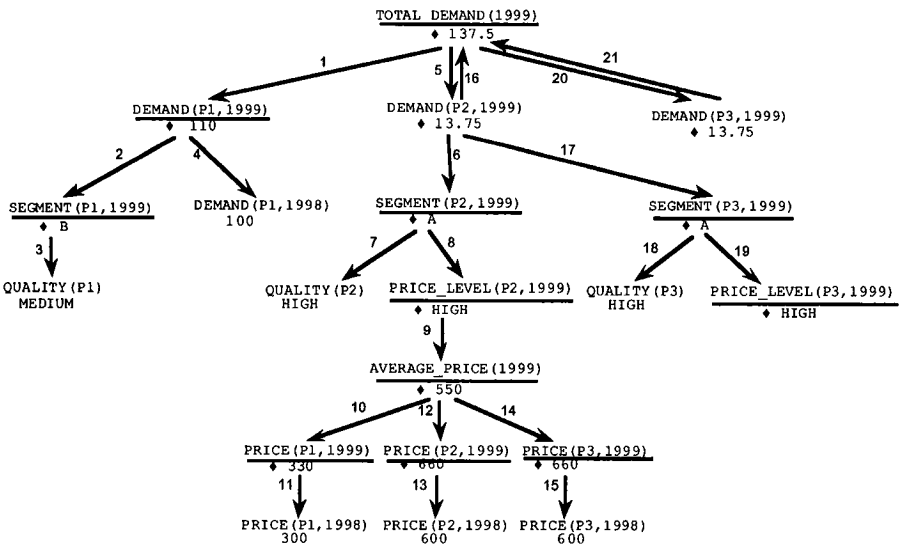

Figure 2: simulation of the market problem. Labels along edges correspond to depth-first search order. Underlined variables correspond to a root of an SCC. Values computed by the equation-solver are preceded by: →.

Let us illustrate the functioning of the simulation algorithm on the market example. Let us assume that a simulation is performed for computing the value of TOTAL_DEMAND(1999). This scalar variable is defined by PWDE (1), which has no condition part. The successors of TOTAL_DEMAND(1999) are DEMAND(P1,1999), DEMAND(P2,1999) and DEMAND(P3,1999). DEMAND(P1,1999) is not yet explored. The depth-first search thus proceeds on this scalar variable, following edge 1 in figure 2. DEMAND(P1,1999) is defined by PWDE (2). This PWDE has a condition which is first evaluated. compute_scalar_variable(SEGMENT(P1,1999)) is thus run (edge 2). SEGMENT(P1,1999) is defined by PWDE (6). compute_scalar_variable(QUALITY(P1)) is hence run (edge 3). QUALITY(P1) = MEDIUM is an input variable. The condition of the PWDE defining SEGMENT(P1,1999) evaluates to FALSE. The expression defining SEGMENT(P1,1999) is thus: B, which has no successor. SEGMENT(P1,1999) is hence the root of the SCC corresponding to the system of one equation {SEGMENT(P1,1999) = B}. This equation is passed to the equation solver which assigns B to SEGMENT(P1,1999). The simulation engine then returns to the evaluation of the condition of DEMAND(P1,1999). The expression defining this scalar variable is 1.1*DEMAND(P1,1998). DEMAND(P1,1998) is given, so {DEMAND(P1,1999) = 1.1*DEMAND(P1,1998)} is transmitted to the equation solver which computes: DEMAND(P1,1999) = 110. The algorithm proceeds in the same manner with the second successor of TOTAL_DEMAND(1999), i.e., DEMAND(P2,1999) until the algorithm reaches edge 16. At this point, since 16 is a back edge, DEMAND(P2,1999) is left on the stack and DEMAND(P3,1999), the third successor of TOTAL_DEMAND(1999), is examined. Edge 21 is also recognized as a back edge and when DEMAND(P3,1999) has been explored, the SCC {TOTAL_DEMAND(1999), DEMAND(P2,1999), DEMAND(P3,1999)} is detected. The set of equations:

{DEMAND(P2,1999)= 0.2*TOTAL_DEMAND(1999)/2;
DEMAND(P3,1999)= 0.2*TOTAL_DEMAND(1999)/2;
TOTAL_DEMAND(1999)=
110+DEMAND(P2,1999)+DEMAND(P3,1999)}

is sent to the equation solver presented below.

The equation solver

The equation solver is a set of classical numerical methods for solving sets of numerical equations. An appropriate method is chosen according to the set of equations transmitted by the simulation engine. For a set consisting of only one equation in which the left-hand side variable does not appear in the right-hand side variable, the right-hand side is simply evaluated. For a set of

linear equations, a gaussian elimination is used. For a set of non-linear equations, the Leverberg-Marquardt algorithm is used. Since there exists no general algorithm for solving sets of non-linear equations (using floating-point arithmetic), the termination of the equation solver is not guaranteed. The equation solver also handles equations involving symbolic variables, but not systems of simultaneous symbolic equations.

If the equation solver does not successfully solve a system of equations, failure is reported to the simulation engine which propagates unknown values on every variable dependent on one of the variables belonging to the unsolved set.

Model management in AMIA

AMIA (Page 1996) has been designed for building large discrete-time knowledge-based simulation systems. AMIA contains an advanced model management system including (Figure 3):

- a graphical environment supporting the modeling and simulation process. It helps the modeler in specifying and building a model, performing simulations, managing data and simulation results. It also includes specialized graphical editors.
- a code translator which compiles an AMIA model to C in order to optimize simulation performance.
- a LATEX generator which produces a document of the model.

AMIA has been successfully used in the development of different knowledge-based discrete-time simulation systems, in various domains. One of them, developed for energy demand forecasting in Europe (the MEDEE models family (Camos, Dumort and Valette 1986) is very large (more than 1.000 equations).

AMIA runs on UNIX platforms and on PC under Windows 3.x and Windows 95. AMIA can freely be obtained from the authors.

Conclusion

In this paper we have presented AMIA, a workbench for developing knowledge-based discrete-time simulation systems. AMIA is innovative in two respects: first it uses an algebraic modeling language for combining numerical and symbolic knowledge. Second, it uses a new algorithm able to exploit this combination of numerical and symbolic knowledge.

References

Widman, L.; Loparo, K.; Nielsen N. eds. 1989. *Artificial Intelligence, Simulation, and Modeling*. J. Wiley & Sons.

Kowalik, J. ed., 1986. *Coupling symbolic and numerical computing in expert systems*. North-Holland.

Holsapple, C.; Whinston, A. 1988. *Manager's Guide to Expert Systems Using Guru*. Dow-Jones-Irwin.

Klein, M.; Methlie, L. 1995. *Knowledge-based decision support systems*. J. Wiley & Sons.

Reboh, R.; Risch, T. 1986. SYNTEL: knowledge programming using functional representations. In: *AAAI-86*, 1003-1007.

Gelman, A.; Altman, S.; Pallakoff, M.; Doshi, K.; Manago, C.; Rindfleisch, T.; Buchanan, B. 1988. FRM: An Intelligent Assistant for Financial Resource Management. In: *AAAI-88*, 31-36.

Fourer, R.; Gay, D.; Kernighan, B. 1990. A Modeling Language for Mathematical Programming. *Management Science* 36 (5), 519-554.

Brooke, A.; Kendrick, D.; Meeraus, A. 1988: *GAMS: a User's Guide*. Scientific Press, Redwood City, CA.

Michel, L.; van Hentenryck, P. 1996. A modeling language for global optimization. In: *Proc. of PACT-96*, London, UK.

Boudis, M. 1997: *Simulation et systèmes à base de connaissances. Ph.D.* Dissertation, Univ. Mendès France, Grenoble, France, (in French).

Tarjan, R. 1972: *Depth-First Search and Linear Graph Algorithms*. SIAM Journal of Computing 1 (2), 146-160.

Page, M. 1996. *AMIA 3.0: manuel utilisateur*. Tech. Rep. 176, Univ. Mendès France, Grenoble, France (in French).

Camos, M.; Dumort, A.; Valette, P. 1986: *MEDEE 3: modèle de demande en énergie pour l'Europe*. Technique & Documentation-Lavoisier, Paris, France (in French).