Solution of Symbolic Linear Systems in OmSim Using Cramer's Rule

Carpanzano, Emanuele; Formenti, Fabio

1994

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Carpanzano, E., & Formenti, F. (1994). *Solution of Symbolic Linear Systems in OmSim Using Cramer's Rule*. (Technical Reports TFRT-7524). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
2

ISSN 0280–5316
ISRN LUTFD2/TFRT--7524--SE

Solution of symbolic linear systems in Omsim using Cramer's rule

Emanuele Carpanzano
Fabio Formenti

Department of Automatic Control
Lund Institute of Technology

October 1994

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* INTERNAL REPORT
	Date of issue October 1994
	Document Number ISRN LUTFD2/TFRT--7524--SE
Author(s) Emanuele Carpanzano, Fabio Formenti	*Supervisor* Sven Erik Mattsson
	Sponsoring organisation

Title and subtitle

Solution of symbolic linear systems in Omsim using Cramer's rule

Abstract

The efficiency of the simulations, of a modelling language, like Omola, depends strongly on the way in which the studied phenomenon is mathematically described.In fact, a symbolic manipulation of the system of equations that make up the model can be executed, in order to obtain faster and exacter simulations. The main purpose of the present work is to create a data structure and an alghoritm, which support the symbolic manipulation and improve the simulations of an object oriented modelling language, like Omola, particularly we realized a symbolic linear system solver, using Cramer's rule. In order to achieve this aim, three different data structures have been introduced: the sparse matrix, the mask and the tree. With the mentioned data structures it's possible to achieve the desired purposes; in fact a linear system solver is implemented as a function of the tree, and is used to transform systems of differential algebraic equations (DAE) into systems of ordinary differential equations (ODE) , in order to improve the simulations executed with Omola; particularly this alghoritm has been tested by performing simulations of mechanical systems. As a result of this work we have new data structures, that allow to obtain a different and interesting representation of dynamical systems, with which it's possible to have a clearer description of the system itself and to support the symbolic manipulation, particularly it's possible to implement a symbolic linear system solver. The considered data structures and functions have been implemented in C++, the program has been written in a way as much flexible, modular and reusable as possible, to permit an efficacious reuse and an easy extension of it, in order to solve new problems in future.

Key words

symbolic manipulation, linear systems, Cramer's rule

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280–5316		*ISBN*
Language english	*Number of pages* 43	*Recipient's notes*
Security classification		

Contents

1 Introduction **2**

2 Problem description and solution approaches **4**

3 Sparse Matrix implementation **5**

3.1 implementation 5

3.1.1 class SparMatrix 6

3.1.2 class SparVector 7

3.1.3 class SparVecel 9

3.2 Comments 10

4 Mask **11**

4.1 implementation 11

4.1.1 class Mask 11

4.1.2 class PresDev 15

4.2 Comments 16

5 The tree **17**

5.1 The basic idea 17

5.2 Extensions of the fundamental tree 24

5.3 The basic operations on the tree 27

5.4 Special functions of the tree 28

5.5 Use of the tree to solve linear systems 29

5.6 Interpretation of the tree 29

5.7 The implementation 33

5.8 Limits and possible extensions of the considered data structure 37

6 Simulations **39**

6.1 Particle Pendulum 40

6.2 Two Particle Pendulum 41

7 Conclusions **43**

of the tree (Section 5), and is used to transform systems of differential algebraic equations (DAE) into systems of ordinary differential equations (ODE) , in order to improve the simulations executed with Omola; particularly this alghoritm has been tested by performing simulations of mechanical systems (Section 6).

As a result of this work we have new data structures, that allow to obtain a different and interesting representation of dynamical systems, with which it's possible to have a clearer description of the system itself and to support the symbolic manipulation, particularly it's possible to implement a symbolic linear system solver, (Section 7).

The considered data structures and functions have been implemented in C++, the program has been written in a way as much flexible, modular and reusable as possible, accordingly to the rules of the object oriented programming, to permit an efficacious reuse and an easy extension of it, in order to solve new problems in future, (Appendix).

avoid if we instead for triangularize A in $A \cdot x = b$, we diagonalize it, in other words we calculate the elements of A^{-1}. The drawback is that is more complex procedure. However, it is a feasible approach if the system is of low dimension or if A is sparse. We will see that this is our situation so the Cramer's rule approach is more appropriate.

3 Sparse Matrix implementation

We will first consider the design and implementation of a class to support represenattion of a sparse matrix. A matrix, to be considered sparse, must have few non zero entries compared with the total number of elements, dealing with such a matrix leads to the problem of storing its elements avoiding allocation of memory for unuseful informations, If the matrix is large the problem must be solved changing the intuitive way of storing a matrix in another, less intuitive but more efficient for this kind of matrix. At the same time it is much easier, for the user, to see the matrix always in the same way, so the purpose of this data stucrure is to hide, the efficient implementation of the matrix, leaving an easy interface to the user.
The simple idea, to implement the sparse matrix, is to store only the non zero elements.Each vector of the matrix, both a row or a coloumn, is represented simply as a list, this list contains the non zero entries and their position. Each non zero element of the matrix will appear both in the vector, corresponding to its row, and in that corresponding to its coloumn. This is not an optimal solution, from the memory point of view, but it makes quicker to scan the elements of a row or a coloumn. Moreover in each vector the element itself, which is an Omola expression, is not present but only a pointer to it. Another project decision is to keep the lists ordered, this is useful since, in this application, the lists are used more to search elements than to insert or delete.
The last general consideration is about the interface of this structure: in order to make useful this structure as support of determinant calculation some feature had to be added, we considered some of them to be useful also for other uses of the matrix, so we included them in the interface.

3.1 implementation

In this section we present , what is the basic architecture of this structure and give information about the most important routines.

in that position there is a zero, so a pointer to a new expression equal to zero is returned.

function EndOfEntryRow (in `row:int`):boolean;
Returns true when there are no more entry in the row.

function EndOfEntryCol (in `col:int`):boolean;
Returns true if there are no more entry in the coloumn.

function GetNextEntryRow `(in row:int):Expression*;`
Returns the position of the current entry, in the specified row, and move the current entry on the next entry in the row. It is an error to call this routine if the corresponding EndOfEntryRow() is false.

function GetNextEntryCol (in `col:int`):Expression*;
Returns the position of the current entry, in the specified col,and move the current entry on the next entry in the col. It is an error to call this routine if the corresponding EndOfEntryCol() is false.

function GetRowNum ():int;
Returns the number of rows present in the matrix.

function GetColNum ():int;
Returns the number of coloumns present in the matrix.

function BuildTheMatrix (in `LIST(EquInst)* equations;`
`in LIST(VarInst)*variables`):SparVector;
Scans the list equations, to each is applied the FactorLinearPart routine of the class EquInst, with the coefficients returned by this routine a row of the matrix is filled, the rempart is inserted in the returned SparVector. This routine is useful to build the system of linear equations:$Matrix \times variables = Vector$.

private the class is composed of two lists of SparVector.
LIST(SparVector) rows;
LIST(SparVector) cols;

end SparMatrix;

3.1.2 class SparVector

class SparVector This is the structure used to implement a vector of the matrix. As in SparMatrix with the term entry we refer to an, explicitly inserted, element of the SparVector.

function Next_non_zero ():unsigned;
Returns the position of the next non zero entry in the vector, the current entry is moved on the next entry. It is an error to call this routine when EndOfEntry() is true.

function GetCurEntry ():unsigned;
Returns the position of the current entry.

function GetIndex ():unsigned;
Returns the identifier of the vector.

private the class is a list of Sparvecel,an iterator to Scans the list and an integer for the index.
LIST(SparVecel);
ITERATOR(SparVecel);

end SparVector ;

3.1.3 class SparVecel

class SparVecel This simple class contains the pointer to the element and the position.

uses Expression;

public procedure SetPosition (in pos:unsigned);
Seta the position to pos.

procedure SetPointer (in elem:Expression*);
Sets the pointer to the specified expression.

function GetPosition ():unsigned;
Return the position

function GetElem ():unsigned;
Returns the pointer to the element

private the class is simply composed of an integer and a pointer to in expression.

end SparVecel ;

4 Mask

During the calculation of the determinant the algorithm has to work on submatrixes of the original matrix, this could make the algorithm more difficult, it should have to take care of details such as: if an element belong to the desired submatrix,what is its position in the current submatrix exct. These reasons convinced us, of the necessity to have a class which takes care of these details, so that the the main algorithm is easier to write and to understand. The main functionality of this class is to operate on submatrixes of a given matrix, showing to the user only the desired part of the matrix. Moreover we decide that was better to encapsulate in this class two of the typic operation, needed during the calculation of the determinant. The first is the choose of the row, or coloumn, used to develope a subdeterminant. The second is the calculation of order two determinant.
The main consideration that lead us to the final structure is that, even if different operation are requested to this class, the basic information used are the components of the matrix, the class itself is no more than a matrix in wich rows and columns are erased. This consideration convinced us to use as base, on which build the new class, the SparMatrix class dicussed in the previous section. This is obtained simply making the class Mask an heir of the class SparMatrix, to this base we added two other, very simple, structures which aim is to bookkeep which are the rows and the column present in the current submatrix and if the vector is currently used to develope the determinant.

4.1 implementation

In this section we present the structure and the interface of the class Mask. In this class we introduced a current vector to make easier the interface of the class.

4.1.1 class Mask

class Mask ;

In this class, where no specified, all the indices reffering to rows and-columns have to be considered indices of the original matrix and not of the current submatrix.

inheret SparMatrix];

procedure EndOfVector ();
: This routine return true when the scan of the current vector, in the current submatrix, is finished.If this routine returns true implicitly the current entry, of the current vector, is set on the first entry. The current vector is also deleted from the list of vector used to develope the determinant.

procedure Translate (in `i, j:int;out curi, curj:int`);
: In curi and curj this routine return the position, in the current submatrix, of the element (i, j) in the original matrix. An obvious precondition to this routine is that the row i and the column j must belong to the current submatrix.

procedure GetNext (out `z, v:int;out r, c:int`);
: This routine return in z, v the position, in the original matrix, of the the next element in the current vector;in r and c is returned the position, of the same element, in the current submatrix.A precondition for this routine is that EndOfVector() is false.

procedure DefineMask (in `Abrows,Abcols:LIST(int)*`);
: In Abrows and Abcols this routine receive a list of rows and coloumns that have to be erased from the matrix. The effect of this routine is, firstable to reset the situation of the mask to the original matrix. After this operation, the specified rows and columns are erased from the matrix. An important consideration about this routine is that it is not equivalent to a sequence of calls to Select and Paint, even if this sequence leads to the same submatrix. Infact none of the erased rows and columns has been selected, this means simply that is an error to call a Clean, for a row and a column which are in the lists passed as parameters. To call this routine passing two void lists, or two nil pointers, as the effect to set the mask to its initial situation.

function Valdet2 (in `R1,R2,C1,C2:int`):Expression*;
: This routine returns a pointer to the expression of the determinat $det = elem(R1, C1) \times elem(R2, C2) - elem(R1, C2) \times elem(R2, C1)$. The expression is built using copies of the elements in the specified positions.

function IsDevRow (in `i:int`):boolean;
: Return true if the specified row was choosen by Select(), and since that all the call to EndOfVector(), with current vector this row,

two lists of PresDev. Moreover a pointer is used to indicate the current vector, and a boolean is true when the current vector is a row.

end Mask;

4.1.2 class PresDev

class PresDev ;
: this class contain the information necessary to the class mask for each vector.

uses boolean;
: This type is imported from the Omsim library defs.H.

public type PresDev ;

- **Procedure** SetPresent();
  set the vector, to which is associated, as present in the current submatrix.
- **Procedure SetAbsent** ();
  set the vector, to which is associated, as absent from the current submatrix.
- **Procedure SetDevelope** ();
  is called from Mask::Select() and store the information that the corresponding vector is used to develope the determinant.
- **Procedure ResetDevelope** ();
  is called from Mask::EndOfVector() and means that the corresponding vector is no longer in the list of vector used to develope the determinant.
- **function IsPresent** ():boolean;
  return true if the corresponding vector is in the current submatrix.
- **function IsDevelope** ():boolean;
  return true if the corresponding vector is in the list of vectors used to develope the determinant.

private the class is composed simply by two boolean;

end PresDev ;

5 The tree

The aim of this data structure is to allow the calculation of the determinant and of the needed subdeterminants, algebraic complements and elements of the inverse matrix, of a symbolic matrix, whose elements can be constants, parameters, variables and functions. It is necessary to execute these calculations in an efficient way, otherwise , specially with large matricies, they imply too complex operations, which require too long times and too much memory to be performed.

Particularly, we used the tree to transform a system of differential algebraic equations (DAE) in an equivalent system of ordinary differential equations (ODE),in order to achieve a better formulation of the system, which allows to improve the speed and the precision of the simulations of the cosidered system, executed with Omola.

5.1 The basic idea

All the above mentioned problems can be lumped together in the following simple problem : calculate the needed elements of the inverse, of a certain symbolic matrix, in the most efficient way. In other words this means that we have to calculate the needed algebraic complements, i.e. the needed subdeterminants, and the determinant of the matrix, in the most efficient way.

To achieve our purpose it's necessary to calculate and to allocate in memory every needed subdeterminant of the symbolic matrix only once, and to be able to refer to it every time it is required. From this it follows that we can subdivide the considered problem into the simpler subproblems:

1-how represent a certain subdeterminant;
2-how represent the value of a certain subdeterminant;
3-how calculate the value of a certain subdeterminant;
4-how find a certain subdeterminant.

Let's now give a look at the ideas we had to resolve these problems, before explaining in detail how we implemented the resolutions.

To represent all the known subdeterminants a particular data structure is defined, that is called tree, because this structure can be illustrated with the use of graph theory and, particularly, it can be represented by a specific graph called tree.

value : pointer to the expression which gives the value of the following moltiplication :

$$(-1)^{(csrow+cscol)} \times element(rowX, colY) \times subdet$$

where subdet is the determinant of the submatrix represented by the considered node, it is to notice that in some particular cases the pointer points directly to the determinant of the present submatrix, as happens, for example, for the first node of the tree (the root of the tree);

detvar : pointer to the variable associated to the expression pointed by value, which can be used whenever the considered expression is required, instead of the expression itself, it is obvious that the use of this variable allows to simplify the calculations which involve the expression in question;

subdeterminants : list of pointers to the nodes of the tree which represent the submatricies, of minor order, whose determinant is needed to calulate the determinant of the submatrix associated with the considered node, in fact, this one is given by the addition of the values of the nodes pointed by the pointers of the discussed list.

Now that we know how a single node looks like, it's possible to study the structure of the tree, by considering the following example.

Example : the fundamental tree

Let's suppose that we have the matrix of order 5 shown in figure 2 .

The correspondent fundamental tree,which allows to represent and to calculate the determinant(and all the needed subdeterminants) is represented in figure 3. In this figure we've got that:

$$\begin{aligned}
\text{val12} &= Z \\
\text{val11} &= a \\
\text{val10} &= 3 \times Z - a \\
\text{val9} &= -b \\
\text{val8} &= 4 \times Z - b
\end{aligned}$$

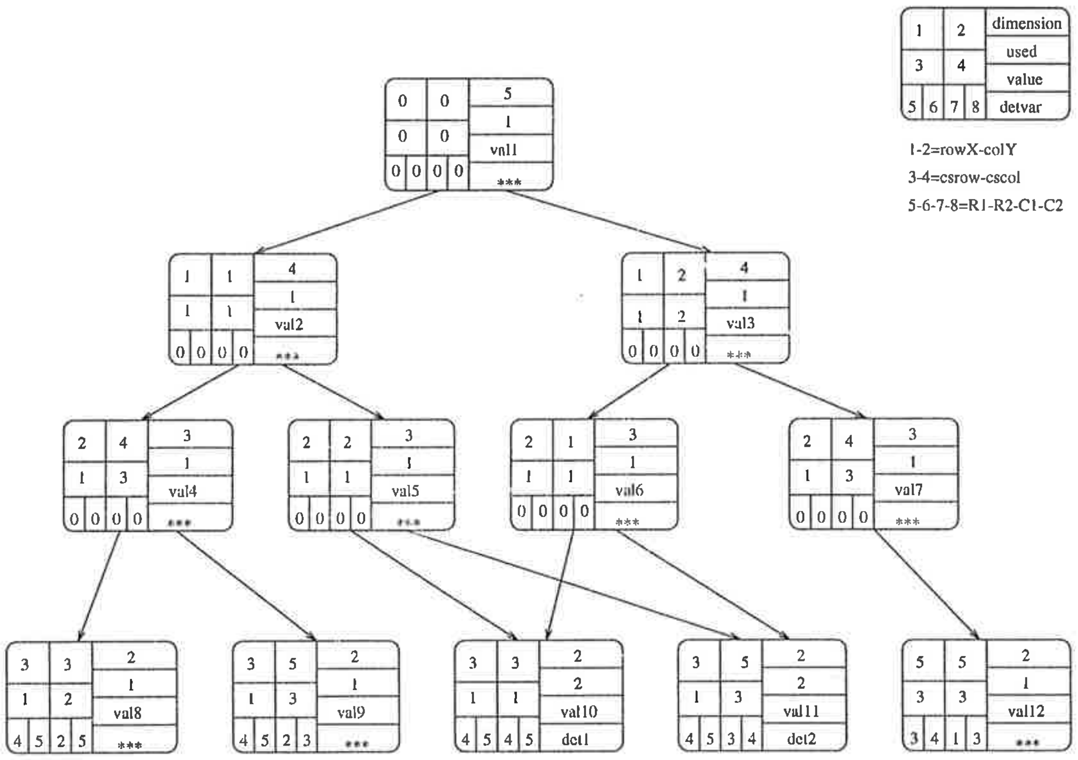

Figure 3: Tree correspondent to the matrix of figure 2.

1-representation of a certain subdeterminant : a certain subdeterminant of the symbolic matrix is associated to the corresponding submatrix, which is represented in the tree by a node; and the submatrix, represented by a node, is obtainable by cancelling, in the matrix, the rows and the columns indicated in the elements rowX and colY of each node crossed by going from the root to the considered node of the tree.

2-representation of the value of a certain subdeterminant : the value of a certain subdeterminant is represented, in the tree, by the part of the tree that starts from the node which represents the submatrix associated to the considered subdeterminant, this value, once calculated, can be stored in the expression pointed by the pointer value belonging to the discussed node of the tree.

3-calculation of the value of a certain subdeterminant : the value of a certain subdeterminant can easily be calculated, if the submatrix corresponding to the subdeterminant is represented by a node of the tree, by applying recoursevly the following simple rule : the value of a subdeterminant, associated to a certain node of the tree, is given by the addition of the values of the nodes pointed by the considered node, if the node corresponds to a submatrix of order major than two, else the value of the subdeterminant is given by the following operation : $((R1, C1) \times (R2, C2) - (R1, C2) \times (R2, C1))$, where R1,R2,C1 and C2 are the rows and columns that form the submatrix of order two corresponding to the node.

4-search of a certain subdeterminant : a subdeterminant can be searched by searching the corresponding submatrix, represented by a node of the tree; and the desired node of the tree can be identified by comparing the rows and columns, of the symbolic matrix, not present in the considered submatrix, with the rows and columns cancelled, step by step, during the construction of the tree, which are stored in the integer variables rowX and colY of each node of the tree; in fact, if we find a sequence of nodes, of the tree, whose cancelled rows and columns coincide with the rows and columns not present in the considered submatrix, then the last node of this sequence is the one that corresponds to the desired subdeterminant.

It's to notice that a node, which represents a submatrix whose determinant is null, is represented as shown in figure 5 .

10	W	0	0	0
a	1	0	Y	0
2	0	1	0	1
1	4	1	3	1
0	b	0	a	Z

MATRIX M

10	W	0	0	0	-2
a	1	0	Y	0	0
2	0	1	0	1	0
1	4	1	3	1	0
0	b	0	a	Z	0
0	7	0	0	0	V

MATRIX A

Figure 6: Matrix M is a submatrix of matrix A.

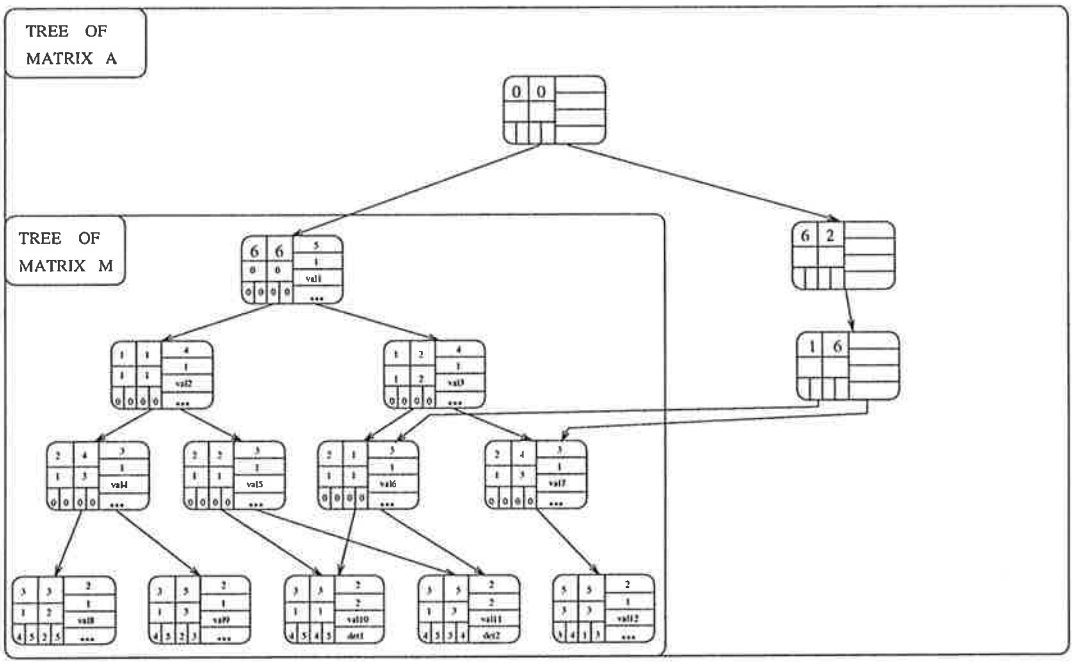

Figure 7: Tree correspondent to matrix A, which contains the tree correspondent to matrix M.

5.3 The basic operations on the tree

Now we will explain how to perform the basic operations on the tree, which are the following ones :

1-construction of the tree;
2-search for a node of the tree;
3-calculation of the determinant or of a subdeterminant represented in the tree.

Let's so start with the explanation of these operations.

1-Construction of the tree

Once initialized with the definition of a root, which can be a primary root (function Construction) or a secondary root (function SubDetConstructor), the construction, of a part of the tree, is executed by operating as follows for each new node (functions Calculatedet and CalculateSubDet) : if the submatrix associated to the new node is already represented in the tree, then the pointers of the list subdeterminants are pointed to the same nodes to which the pointers of the already existent node are pointed, (function Connect), else the most convenient row or column is selected (function select of the mask) and for every needed subdeterminant, of minor order, a new node is created and pointed by a pointer, of the list subdeterminants, (function Explode).

2-Search for a node of the tree

In order to find the node corresponding to the submatrix currently represented in the mask, this submatrix is compared with the submatricies present in the tree, if the same submatrix is found then the pointer SearchedPointer points to it else it points to nil, (function Search). The above mentioned comparison between the current submatrix and the submatricies represented in the tree, is performed by one of two different recoursive functions, depending on whether the dimension of the considered submatrix is bigger than the dimension of the matrix, divided by two, or not, this in order to make the comparison more efficient. Precisely the first, of this two functions, checks if the submatrix currently considered, is already represented in the part of the tree which starts from the node pointed by a certain pointer, by checking

procedures ShowDet, ShowAlgComp, ShowInvElem : these functions allow the user to get the values of the determinant, or of a known subdeterminant, or of an algebraic complement or of an element of the inverse, of the considered symbolic matrix, in an interactive way.

5.5 Use of the tree to solve linear systems

It's easy to understand that the illustrated data structure can be used in order to solve systems of linear equations, like the following one :

$$Ax = b$$

where A is a non singular square matrix of order n, while x and b are vectors of n elements, the first containing the unkowns and the second containing known values.

The solution of this linear system is given by :

$$x = A^{-1}b$$

This equation is solvable, in an efficient way, with the use of the tree, by finding, for every unknown variable of the vector x, the elements of the inverse of A, corresponding to the non zero elements of the vector b, and by executing the considered operation. In order to make these operations as efficient as possible, it's convenient to construct first the tree so that every needed subdeterminant of the matrix is represented in it, then the solution of the linear system is calculable by "reading" the values of the needed subdeterminants in the tree. Particularly, by operating in this way, it's possible to calculate only once the value of every needed subdeterminant and to store the values of the subdeterminants, needed more than once, in appropriate variables, that allow to reduce the computational complexity of the found solution. The illustrated operations are performed by the function of the tree ExpSystConstructor.

An application of this function, of the explained data structure, is studied in Chapter 6, where this function is used to transform DAE systems, which represent models of mechanical systems, into ODE systems, in order to support the symbolic manipulation and to improve the simulations in Omola.

5.6 Interpretation of the tree

The data structure here illustrated, gives a meaningful description of the considered system, in fact the value associated to each node of the tree can be interpreted as follows.

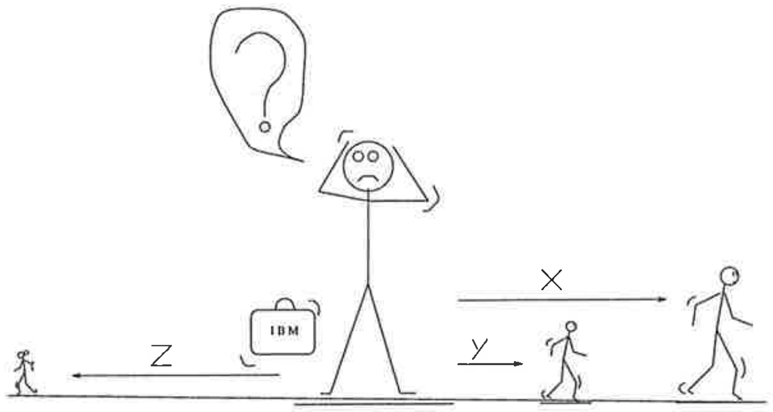

Figure 10: Big problems!

The engineer is able to control the distance from him of only one of the children, so he has to decide which is the best one to control in order to take care of all three of them, as good as possible. To solve this problem he decides to modelize it.

Model

By defining with X, Y and Z the distances of the children, the engineer obtains the following model:

$$
\begin{aligned}
(ab(c-d)(e-1))X &= b(c-d)\dot{X} + ade\dot{Y} - ab\dot{Z} \\
(ab(c-d)(e-1))Y &= b(c-d)\dot{X} + (a(c-d) - ace)\dot{Y} - ab\dot{Z} \\
(ab(c-d)(e-1))Z &= be(c-d)\dot{X} - ade\dot{Y} + ab\dot{Z}
\end{aligned}
$$

where : a,b,c,d,e are positive parameters.

This model is not clear enough to solve the problem ! Let's so try to manipulate this model.

Symbolic manipulation

First of all we can write the model in the standard way:

Ax=b

$$\dot{Z} = c(eX - Z) - d(eY - Z)$$

And this formulation of our system is much easier to interpret.

Interpretation of the system

We can see that dot X is positive when Z is bigger than X, this means that when Z is bigger than X when X increases, the interpretation of this fact is that the bigger brother follows the smallest, with speed proportional to the parameter a; in the same way we can see that the second brother follows the first one with speed proportional to b; finally we can notice that the youngest follows the biggest with speed proportional to c, but tries to go as far away as possible from the second with speed proportional to d, moreover the smallest child makes an error proportional to e in estimating the distances of his brothers from him.

It's obvious that now, by using the linear system solver to put our model in space state form, we've a much clearer description of our system, which allows to interpret easily the equations and the parameters of the system, as shown above. Now it's simple to solve the considered problem.

Solution of the problem

By considering that the first child follows the third one, and the second follows the first one, it's obvious that the complete system can easily be controlled by taking care of the smallest child.

5.7 The implementation

The considered data structure has been implemented in C++ simply with one class, which represents a single node of the tree; now we will describe this class and its functions in a simple and easy way, a more detailed description is situated inside the code, in section 6 .

The mentioned class can be illustrated with the TDN technique as follows:

procedure CalculateSubDet `(in : int X,Y)` : calculates the subdeterminant of order (n-1) obtained by deleting row X and column Y from the symbolic matrix

procedure Connect `( in : int X,Y,R,C )` : connects the subdeterminant to calculate to the same subdeterminant already calculated;

procedure Explode `(in : int X,Y,R,C)` : continues the calculation of a subdeterminant, if it isn't already known, by finding out all the subdeterminants of minor order needed to obtain the desired subdeterminant, and asking for their calculations;

procedure Search : compares the current submatrix with the submatricies whose subdeterminant has already been calculated, if the same submatrix is found then the pointer SearchedPointer points to it else it points to nil

procedure ExploreBig `(in : int cdim; DetXY* podetxy)` : checks if the determinant of the submatrix currently considered is already represented in the part of the tree which starts from the node pointed by podetxy, which correspondes to a submatrix of dimension cdim, by checking if the rows and column cancelled from the matrix are present in the rows and columns not considered in the current submatrix

procedure ExploreLittle `(in : int cdim; DetXY* podetxy)` : checks if the determinant of the submatrix currently considered is already represented in the part of the tree which starts from the node pointed by podetxy, which correspondes to a submatrix of dimension cdim, by checking if the rows and column cancelled from the matrix aren't present in the rows and columns considered in the current submatrix;

function ReadDet `(in : PointerToListOfInt AbRows,AbCols; in/out : LIST(EquBlock) & BLT1; LIST(VarInst)& createdvars)` returns a pointer to the value of the subdeterminant of the symbolic matrix, obtained by cancelling the rows and the columns, indicated in AbRows and Abcols, from the matrix, if it is possible to calculate the requested subdeterminant with the constructed tree, otherwise this function returns a nil pointer, and informs the user that the desired subdeterminant isn't known;

function ValDet `(in : DetXY* DetToCalculate; int rowx,coly;`

private :

dimension : dimension of the submatrix;

rowX-colY : last row and last column cancelled from the matrix;

csrow-cscol : positions of the last row and last column cancelled in the previous submatrix;

R1-R2-C1-C2 : rows and columns of the symbolic matrix that form the present submatrix of order two, if the current submatrix is of order maior than two then this integer variables are setted to zero;

used : number of times the subdeterminant, represented by the considered object, is used;

subdeterminants : list of pointers to objects of the class DetXY, i.e. to nodes of the tree, that represent the subdeterminants needed to calculate the value of the considered subdeterminant;

value : pointer to the expression which gives the value associated with the considered object of the class DetXY;

detvar : pointer to the variable that contains the value associated with the considered node of the tree, if it is convenient to store this value in a variable, otherwise this pointer points to nil.

5.8 Limits and possible extensions of the considered data structure

The data structure here illustrated can be improved both by making the present representation more efficient, and by adding new functions to this structure. For example it could be interesting to introduce more pointers for every node of the tree, in order to be able to visit the tree, not only in one single way (up-down), but in both the possible ways up-down and down-up, with this modification the complexity of the tree increases considerably, but the operations on it (search for a node, calculation of the value associated to a node, ecc.) could result much quicker; another possible way to improve the efficiency of this structure could be the introduction of a stack, in order to support the storing of the values associated to the nodes of the tree. On the other hand, there are also a lot of new functions that can be introduced, in order to make the tree more useful, for example functions can be needed to represent in the tree particular subdeterminants of the considered matrix

6 Simulations

In this section we show the results of the test for the data structures and the algorithm presented in this paper.
To test these modules a new version of Omsim has been set up, this new version support the solution of linear equations systems. This version has been used to simulate a set of Omola models, and the results compared with those obtained using another Omsim version.
The Omola models, used for the test, have been developed using a recently introduced mechanical library, [Per Anell: Modelling of Multibody System in Omola] this library is based on an approach [Otter et al., 1993] that makes possible to put in state space form, the models of mechanical system, if they do not have closed circuits.
This means that models, of tree structured systems developed using this library, can be manipulated an trasformed so that an ODE solver can be used.
The new Omsim version, to which we will refer as Omsim+, can perform this manipulation, so the same model is instantiated using Omsim+ and Omsim, and simulated using, respectively, an ODE solver and a DAE solver.
The results obtained are compared, both in term of trajectories of the variables, and in term of time. To make more significant the result it is useful to distinguish, for each model, between the time necessary for the simulation and the time necessary for the instantiation.
The Omsim+ algorithm for the solution of linear systems is used during the instantiation phase, so we expect an increased instantiation time if compared with the Omsim instantion time for the same model. During the simulation the situation is different, in this case Omsim+ has to simulate a simpler model than Omsim.
We have to test if the linear system solver is correct, after that we have to check if the longer instantiation time is compensated by the reduction of the simulation time. To check the correctness of the result we will use also a reference model, which is an Omola model written without using the mechanical library, but an ordinary approach, this means that it is simpler than the model developed using the library and its behavior must be considered the reference for the other simulations.

model	*instantiation time*	*simulation time*
Ref. model	1	2.8
Omsim	5	34.5
Omsim+ (Dasrt)	6	10.3
Omsim+ (Dopri45)	6	16.4

Table 2: Simulation time results for 100 sec with Particle Pendulum

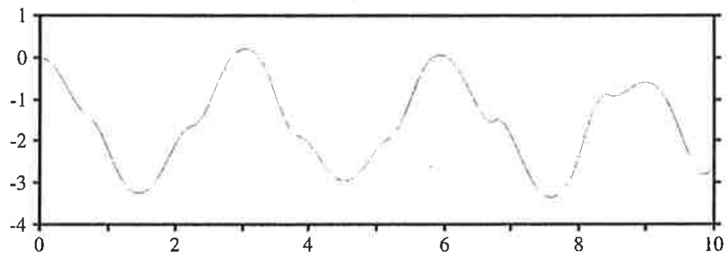

Figure 13: Two particle pendulum model simulated, 10 sec, variable *phi*1.

simulation using Omsim+ and Dasrt is three time speeder than the Omsim one, see table 2.

6.2 Two Particle Pendulum

In this section simulations of a double pendulum are presented.
Simulation on 10 seconds interval is presented, for the reference model, the model simulated using Omsim, and Omsim+, for this one both the Dasrt and Dopri45 are used. To use Omsim is necessary, before simulating, to enable the algebraic solver, this operation is not needed using Omsim+.

Even in this case the simulations result, obtained using Omsim+, are

model	*instantiation time*	*simulation time*
Ref. model	1	2.1
Omsim	9	48.3
Omsim+ (Dasrt)	48	15.3
Omsim+ (Dopri45)	48	17.3

Table 3: Simulation time result for 10 sec with Two Particle Pendulum

7 Conclusions

The results of our work can be summarized by saying that we have created new data structures, which allow us to execute symbolic manipulations on a given system; particularly, with the use of these structures, it's possible to obtain a Symbolic Linear System Solver. We have shown how the symbolic linear system solver can be used to transform an explicitable model from implicit (DAE) to explicit (ODE) state space form, and we have also seen that this transformation gives us a clearer description of a system and permits to obtain a new version of Omsim (able to put in state space form explicitable models) which performs faster simulations, but requires longer instantiation times, when used.

Moreover, it has been illustrated that, the considered data structures, give a description of a system, which allows to understand the interactions between the subsystems forming the system, so we can say that the discussed structures describe a system in a new interesing way, therefore we hope that our work will be useful, for new and different purposes, in the future.