Omola -- An Object-Oriented Modelling Language

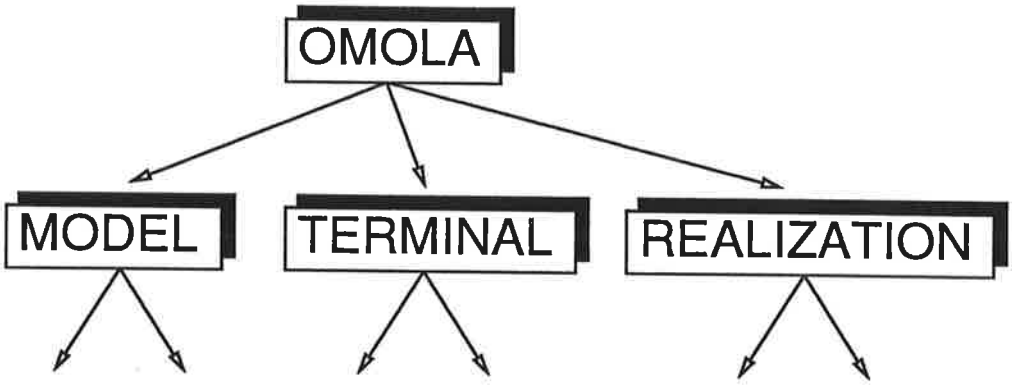
Andersson, Mats

1989

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

CODEN:LUTFD2/(TFRT-7417)/1-018/(1989)

Omola – An Object-Oriented Modelling Language



Mats Andersson

Department of Automatic Control
Lund Institute of Technology
April 1989

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Report

Date of issue
April 1989

Document Number
CODEN:LUTFD2/(TFRT-7417)/1-018/(1989)

Author(s)
Mats Andersson

Supervisor

Sponsoring organisation
The Swedish Board of Technical Development

Title and subtitle
Omola – An Object-Oriented Modelling Language

Abstract

This report presents a new language for structured dynamic models. The language is based on ideas from object-oriented programming. Models are represented as classes with attributes. Inheritance and hierarchical submodel decomposition improves model structure and facilitates resuse of models. The language is designed to be general and extendable in order to represent future, yet unpredicted, model representation concepts.

Key words
Computer Aided Control Engineering, object-oriented, modelling language

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
18

Recipient's notes

Security classification

1. Introduction

During work on the CACE project "Tools for model development and simulation" it has become more and more obvious that a new formal model language is needed. Such a language must be powerful enough to express all new model structuring concepts introduced in the project. Also, it is desirable that the language is general enough for future needs to express yet unpredicted modelling concepts. It has been clear that many model structuring concepts resemble concepts in object oriented programming. It has been equally clear that most of the objectives of model development are the same as those of program development. Among these objectives are fast development of new models, reuse of existing models and secure check of model consistency and correctness. Based on these observations and on experiences gained during the development of new modelling concepts, the new modelling language Omola – Object-oriented MOdelling Language – was designed.

In the remaining part of this intoduction we will give a brief presentation of some concepts for model structuring. In the following sections we will first introduce Omola by some simple examples. Then there will follow a more formal and detailed description of the language. Finally, there are some discussions about Omola as a part of CACE environment, different interpretations of Omola code and how the language can be extended in various directions.

Model structures

Models of real plants tend to become very large. Hundreds of states and equations are not unusual. To build and to understand such a large model is very difficult unless it can be divided into smaller parts that can be analyzed and comprehended separately. Many technical systems are built from standard components connected by electric wires, pipes, shafts etc. Naturally, it is convenient to describe models as connected submodels in a similar fashion. Models of standard components can be saved in libraries and reused. A modelling language supporting hierarchical submodel decomposition, called DYMOLA, is proposed in [Elmqvist, 1978]. The model structuring concepts considered in this report are based on hierarchical submodel decomposition, structured terminals and multiple realizations. These concepts are explained very briefly here. For a more comprehensive discussion see [Mattsson, 1989b] and [Andersson, 1989].

Model is the main structural entity. A model has a number of *components* defining its interaction with the environment and the user, its behaviour and other attributes.

Terminal is a model component for defining the model's interaction with other models. There are different kinds of terminals, for example, simple terminals and record terminals. A simple terminal corresponds to a scalar variable and has attributes describing physical aspects of the terminal. A record terminal, represents interaction which involves a set of variables.

Realization is a component defining the behaviour of the model. A model can have multiple realizations and there are different kinds of realizations for different ways of describing model behaviour. Some of these are:

- A *primitive realization* defining model behaviour as a set of differential algebraic equations.
- A *transfer function.*

- An *ABCD-form* for linear state space representations.
- A *structured realization* defining behaviour as a set of connected submodels. This kind of realization is the foundation of the hierarchical model decomposition.

Parameters are attributes of models and realizations making it possible to adapt the model for different purposes. A parameter is a (simulation-) time invariant variable that can be changed by the user.

Submodel is a model used as a component of a structured realization. Any model can be used as a submodel.

Connection is a component of a structured realization representing a relation between terminals of the super-model and of the submodels.

2. Model Representation in Omola

In this section we will introduce omola Omola by a number of small examples.

Omola is an object-oriented modelling language. Concepts and terminology from object-oriented programming will be used in following. See [Stefik et al., 1986] for a brief introduction to object-oriented programming or [Mayer, 1988] for a comprehensive one.

Models and model components like terminals and parameters are represented as classes. Every class is a descendant (subclass) of some previously defined class from which it will inherit properties. We can assume that there are predefined super-classes for models (called Model) and different component types (e.g. Terminal, Realization) in the system. Here is an example of a model definition:

```
Tank IS A Model WITH
  terminals:
    inflow IS A Terminal;
    outflow IS A Terminal;
  parameters:
    tank_area := 5.0;
    outlet_area := 0.05;
END
```

The tank model is defined as a subclass of Model with four local attributes: `inflow`, `outflow`, `tank_area` and `outlet_area`. The first two attributes are subclasses of Terminal without any local attributes. The other two attributes are type defined variables with default values. A default value can be changed in a subclass of Tank or in a tank instance involved in a simulation.

The Tank model defines only the model interface and not the model behaviour. In order to get a tank model with some description of its dynamic behaviour we make a new definition:

```
Tank2 IS A Model WITH
  terminals:
    inflow IS A Terminal;
    outflow IS A Terminal;
  parameter:
    tank_area := 5.0;
    outlet_area := 0.05;
```

```
  realization:
    re IS A  Primitive WITH
      variable:
        level := 0.0;
      equations:
        tank_area * dot(level) = inflow - outflow;
        outflow = outlet_area * sqrt(level);
    END;
END;
```

In this new tank model we have added one more component – a realization. The realization is a subclass of Primitive which indicates that the behaviour of this model is defined as a set of differential equations. A primitive realization typically have attributes which are variables and equations. The equations involve parameters and terminals of the model, variables and time derivatives of variables. In this example "`dot(level)`" indicates the time derivative of the level.

In the new tank model, only the additional realization attribute is different from the original tank model; the model interface is the same. In Omola we can instead use inheritance to define the new tank. In this case, the new tank is defined as a subclass of the original tank model. Tank3 will inherit all attributes from Tank and add a realization. The Tank3 model will be identical to the Tank2 model.

```
Tank3 IS A Tank WITH
  realization:
    Re IS A  Primitive WITH
      variable:
        level := 0.0;
      equations:
        tank_area * dot(level) = inflow - outflow;
        outflow = outlet_area * sqrt(level);
    END;
END;
```

One advantage of using inheritance in this case is that we can define a number of different tank models inheriting the same interface from Tank. If a tank is used as a component of a larger system, we can easily exchange different tank models since they all have identical interfaces.

We shall extend the example a bit further. Assume we want to define a system with two connected tanks. We do this by defining Tank-System with a structured realization with two submodels:

```
Tank_System IS A Model WITH
  realization:
    Re IS A Structure WITH
      submodels:
        Tank_a IS A Tank3;
        Tank_b IS A Tank3;
      connections:
        Tank_a.Outflow AT Tank_b.Inflow;
    END;
END;
```

Here, the realization Re is defined as a subclass of Structure which is a special kind of realization. A structure is a set of connected submodels. The sub-

models and connections are defined as attributes of the realization. Tank_a and Tank_b, are components defined as subclasses of the previously defined Tank3. A special syntax is used to define connections.

New kinds of terminals can also be defined as classes. In the system there is a predefined terminal class called "SimpleTerminal". Even though it is predefined it could have been defined as something like:

```
SimpleTerminal IS A Terminal WITH
  attributes:
    value         TYPE real;
    default_value TYPE real;
    quantity      TYPE Quantity;
    unit          TYPE string;
    direction     TYPE (In,Out,Across);
    ...
END;
```

A SimpleTerminal has a number of different attributes which are type declared but with unknown values. We can now specialize this class and define the subclass VoltageTerminal, where some of the attributes are given default values:

```
VoltageTerminal IS A SimpleTerminal WITH
  quantity := Voltage;
  unit := "V";
  direction := Across;
END;
```

In a similar fashion we can define a current terminal. It is also possible to define structured terminals, for example, an electric terminal with a voltage and a current component. All structured terminals are subclasses of the predefined terminal class "StructureTerminals". A definition of an electric terminal might look like:

```
ElectricTerminal IS A StructureTerminal WITH
  components:
    I IS A CurrentTerminal;
    U IS A VoltageTerminal;
END;
```

3. The Design of Omola

It has been an attempt to design Omola in such a way that it is not limited to the model structuring concepts presented in previous section. Rather, Omola is based on a few very general concepts from object-oriented programming which can be used to represent high level model structures of various type. For this reason, is instructive to view Omola as divided into three distinct concept levels:

- The *model representation* level containing concepts like model, terminal, realization, etc.
- The *structure* level with concepts like class, component, attribute and inheritance. This level is defined by the formal syntax of Omola.

- The *data* level supporting the mathematical framework for model behaviour. It includes concepts like number, expression, matrix, polynomial, equation and function.

In the following sections we will present these levels of concepts in more detail by starting with the last one.

4. The Data Level

The data level of Omola contains low-level data types like integer number, real number, string, etc., as well as a number of higher level mathematical primitives for describing model properties and behaviour. As far as possible we have used MATLAB syntax [Moler et al., 1987] for mathematical objects. However, MATLAB has only one data type, the complex matrix, and other kinds of data such as polynomials and transfer functions have to be coded as matrices. On this matter we agree with [Rimvall, 1986] that "Structurally different data (e.g. a matrix and a transfer function) should be stored in separate data structures, however, data with only semantical variation (e.g. the state-matrices of a continuous and a discrete system) should be treated equally by the system". For this reason we need to invent some special syntax not present in MATLAB.

In this section we will present those data level objects that are special to Omola and not found in MATLAB or other programming languages. In the appendix, there is a complete list of all data types in Omola.

Polynomials Polynomials are indicated by brackets ("{}") and can be expressed in one of two different forms: coefficient form or root form. For example, the polynomial

$$p^2 + 2p + 3$$

on coefficient form is written in Omola as `{1,2,3}`, while the polynomial

$$2(p-3)(p-4)$$

is written as `{2: 3,4}` on root form.

Operator overloading for the common operators ($+$, $-$ and $*$) on polynomials is used. The division operator "/" indicates rational polynomials.

Equations Differential algebraic equations can be used to express model behaviour [Mattsson, 1989a]. We have chosen to use the equal sign "=" as the equation operator. A special operator ,dot, is used to express time derivatives. For example, `dot(x)` means dx/dt. Both scalar and matrix equations are allowed. Here is an example of a differential algebraic matrix equation:

```
E*dot(x) = A*x + B*u
```

where the constant matrices `A`, `B` and `E`, and the vector variables `x` and `u` are assumed to be declared. The matrix dimensions must agree for the equation to be valid.

Constraints are used to propagate parameter values between model components. A constraint is a directed relation between two parameters binding the value of the left hand side parameter to the value of the right hand side parameter. For example, when a composite (structured) model is defined, it is usually desirable to bind submodel parameters to parameters of the composite

model. Assuming we have two submodels S1 and S2 each with a parameter p which we want to be bound to the value of the parameter p of the composite model. This can be specified by the constraint expression (defining two constraints):

```
S1.p :- S2.p :- p;
```

where dot-notation is used to access parts of structured objects. This means that only the parameter of the super-model can be changed by the user and that this change is propagated downwards to the parameters of the submodels. Simple arithmetic expressions can be used in constraint expressions.

In future versions of Omola, also undirected constraints might be considered.

Connections are used in composite models to express relations between terminals. In most cases a connection is an equality relation between terminal variables and an equation would be an appropriate notation. However, for some types of terminals a connection implies a "zero sum" equation. For this reason we have adopted a special notation for all connections from Dymola [Elmqvist, 1978]:

```
<terminal> AT <terminal>
```

5. The Structure Level

The structure level contains the object-oriented aspects of Omola. It includes concepts for defining classes of objects with component and variable attributes. Concepts on the structure level as well as on the data level is reflected in the syntax of Omola. A formal definition of the Omola syntax is given in the appendix. The structure level of Omola is designed to represent object-oriented data structures of any kind, not just models. These general aspects of Omola are described in this section.

The primary structural entity in Omola is the *class*. As in object-oriented programming, a class defines properties common to a set of *instances*. Instances will be discussed later in this report.

A class has a name, a super class (sometimes called parent class) and possibly some local attributes. A class will inherit all attributes of the super class. A class definition with a class body of local attributes is written like:

```
<name> IS A <super class> WITH
  <body with local attribute definitions>
END
```

Class attributes

A class can define two kinds of attributes:

- *variable* attributes (or just variables) and
- *component* attributes (or just components).

Variable attributes correspond to instance variables in object-oriented programming. Entities from the data level previously discussed are used as values of variable attributes. A definition of a variable attribute in a class body looks like:

```
<name> TYPE <type name> := <initial value>
```

where the initial value is optional. The type name is the name of a built-in data type listed in the appendix. When an initial value is given, the "TYPE <type name>" part can be omitted whenever the type can be infered from the value.

Component attributes are similar to class definitions lexically scoped inside other class definitions. This is allowed in some object oriented programming languages (e.g. Simula) but it is not very common. In Omola a component attribute is just a class definition in the body of another class definition. The scope rules of components are similar to the scope rules of nested procedures in for example Pascal but there are some exceptions discussed further below.

Attribute declarations in a class body are terminated by a semicolon ";". If the defined class has no local attributes the class body can be empty or omitted.

Categories

The set of attributes of a class is divided into subsets called *categories*. This is a way of adding more structure to a class definition which is not found in any commonly used object-oriented language.

A category tag is a name ending with a colon ":" and it determines the category for attributes following. A class definition accepts any category tags but certain tags will be assigned specific meanings when Omola is used for model representation. This will be the issue of the next section. Attribute declarations without any category tag will by default belong to the standard category "attributes:". Here is a simple example of a class definition with different categories:

```
CACE_project IS A Research_project WITH
  leader:
    SvenErik IS A Person;
  staff:
    Bernt IS A Person;
    Dag IS A Person;
    Mats IS A Person;
  properties:
    Budget TYPE real := 1.0;
END;
```

Here, the class body defines five local attributes – four components and one variable – with three different tags. The components are all class definitions without bodies, i.e., they have no attributes except those inherited from their super-classes.

Inheritance

Whenever a new class is defined as a subclass of an existing class all attributes of that class will be inherited by the new class. An inherited attribute belongs to a class in the same way as if it was defined locally in the class. If a local attribute is defined with the same name as an inherited attribute, the local definition will override the inherited one. In this case, the local attribute must be defined in the same category as the inherited attribute or before the first category tag. For example, we can define a subclass of the CACE_project:

```
CACE_subproject IS A CACE_project WITH
```

```
  Budget := 0.5;
  staff:
    Tomas IS A Person;
END
```

which has one additional attribute in the staff category and another value for the budget variable.

The scope of names and class definitions

Classes can be defined globally or as component attributes of other classes. A global class definition is available everywhere and can be used as a super-class in any other class definition. A class must be defined before it can be used as a super-class.

Classes defined locally as attributes of another class are not available as super-classes outside the body of that class. However, a local class definition can be used in class definitions following within the same body. For example, the following definition is correct:

```
C1 IS A Class WITH
  X1 IS A Something;
  X2 IS A X1;
END
```

if "Something" is defined globally. Here, the class X2 is a subclass of the locally defined class X1. Also, in the body of subclasses of C1 both X1 and X2 are available as super-classes.

Attributes with a special syntax

So far, attributes have been defined as named objects. In some cases it is not appropriate that the user has to invent names for all attributes. For example, in the next section we will see how a class representing a model have a set of equations describing its behaviour. The equations are all individual attributes of the class but it would be inconvenient to require all equations to be declared as named attributes in the standard way. Therefore, the Omola syntax allows some attributes to be declared anonymously and with a special syntax. Connections and equations are examples of such attributes. Special tags (in this case "connections:" and "equations:") are used to direct the Omola parser to accept the special syntax of connections and equations.

The Omola syntax is designed in such a way that it is easy to extend it with additional special syntax variants.

6. The Model Representation Level

In the previous two sections we have presented the foundations of Omola as language for defining general structures of objects with attributes. In this section we will see how this language can be used for model representation.

The model representation level can be seen as an additional semantic level on top of the data level and structure level. Model representation is not inherent in the syntax of Omola. For example,

```
C IS A Class WITH
  p: x TYPE real := 0;
END
```

is correct Omola code but it has no meaningful interpretation as a model object. On the other hand,

```
C IS A Model WITH
  parameter: x TYPE real := 0;
END;
```

has a precise meaning as being the definition of the model `C` with the parameter `x`. In this case our interpretation of the code comes from the fact that we have a feeling for what a "Model" is and what a "parameter" is. In other words, there is some semantics hidden behind the words "Model" which is a class and behind the word "parameter" which is a category tag. It is important that different users and tools accessing Omola structures agree upon this semantics in order to get the same interpretation of the code.

The model representation level of Omola can be described as

- a set of predefined classes,
- for each predefined class a set of predefined attributes,
- for each predefined class a set of admissible category tags, and
- for each admissible category tag some rules concerning the attributes defined in the category.

In this section we will present the classes and their admissible category tags, of the different model objects predefined in Omola.

Models

By *models* we mean all classes which have the system defined class `Model` as a direct or indirect super-class. In a model definition the following category tags are special:

- `terminals:` All attributes defined in this category must be component attributes that are descendants of the system defined class `Terminal`.
- `parameters:` Parameter attributes are variable attributes or components that are descendants of the system defined class `Parameter`. The defined value of the parameter or the component is used as a default value.
- `realizations:` These are component attributes that are descendants of the system defined class `Realization`. When a model has more than one realization the last one has a special status as being the *primary realization* of the model. The primary realization defines the default behaviour of the model.
- `variables:` The variable attributes defined with this tag are considered to be time varying (or state) variables of the model. The defined value of such a variable is used as its default initial value.
- `constraints:` Constraint attributes are written in a special syntax (see previous section) and are used to express constraints between parameters of the model and parameters of other components.

Terminals

By *terminal* we mean all classes which have the system defined class `Terminal` as an indirect super-class. No attributes are defined for `Terminal`. Instead, `Terminal` has three system defined subclasses which can be used as model components; these are `SimpleTerminal`, `RecordTerminal` and `VectorTerminal`, and they all have different sets of predefined attributes which will be

described in this section. Terminal semantics are discussed in [Mattsson, 1988 and 1989b].

SimpleTerminal This kind of terminals are similar to variables and they can be used as variables in equations. A simple terminal can be viewed as having the following Omola definition:

```
SimpleTerminal IS A Terminal WITH
  attributes:
    value         TYPE real;
    default_value TYPE real;
    direction     TYPE (Across, In, Out);
    causality     TYPE (Read, Write);
    variability   TYPE
      (TimeVarying, Parameter, Constant) := TimeVarying;
    low_limit     TYPE real;
    high_limit    TYPE real;
    unit          TYPE string;
    quantity      TYPE Quantity;
END
```

The type specification for direction and causality are enumerations of possible symbolic values. The type `Quantity` means any object in a database of quantities.

RecordTerminal A record terminal is somewhat similar to a record in Pascal; it has a set of components which are terminals themselves. The tag "components:" should be used for the component terminals. A record terminal can not be used directly as a variable in equations. From the inside of a model dot-notation can be used to access record terminal components. From the outside of a model the the record terminal can only be accessed as a whole, and connected to other terminals with similar structure.

VectorTerminal A vector terminal has a number of identical component terminals and it can be used as a column vector variable in equations. It has a `length` attribute with an integer value, and `comptype` attribute which is the terminal class of all elements. `VectorTerminal` can be be viewed as having the following Omola definition:

```
VectorTerminal IS A Terminal WITH
  length TYPE integer;
  comptype IS A Terminal;
END
```

Parameters

Parameters are components of models and realizations defined as subclasses of the predefined class `Parameter`. A parameter can be used as a time constant equation variable and it has one predefined variable attribute: `value`.

Realizations

Realizations are used to define model behaviour. A realization is always defined as a component of a model and it refers to parameters and terminals of that model. It has no meaning outside the model in which it is defined.

There are two important subclasses of the predefined class `Realization`; these are `Primitive` and `Structure`.

Primitive This kind of realization defines model behaviour as a set of differential algebraic equations. The following tags have a special meaning:

- `variables`: with the same meaning as in models,
- `constants`: which are variable attributes with defined values,
- `parameters`: with the same meaning as in models, and
- `equations`: which are given with a special syntax discussed above.

The equations of a primitive realization can refer to variables, constants and parameters of the realization as well as variables, parameters, terminals and terminal components of the model of which the realization is a component.

Structure A structure realization defines model behaviour as a set of submodels and connections. The following category tags are accepted:

- `submodels`: where the attributes should be models,
- `connections`: where the attributes should be connections written in special syntax, and
- `constraints`: where the attributes should be constraints written in special syntax.

7. An Environment for CACE

Omola is designed to be a language used for model representation in an integrated environment for model development, simulation, control design (CACE), etc. Omola code can be turned into data structures representing models which can then be displayed graphically and manipulated iteractively. Models that are created from scratch within the environment can be turned into Omola code and saved. Omola can also be used for documenting models and as a standardized form for exchanging models between users and different systems.

8. Interpretation of Omola objects

Since Omola is a declarative language Omola code can be interpreted in different ways depending on the circumstances. Many other simulation languages are more imperative to its nature, giving explicit expressions how variables should be computed. These languages are less flexible and more devoted to a special purpose, for example simulation.

Omola class definitions can be interpreted in different ways by different tools in the CACE environment. Normally in object-oriented programming, classes serve as descriptions of run-time objects. The creation of an object from a class is called *instantiation*. In a CACE environment, 'run-time' corresponds to the simulation of the model. Before a model can be simulated it has to be instantiated. This means that separate instances are created for each class definition and class references in a model definition. Instantiation is a recursive procedure over a structure of class definitions.

Instantiation for the purpose of simulation is only one way to interpret a set of class definitions in Omola. Here are some other purposes and interpretations:

- Generating a graphical picture of the system structure, for example, a block diagram.
- Generating text descriptions of the system for documentation or user information.
- Generating special purpose code, for example, regulator code or simulation code in other simulation languages.
- Generating standardized system descriptions in order to communicate with other control engineering packages.
- Derivation of different kinds of systems properties like stability margins, loop gains, etc.
- Transforming a model into a form accepted by a specific control design tool.

9. Extending Omola

Omola is designed to be extendable. New model structuring concepts and new aspects of models can normally be expressed within the current framework of concepts on the data and structure level. However, there are some limitations of Omola as compared with some object-oriented programming languages. Some of these might be useful to have in a modelling language as well.

Multiple inheritance

Multiple inheritance is possible in some object-oriented languages and sometimes considered very useful. Currently, it is not possible for an Omola class to inherit properties from more than one super-class. However, we believe it is useful also in model representation and ought to be included in the language.

As an example where multiple inheritance is useful, consider a model of a chemical reactor. A reactor tank has its physical properties in common with all tanks. These properties can be inherited from a previously defined standard tank model. Beside the properties of the physical device, we also have a model of the chemical reaction going on in the tank. This reaction can be described independently of the vessel in which it takes place. Therefore, we would like to define a class which inherit properties from both the class defining the tank model and from the class defining the chemical reaction model.

There are no principle problems by including multiple inheritance in Omola but some difficulties are to be solved.

Procedural specifications of model behaviour

As presented, Omola is a purely declarative language. Since methods can not be defined, there is no concept of procedural or functional knowledge. If methods are to be included, we have to decide in what language they should be written. In a "quick and dirty" Lisp (or KEE) implementation of a model representation environment based on Omola, the simple solution is to use Lisp to implement the methods. If tools like Matlab and Macsyma are integrated into the environment, it should be possible to implement and execute methods in these languages.

If procedures can be associated with models, another interesting possibility appears. Models can be supplied with knowledge about how to use the

different tools in the environment for analyzing and manipulating the models themselves. For example, a non-linear model realization can have method for linearization associated with it. When all parameters of a model are defined and the operating point is settled, possibly by simulation, the user can send a message to the model asking for its linearization.

Discrete events

Many phenomena in real world systems are most conveniently modeled as discrete events rather than continuous time processes. Typical examples are models of batch processes, production lines etc. Examples of events are a tank that becomes full and causes a valve to close, and a processing machine that becomes ready for the next piece of material. Often unmodeled dynamics considered to be fast compared with the rest of the system can be represented by events. For example, a relay with hysteresis is conveniently described in a language containing the event concept. Sampling is also conveniently represented as events.

Concepts for describing discrete event behaviour can be added to Omola without too much effort.

10. Acknowledgements

The author would like to thank Sven Erik Mattsson, Bernt Nilsson, Dag Brück, and Tomas Schönthal, for interesting discussions and useful ideas. This work has been supported by the National Swedish Board for Technical Development (STU) under contract 87-2503.

11. References

ANDERSSON, M, (1989): "An Object-Oriented Modelling Environment," *Proc. of the 1989 European Simulation Multiconference, Rome, June 7-9.*

ÅSTRÖM, K. J. and W. KREUTZER (1986): "System representations," *Proc. IEEE Control Systems Society Third Symposium on Computer-Aided Control Systems Design (CACSD).*

BIRTWISTLE, G. M., O-J. DAHL, B. MYHRHAUG and K. NYGAARD (1973): *Simula Begin*, Auerbach, Philadelphia, Pa.

ELMQVIST, H. (1978): *A Structured Model Language for Large Continuous Systems.*

MATTSSON, S. E. (1988): "On Model Structuring Concepts," *Preprints of the 4th IFAC Symposium on Computer-Aided Design in Control Systems (CADCS)*, August 23–25 1988, P.R. China, pp. 269–274.

MATTSSON, S. E. (1989a): "On Modelling and Differential/Algebraic Systems," *Simulation*, 52, No. 1, 24–32.

MATTSSON, S. E. (1989b): "Modelling of Interactions between Submodels," *Proc. of the 1989 European Simulation Multiconference, Rome, June 7-9.*

MEYER, B. (1988): *Object-oriented Software Construction*, Prentice Hall.

MOLER, C., J. LITTLE and S. BANGERT (1987): *PRO-MATLAB User's Guide*, The MathWorks, Inc., Sherborn, MA.

RIMVALL, C. M. (1986): *Man-Machine Interfaces and Implementational Issues in Computer-Aided Control System Design*, Dissertation, Swiss Federal Institute of Technology Zurich.

STEFIK, M. and D. G. BOBROW (1986): "Object-Oriented Programming: Themes and variations," *AI Magazine*, **6:4**, pp. 40–62.

Appendix A. – Omola Syntax

The formal syntax of Omola is presented in this appendix. The syntax is presented as a context free grammar on BNF form. Non-terminal symbols are written in upper case letters while terminal symbols are written in lower case letters. Key words and special characters that appears litterally are enclosed in dubble quotes. A vertical bar "|" separates alternatives. An expression within square brackets is optional, i.e., [exp] is the same as (exp | $), where $ is the empty string.

The syntax rules are divided into three groups: basic Omola syntax, special syntax for model representation and syntax for mathematical expressions.

Basic Omola syntax

The rules defining the basic omola syntax is here listed in alphabetic order. The non-terminal symbol "CLASS-DEFINITIONS" is the start symbol, i.e., it represents a complete Omola block of code.

```
BODY ->
  NAME-LIST (CLASS-DEF | TYPE-DECLARATION | C-ASSIGNMENT)
  ";" [BODY]

C-ASSIGNMENT -> ":=" LITTERAL

CLASS-BODY -> (BODY [TAG-BODY] | TAG-BODY)

CLASS-DEF ->
  ("is a" | "is an") SUPER-CLASS ["with" CLASS-BODY "end"]

CLASS-DEFINITIONS ->
  NAME CLASS-DEF ";" [CLASS-DEFINITIONS]

LITTERAL ->
  real-number | integer | MATRIX | POLYNOMIAL

NAME -> identifier

NAME-LIST -> NAME ["," NAME-LIST]

SUPER-CLASS -> identifier

TAG ->
  "terminals:" | "parameters:" | "attributes:" | ...

TAG-BODY ->
  (TAG BODY [TAG-BODY] | SPECIAL-TAG-BODY [TAG-BODY])

TYPE-DECLARATION ->
  "type" TYPE-DESIGNATOR [C-ASSIGNMENT]

TYPE-DESIGNATOR ->
  ("matrix" integer integer |
```

```
    "row" integer |
    "column" integer |
    "scalar" |
    "real" |
    "integer" |
    "string" |
    "symbol" |
    "polynomial"
    "(" NAME-LIST ")" |
    NAME )
```

Special syntax

Here is the syntax for a number of Omola constructs specially introduced for model representation.

```
SPECIAL-TAG-BODY ->
  ("constants:" | "constant:") CONSTANT-BODY

SPECIAL-TAG-BODY ->
  ("variables:" | "variable:") VARIABLE-BODY

SPECIAL-TAG-BODY ->
  ("equations:" | "equation:") EQUATION-BODY

SPECIAL-TAG-BODY ->
  ("connections:" | "connection:") CONNECTION-BODY

SPECIAL-TAG-BODY ->
  ("constraints:" | "constraint:") CONSTRAINT-BODY

CONSTANT-BODY ->
  NAME-LIST C-ASSIGNMENT ";" [CONSTANT-BODY]

VARIABLE-BODY ->
  NAME-LIST (TYPE-DECLARATION | C-ASSIGNMENT)
  ";" [VARIABLE-BODY]

EQUATION-BODY ->
  EQUATION ";" [EQUATION-BODY]

CONNECTION-BODY ->
  CONNECTION ";" [CONNECTION-BODY]

CONSTRAINT-BODY ->
  SIMPLE-EXPRESSION ":-" CONSTRAINT-EXPRESSION
  ";" [CONSTRAINT-BODY]

CONSTRAINT-EXPRESSION ->
  SIMPLE-EXPRESSION [":-" CONSTRAINT-EXPRESSION]

EQUATION ->
```

```
  CONDITIONAL-EXPRESSION "=" CONDITIONAL-EXPRESSION
```

Syntax for arithmetic expressions

```
C-POLY -> ELEMENTS

CONDITIONAL-EXPRESSION -> EXPRESSION

CONDITIONAL-EXPRESSION ->
  "if" EXPRESSION "then" EXPRESSION
  "else" CONDITIONAL-EXPRESSION

ELEMENTS -> CONDITIONAL-EXPRESSION ["," ELEMENTS]

EXPRESSION -> SIMPLE-EXPRESSION [REL-OP SIMPLE-EXPRESSION]

FACTOR -> ["not"] FACTOR | PRIMARY ["^" FACTOR]

FUNCTION-DESIGNATOR ->
  identifier "(" [ELEMENTS] ")"

MATRIX -> "[" ROWS "]"

NUMBER -> real-number | integer

POLYNOMIAL -> "{" (C-POLY | R-POLY) "}"

PRIMARY ->
  VARIABLE | MATRIX | POLYNOMIAL |  NUMBER |
  "(" EXPRESSION ")" | FUNCTION-DESIGNATOR

R-POLY -> ARITHMETIC-EXPRESSION ":" ELEMENTS

REL-OP ->
 (">" | "<" | "==" | "~=" | "<=" | ">=")

ROWS -> ELEMENTS [";" ROWS]

SIMPLE-EXPRESSION ->
  [("+" | "-")] TERM [("+" | "-" | "or") SIMPLE-EXPRESSION]

TERM ->
  FACTOR [("*" | "/" | "and" ) FACTOR]

VARIABLE -> identifier
```

Appendix B. – Data Types

Here is a listing of all data types currently available in Omola:

Type	Declaration	Example litteral	Comment
Constraint	`constraint`	`p1 :- p2`	
Equation	`equation`	`dot(x) = x-1`	
Integer	`integer`	1	
Matrix	`matrix` *mn*	[1, 0; 0, 1]	
	`row` *m*	[1, 0]	
	`column` *n*	[1; 0]	
Polynomial	`polynomial`	`{1,2,3}` or `{2: 1,1}`	
Real	`real`	3.14	
Scalar	`scalar`	3.14	Identical with real.
String	`string`	`"m/s"`	
Symbol	`symbol`	`Tank`	
	`(A,B,C)`	`A`	The declaration can be any list of symbols.
	a class name	`Voltage`	Reference to an object.