Representation and Visualization of Systems and Their Behaviour

Mattsson, Sven Erik

1987

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

CODEN: LUTFD2/(TFRT-3194)/1-033/(1987)

Representation and Visualization of Systems and Their Behaviour

Sven Erik Mattsson

STU project 86-4049

Department of Automatic Control
Lund Institute of Technology
September 1987

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

Document name
Report

Date of issue
1987-09-30

Document Number
CODEN:LUTFD2/(TFRT-3194)/1-033/(1987)

Author(s)
Sven Erik Mattsson

Supervisor

Sponsoring organisation
The National Swedish Board of Technical Development (STU contract 86-4049)

Title and subtitle
Representation and Visualization of Systems and Their Behaviour

Abstract

This report describes the work done in the project "Representation and visualization of systems and their behaviour" (STU project 86-4049). It has been a part of the project "Computer Aided Control Engineering (CACE)" at the Department of Automatic Control, Lund Institute of Technology.

The purpose of this work has been to study various aspects of system representation. The representation of systems is an important and critical part of a CACE system, since it should be common to make the various tools integrated. It is also important to separate the user interface from the processing parts to get flexible tools allowing adaptation of the user interface to the needs and the desires of various users. We believe that it is possible to agree upon a common set of modelling concepts, but that it would be useful to allow the concepts to have different textual and graphical representations. Basic modelling concepts are outlined and five complementary modelling structuring principles are proposed.

We have also focussed on the use of graphics to visualize systems and their behaviour. First, experiences of using hierarchical block diagrams and Hibliz are discussed. Second, a prototype window-based environment for Simnon on the Sun workstation has been developed. systems. The prototype is usable and can thus give us user feedback. Third, a program for real-time animation in 3-D graphics of ASEA's Industrial Robot IRB 6/2 has been developed. The animation feels very reel. Fourth, graphics and window standards are discussed.

The work has been influential in shaping the future of the CACE project, which will be focussed on development of prototype tools for model development and simulation.

Key words
Computer Aided Control Engineering; Computer Aided Control System Design; Software; System Representations; Computer Graphics

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
33

Recipient's notes

Security classification

Contents

1. Introduction

This report describes the work done in the project "Representation and visualization of systems and their behaviour" (STU project 86-4049). It has been a part of the project "Computer Aided Control Engineering (CACE)" at the Department of Automatic Control, Lund Institute of Technology.

This introductory chapter is organized as follows. First, we briefly motivate why representation of systems is an important issue in CACE. Second, we consider how some basic requirements on CACE systems influence the design of a system concept. Third, the organization of the report is described.

1.1 Motivation

The notion of system is an essential element of control theory. The system concept is used in all steps from specification to implementation of control systems. Consequently, the representation and visualization of systems and their behaviour are from several aspects a key issue in CACE.

It is fruitful to view a CACE system as a high level problem solving language. A language should be more than just a means for instructing the computer to perform tasks. It should also serve as a framework within which we organize our ideas about the application. It should support the user's mental model of the problem. Consequently, the designer of a user interface must know the user. This is a basic and important reason why control engineers should take part in the design of CACE programs.

In the subproject "High-level problem solving languages for computer aided control engineering" (STU contract 85-4808) (Åström and Mattsson, 1987) existing CACE packages were viewed as high level problem solving languages. Examples of system representations used in current CACSD packages are given in the report. The study showed that system representations are poorly dealt with in existing CACE systems.

Most of today's CACE programs are dealing with pure mathematical descriptions. However, there is also a need for being able to discuss qualitatively about systems and for handling of incomplete and maybe inconsistent information. Ideas and concepts from artificial intelligence and knowledge engineering may be useful in this context.

Existing CACE packages were designed for computers, which by today's standards have moderate computing power and primitive devices for input and output. Graphics plays an important role in control engineering. First, many of the techniques for the analysis of control systems rely on graphics; trend curves, phase planes, Bode plots, Nyquist plots and root locus plots. Second, it is natural for a control engineer to use block diagrams when describing a process or a model. Unfortunately, the use of graphics in CACE has been hampered by lack of feasible hardware for a long time. However, the situation is changing drastically. The new workstations with high performance, real-time graphics offer new possibilities to introduce graphics in CACE.

1.2 Basic Requirements

We will in this section discuss how some basic requirements on CACE software influence the design of a system concept.

Integrated toolkit, not encapsulated packages

It is today commonly agreed that it is impossible to make the ultimate CACE package. First, new methods for analysis and design are developed. Second, different users may have conflicting demands. Third, such a package would be of incredible size and impossible to manage and maintain. A CACE system should not be an encapsulated package. The CACE software should be flexible and modularized with well-defined interfaces between the modules. The buzzwords are tool, toolkit (tool box) and tool machine. The last one indicates that it is important that a CACE system is not merely a collection of tools, but an integrated system. By integration we mean that there need not be any explicit context switches or conversions when switching between tools and that a user should have a uniform, consistent and predictable interface to all tools.

Separate the user interface from processing parts

To get a flexible tool box it is important that the CACE software is properly modularized. In rough outline a CACE system can be decomposed into three parts: the user interface, data and processing tools.

A user interface could be designed in many different ways using various kinds of input and output devices. The programmer should be able to include new tools that support new I/O devices and new ways to interact. Since various users have different preferences and need different kinds of support, it would be attractive if the user could design his favourite interface. The CACE system must of course provide a number of typical interfaces, especially for beginners and casual users.

The feasibility of separating the user interface from the processing parts is supported by the observation that it is useful to view a CACE system as a high level problem solving language. The design of a CACE system can then be split into two parts: first, the language and the user environment or in other words the user interface and second, the interpreter with the processing parts. A CACE system can be considered as executing a read-evaluate-print loop. In the read- and print-phases the user interface is active. A user interface can be viewed as consisting of a language and an environment. The language should support concepts that are powerful and natural for the user. For languages there are concepts such as syntax and semantics. We may say that syntax defines allowable forms and semantics has to do with the meaning and interpretation. In an interactive environment where menus and graphics are used for input, the borderline between the language and the environment is less distinct. The user can just as well view the syntax as a part of the environment.

The user interface can be made flexible by allowing various forms of syntax: text input from keyboard, menus, graphics etc. Various help and expert system interfaces could be included on this level. The user interface should collect the user's input and then convert it into a textual representation. When this is done the proper processing tool should be invoked. When the user interface has got the result, it should be presented to the user. The textual representation

of a command is useful for the documentation and when programming new composite commands.

The discussion above indicates that the representation of systems is an important and critical part of a CACE system, since it should be made common to all tools. We believe that it is possible to agree upon a common set of modelling concepts, but that it would be useful to allow the concepts to have different textual and graphical representations.

Standardization efforts

The importance of having a common, standardized way of describing systems is now widely recognized in the control community. As an outcome of a discussion session on standardization of software for CACSD at the 3rd IFAC/IFIP International Symposium on Computer Aided Design in Control and Engineering Systems, CADCE '85, Lyngby, Denmark, July 31 – August 2, 1985 an IFAC Working Group on "Guidelines for CACSD Software" was formed. It is chaired by prof. Magnus Rimvall, ETH, Zürich. Three subgroups has also been formed: algorithms (Valk, Delft, the Netherlands), data structures (Maciejowski, Cambridge University, UK) and user interface (Rimvall). Members of the CACE project are participating in this work.

The subgroup on data structures are currently concerned with format and structure of external files for transfer of data between CACSD software. There is a proposal for state space models (linear, time invariant), transfer function matrices, matrix fraction descriptions and pole-zero-gain definitions. The standardization of more complex models has not been addressed yet. The subgroup on user interfaces is also considering system representation but from another viewpoint. It is a well-known fact that the matrix language MATLAB has inspired the developers of CACSD packages for analysis and design of linear systems. CTRL-C, MATRIX$_X$, PC-MATLAB and PRO-MATLAB are three commercial examples, and Blaise, IMPACT and EAGLES/M are three examples from university or non-commercial research institutes. The impact of MATLAB has implied that the syntax of expressions etc. is very similar. However, there are minor, but annoying differences which makes it impossible to use the same system descriptions or command procedures. This situation indicate that it would be possible to come up with a standard in this area. Rimvall has submitted a draft proposal.

Declarative models

The desire to have models that could be used in various contexts implies that declarative forms (equations) should be used to describe the behaviour of submodels and the interactions between models. Declarative models on symbolic forms can be used in various contexts, since they can be manipulated automatically to generate efficient code for simulation, code for calculation of stationary points, linear representations, efficient control code, descriptions which are accepted by other existing packages etc. Furthermore, it is a global problem to make a procedural description or make the algorithm from which the behaviour can be calculated. It cannot be made for the individual submodels independently of how they are used.

1.3 Outline of the Report

The work of the project described in this report has been focused in a number of directions. In Chapter 2 the focus is put on semantical issues whereas syntactical issues is discussed in Chapter 3. The purpose is as indicated above to separate the design of a system concept and its semantics from the design of its syntax and representation. It this way we can hopefully obtain a common system concept so that the tools could be made integrated, while at the same time allowing different users to have different textual and graphical representations. Chapters 2 and 3 start with an outline of their content. Conclusions are presented in Chapter 4.

2. Model Structuring Concepts

From practice we know that models for real plants often become complex, large and difficult to handle and understand. In most cases it is also difficult to reuse a model or a part of a model in another application. The objective of this chapter is to discuss how we by imposing structure on models and collections of models can remedy these complexity problems.

This chapter is organized as follows. In Section 2.1 we identify five complementary model structuring principles that are useful to handle model complexity. In Sections 2.2–2.4 these principles are discussed in more detail to find out what concepts are needed to support them. The approach is to first indicate concepts that in some way may be useful without bothering too much about whether our desires may be contradictory. To begin with, the focus is put on semantical issues while representational and syntactical issues are disregarded. When possibilities to represent a concept are indicated, the purpose is to make the discussion somewhat more concrete or to indicate that it is possible to come up with a useful representation.

It is our aim to separate the design of the structuring concepts and their semantics from the design of their syntax and representation. We believe that various users could agree upon a common set of concepts, but that it would be useful to allow the concepts to have various textual and graphical representations. This approach is also consistent with the desire to separate the user interface, database and the processing parts from each other.

2.1 Model Structuring Principles

Abstraction and modularization are two basic principles that can be used to impose structure. The essence of abstraction is to extract important properties while omitting insignificant details. By introducing different views the attention can be focused on various properties of the model. By introducing levels of abstraction the model can be viewed in more or less detail, allowing the model to be viewed with gradually increasing detail. The amount of information increases at lower abstraction levels. Modularization can be used in order to maintain useful views with a limited number of related concepts. Modularization means that the information at a certain abstraction level is decomposed into smaller entities.

We will in this section point out five complementary model structuring principles or classes of abstraction views, namely:

1. Hierarchical submodel decomposition
2. Model types
3. Model categories
4. Multiple versions
5. Multiple presentations

which are natural and useful for handling model complexity.

Hierarchical submodel decomposition

Concepts allowing a hierarchical decomposition of a model into submodels are useful. Modularization simplifies the modelling in several ways. First, the model developer can focus on a smaller part of the system at a time. Second, libraries of submodels can be built. This means that models can be reused. Technical systems are often built in a modular way composed of standard components. Their behaviour may be well-known. Good, generally accepted models may already exist. Third, modularization facilitates testing. It is difficult to verify that a simulation program implements the intended mathematical model. With a modular approach we split the problem into smaller parts. Fourth, the model becomes more flexible and easier to adapt.

Model types

In a complex plant, a component may appear several times. For example a chemical plant may have several identical PID controllers. The maintenance of a model for this plant is facilitated if the models for all those PID controllers share a common description. Common descriptions can be supported by introduction of a model type concept.

Model categories

To make a model library useful, structuring mechanisms which allow classification of the models into various categories should be provided. Without structuring mechanisms the libraries would be messy and it would be difficult for a user to find the models that might be of interest for him. It should be possible to reference a library and automatically have a menu of its models. If the category name is used when referencing a model the name space of model names are split up and the risk of name conflicts is decreased.

Multiple versions

Modularization by itself supports flexibility, but there is also a need for other model structuring concepts to support flexibility. Two conceptually different needs of adaptability can be identified: adaptability with respect to different plant designs and adaptability with respect to model complexity.

When developing a new model it is convenient to start with a simple model and to test it, and then refine it stepwise by adding new features, while retaining the old versions for comparison. Furthermore, it is impossible to make a model which can simulate all aspects of a given plant. Models of different complexity must be used for simulation of different events. If all these models have the same interfaces to the environment, it is convenient to view them as different versions of the same model. The user can then easily adapt the model by selecting an appropriate version for example by using the mouse and an automatically generated menu. One can also visualize a system which can select the particular version automatically.

We also need structuring concepts to make the model flexible with respect to different process designs. During the design of a system the model has to be updated as the design proceeds. Questions of the type "What happens if we modify the design in this way?" arise frequently in simulation projects. For example, if we simulate a wind power plant it should be simple to first simulate the system when it has a synchronous generator and then when it has an asynchronous generator. This can be achieved if we define a generator

model with properly defined interfaces, and if the simulator offers a facility to exchange the content of the generator model easily.

Multiple presentations

The model developers and users may want different views of a model and their needs may change with time. As an example consider a user who selects models from a library. The user then first wants a list of contents (an index) for quick scanning. If he finds a model of interest, he wants basic information about the model like:

1. Purpose of the model
2. Assumptions done
3. Range of validity
4. References

Assume that the user finds a model that seems to fulfil his requirements. He then wants to know how it is used. Two types of information are of interest. First, how can the model be controlled; what are the parameters? Second, which variables can be inspected, plotted, stored etc.? The information about parameters and variables should be easily available on the user's request. Note that he is not at any of these stages interested to see the equations themselves. This is too low a level.

2.2 Hierarchical Submodel Decomposition

As motivated above it is important that a CACE system has facilities which allow a hierarchical decomposition of a model into submodels. We view hierarchical submodel decomposition as the conceptually most important concept to handle model complexity. It supports our way of thinking about systems. The other four concepts mentioned above are of course important and useful, but they are more technical to their nature. They mainly support bookkeeping and maintenance of models. We will now discuss what we should mean by a submodel and how the concept should be formalized.

Which submodel concept should we have?

The selection of submodel boundaries should be guided by our own perception of the problem space. Our main intended application is modelling and simulation of technical systems. It is thus natural to consider our perception of these systems. Technical systems are often designed and built in a modular way. Whether we consider large plants or smaller systems, it is natural to talk about components and subsystems. A person with some technical knowledge about a system views it as consisting of a number of components. There exists a terminology. Consequently, it is of interest to support a hierarchical submodel concept that could map the component structure of real systems. If a hierarchical component concept is supported, it is possible to build model libraries. Models for more complex systems could be built in a similar way as the system itself by composing existing submodels. There are more good reasons for having an easily identifiable model for each component of a system. First, it would give a very good technical documentation of the components. Second, as indicated in Section 2.1 different users want different views of a model. When the submodels corresponds to components, various

stylized pictures and component symbols could be provided as representations for the parts. This implies that various graphical representations like circuit diagrams, mimic diagrams or stylized cartoons could be created with reasonable effort. Third, it would be more natural for manufacturers of technical components to supply a useful model instead of awkward data sheets. We believe that purchasers will in the future demand that suppliers of technical components provide dynamical models for their components. Consequently, standardization is needed, as well as procedures and organizations for validation and certification. Fourth, it would be easier to reconfigure the model and thereby easier to investigate the effect of new or or alternative components.

Encapsulation

A difficult part when designing a modularization concept is how interaction between modules should be defined and treated.

Today most computer scientists favor strongly encapsulated modules. The package concept of Ada and the popularity of object-oriented programming are two manifestations of this fact. The basic reason for having encapsulated modules is that it supports abstraction. A user need not bother about implementation details and the developer is free to modify the implementation as long as he preserves the properties of the module as defined by the interface. Encapsulation also allows the system to check that a module is used properly.

When discussing encapsulation we must consider nesting of model definitions. We do not see any good reasons to allow nested model definitions. The name space of model names could as indicated in Section 2.1 be split up using a category concept. Non-nested definitions support the building of component libraries implying that it will be easier to reuse models and parts of models. The basic scope rule should be that the environment should be invisible from inside a submodel. A submodel should basically only be able to see its own internal variables and its interface variables. However, there are good reasons to have a few exceptions from this rule. Time is such example. Every submodel should be able to read the universal time.

Let us now consider what should be visible and manipulatable from outside a submodel. Our starting point is that a submodel is a encapsulated unit, i.e., that nothing is implicitly available from the outside. It must be possible to model the physical interaction between components of a plant. It is also of interest that the models to some extent are adaptable so they can be used in a range of applications without having to be edited. For example, it ought to be possible to make a model that could describe pipes having various length and diameter. A simple but rather powerful approach to introduce parameters that could be set from outside a model. We will return to these issues after having considered modelling of interaction between components.

Modelling of interaction between components

Interaction between two components can in many cases be described using an idealized model of a real "physical" connection. Typical examples are shafts, pipes and electrical wires. The "ideal" shaft has no mass, is rigid and cannot break. Its attributes are position, speed and torque. The "ideal" pipe has no losses, no transportation time (infinitely short length) and do not mix the material. Its attributes are flow rate, pressure, temperature, enthalpy, various concentrations etc. The "ideal" electrical wire has no resistance, no delay time and does not distort wave forms. Its attributes are voltage and current.

To be able to describe relations between attributes of submodels, we must be able to access the attributes from the outside. We will do that by letting the submodels have variables called terminal variables, which are accessible from the outside. The physics of the idealized interactions discussed above indicates that it is useful to introduce two types of terminal variables. One type is those terminals where a connection implies that connected terminal variables should be equal. Variables obeying this kind of connection semantics are sometimes called "across variables" in the literature (Koenig, Tokad and Kesavan, 1967). Examples of across variables are voltage, pressure and temperature. The other type is called through variables and has an associated direction (in or out) and the connection semantics implies that connected variables should sum to zero. Examples of through variables are current, mass and energy flow, thrust and torque. If a real physical connection cannot be modelled in this idealized way, the natural solution is to introduce a new submodel to describe the more complex relation.

The concept of using simple relations to describe interaction between submodels seems to be very natural. It was proposed by Elmqvist (1978). However, most of today's languages for continuous simulation do not support this concept. Instead the models can have inputs and outputs. In the simplest case interaction is modelled by connecting an output to each input. This means that the value of an input is defined by an assignment statement saying that the value should be set to the value of an output. This can be generalized somewhat by allowing the right part of the assignment statement to be an expression including various outputs and parameters. However, as thoroughly motivated in, for example, Elmqvist (1978) and Elmqvist and Mattsson (1986), the need to define for each submodel which of its variables are inputs (in other words are known) and which of its variables are outputs (defined by the model) puts constraints on the possibilities to build model libraries. What is input and output of a submodel from a computational viewpoint depends on how the submodel is connected to other submodels and cannot be decided locally inside a submodel. The conclusion is that in order to support model libraries, the submodels should have explicitly defined terminals which could be referenced on the parent level, and the concepts for describing interactions should be based on equations and not assignment statements.

Terminals

Structuring concepts for terminal variables are needed, since, as indicated above, an interaction between two submodels often involves several attributes. We must be able to aggregate terminal variables. With a hierarchical submodel concept it is natural to have a hierarchical aggregation concept for terminal variables. The word "terminal" will in the following denote such an aggregate.

A connection between two terminals should of course imply that the components of the terminals are connected, but how should the components be matched? There are two major approaches. First, the position could be used; the first components are considered to be connected and so on. Second, the names or other significant information about the components could be used. These two approaches and probably all other have the drawback that the user may have to make additional definitions or is forced to introduce new simple subsystems that works as adapters when he wants to connect two subsystems that have not been designed with explicit purpose of being connected to each other. One way to avoid this problem at least within certain application do-

mains or companies, is to explore the idea that terminals are idealized models of real physical connections and build libraries for terminals. Such a library could also contain conversion rules or adapters.

A basic task is to make the use of library models user friendly, safe and reliable. The encapsulation of submodels prevents to a large extent unintended abuse. However, the terminals are delicate holes in the wall. It would be nice if the user could get automatic warnings when making improper connections. To be able to issue automatic warnings, the system must be given rules for what can be accepted. Connections between across and through variables should not be accepted, and boolean and numerical terminal variables should not be connected. To be able to provide more elaborate checks, the model developer must supply more information. One approach is to associate a type to each terminal and only accept connections between terminals of the same type. This demands a common agreement on the names of the types. To make it possible to use models developed at various places without to much work, it ought to be possible to define conversion rules or adapters in a simple way. Furthermore, to make it possible to use generic models (e.g a first order system) there must be a universal type. A second approach is to give attributes like units and other characteristics, for example, `water temperature [K]`. This approach also requires agreements, now in the form of what kind of information that should be supplied and how.

Let us consider the pros and cons of the type based approach and the attribute oriented approach a bit further. The type based approach is probably simpler to implement. The attribute oriented approach requires rules and pattern matching. The type based approach gives very good opportunities to detect erroneous connections, if the model developers have used the type concept properly and introduced sufficiently many types. However, the is a risk that the model developers associate the same type or the universal type to all terminal variables. The attribute oriented approach might be easier to use. The user does not have to decide at the start of a model development project which types of terminal variables should exist. Attributes can be added incrementally. The model becomes more readable. The attributes of the terminal are explicit and not encoded into a type. To sum up, for us it is an open question whether a type based or attribute oriented approach, or maybe something else, should be recommended. The idea mentioned above to view a terminal as an idealized physical components indicates a middle course. The various types could then correspond to different idealized components as electrical wires, pipes, shafts etc or combinations of those. By allowing parametrization of the terminals additional safety checks could be provided. For example, a terminal corresponding to a pipe could have a parameter indicating the pipe diameter and it could then be checked automatically that two connected pipes have the same diameter.

A way of preventing unintended abuse of a model during simulation is to allow the model developer to define ranges for the terminal variables and issue warnings when some terminal variable gets outside its allowed range.

Measurements

A drawback with strong encapsulation is that the model developer must anticipate all reasonable ways that the submodel can interact with its environment. If he has forgotten to include variables that you think are useful to describe an interaction, you have to modify and edit the model. For technical systems

it is advisable, and a reasonable task for the model developer, to include interface variables to that extent that those ways of interaction anticipated by the designer of the real component can be modelled.

It may be difficult to anticipate hooking on of measurement devices. Consequently, it may be reasonable to allow exceptions from the strong encapsulation rules when the user wants to model an ideal measurement in order to be able to rapidly check the benefits of a new measurements. An ideal measurement does not directly influence the dynamics of the submodel, since it is just a reading of a value.

Time

When solving an ODE, the independent variable has a special status and it ought to be accessible from all submodels. For obvious reasons we usually refer to it as time. A nice and natural way of making the actual time accessible in a submodel is to let the user define a local name for the time, as done in ACSL "`Variable t`" or as in Simnon "`Time t`" where `Variable` and `Time` are special reserved keywords. The user is here free to select any convenient name.

Adaptability – parameters

To make a model library useful, it is important that the models could be designed to cover a wide range of applications. Adaptable model types would also mean that the number of model types needed in an application domain could be reduced. However, the concepts for supporting adaptability of model types must be designed carefully so the use of library models does not become unnecessary laborious.

The extreme way of adapting a model to a certain application is of course to edit its type definition. The user then has to decide if he wants to modify the existing definition or if he wants to create a new model type using the old one as a starting point. As indicated in Section 2.1, it would be useful to have a version concept so versions of a model could be collected into one model type. This allows editing of a model type without destroying the existing one. Thus a model version concept supports flexibility. It may be argued that it is clumsy to use. However, when the versions are available, the user just has to select. All alternatives need not be developed at once. They could be developed when needed. If also a default alternative is provided, the casual user may even be unaware about the flexibility and no extra burden is put on him. An example where this technique may be useful is when a component may be implemented in several structurally different ways. For example, we can have a general model for electrical generators where the synchronous type and the asynchronous type are two major alternatives.

Unfortunately, it is in many cases not sufficient to have just a small number of alternatives. Flexibility can then be supported by leaving suitable properties undefined and assuming that they are provided via terminals of the model. This is in many cases an unnecessarily clumsy and laborious approach. As an example consider a model for an ideal resistor. A natural demand is that the user should be able to set the resistance in a simple way. This kind of flexibility can be supported by introduction of parameters having default values.

By a parameter we mean basically a variable which value is fixed during a simulation but can be changed in a simple way between two simulations. A model of a straight pipe may for example have length, width and surface

roughness of the pipe as parameters. If the user can change the value of a parameter by issuing a simple command, he can easily try out a number of process designs that do not involve structurally modifications of the plant. Parameterization also means that the number of model types can be kept lower. For example, just one resistor model type is needed.

The value of a parameter cannot be a property of a model type, since it is natural demand that a model type could be used at several places in a model simultaneously. For example, an electrical circuit contains typically several resistors having different resistance as well as a chemical plant has several pipes of various length. However, it is reasonable that the default value of a parameter is a property of the model type.

A hierarchically decomposed model will typically have a lot of parameters. Consequently, it is important to consider the handling of parameters carefully. When a number of components are put together to form a new and more complex component, the components usually must fulfil some constraints to fit together and to meet the specifications. For the model this implies that parameters should be related to each other, and that some parameters should become fixed and constant. For example a simple connection of two pipes may demand that the shapes and sizes of the two cross sections are equal. This indicates that the model developer should be able to define a new set of parameters which defines the values of parameters at lower levels. Or in other words, he should be able to define old parameters in terms of new ones. In this way existing parameters could be given new default values. Furthermore, the model developer should be able to fix the value of an old parameter and tell the simulator that it should be considered as a constant. This approach also supports the introduction of performance related and functionally related parameters at higher levels. A more pragmatic reason is that the model developer may want to assume certain relations between the parameters of different submodels to make the new model less complex. For example a model may assume that a number of components of the same type have identical parameter values.

In many cases a simple numerical or boolean variable will do as a parameter. However, there are motives for having more complex parameters. The characteristics of a linear spring can be described by a spring coefficient. If the spring characteristics is nonlinear the situation becomes more complex. One way to solve the problem partially is to parameterize the nonlinearity. However, we then cannot handle general spring characteristics. Only certain classes anticipated by the model developer are available. The need to describe nonlinear characteristics in models of technical systems is quite common. For example, IEEE (1968) has developed standard models for excitation systems of electrical generators. These models contain non-linear saturation functions. Analytical expressions are not given for these functions. The standard contains instead procedures for how to calculate those from data. These models should of course be available in a model library. It would certainly be convenient if such functions could be supplied as parameters. Parameters have the important advantage that they may have default values. This means that unsophisticated use of the model could be simplified.

The design of user interfaces for parameter handling is also important. Complex submodels in a model library should have a nice user interface so the user can get quick answers to relevant questions. When it concerns parameters of the model, he should be able to get a description of them without having to

inspect each submodel. The simulator could for example display descriptions of parameters in a window on request. It should be possible to set parameters both from the terminal using mouse and menus (tables) and from a user defined macro. It should be possible to store and retrieve parameter values from files. Sometimes, different parameter settings corresponds to different cases that the users have names for, for example: design1, model540, lowpower and highpower. He should then of course be able to reference these cases by name.

To prevent unintended abuse of a model, the model developer should be allowed to define ranges for the parameters, and a user should not be allowed to set a parameter outside its allowed range. The information about allowed ranges for parameters may also be useful in another context. It is possible to use symbolic formula manipulation to facilitate numerical calculations. It may be correct to make a certain manipulation for most parameter values, but there may be certain combinations that cause division by zero. However, in many cases those combinations never will show up, for example, because they are non-physical. This indicates that it could be useful to have ranges for parameters. The simulator may find the problem considerably simpler if some conditions on parameter values are fulfilled. It is then desirable if the simulator could ask if this is the case. The user may be able to confirm that the conditions are met in his case. When developing a model, it is often not obvious what is critical.

2.3 Multiple Versions

In Section 2.2 it was indicated that it would be useful to let a submodel be the description of a physical component. For a given component there are typically several possible models. Models of different complexity are used for simulation of different events, since it is unpractical and probably impossible to make a model which can simulate all aspects of a given component. The "complete" model would be cumbersome. For example, simulation of normal operation on one hand and malfunctions and emergency situations on the other hand, may require completely different models. Phenomena that can be neglected during normal operation could be of outmost importance in an emergency situation. In other words, models that are good for simulation of normal operation could become invalid in emergency situations. Consequently, there are reasons to introduce concepts so a model could contain alternative descriptions. Before discussing these concepts, let us indicate two additional reasons for introducing them. First, when developing a new model it is convenient to start with a simple model and test it, and then extend it stepwise by adding new features, while retaining the old versions for comparison. Second, a "simpler" version may also have its intrinsic value. Simulation is often used for empirical studies with the intention to gain understanding. When studying the importance of various dynamics it is very useful to be able to include or exclude different features of the model. Inclusion and exclusion of dynamics can to some extent be handled by parameters. Sometimes the simpler version could in theory be obtained be setting a real valued parameter of the more complex model to an extreme value (typically zero or infinity). However, this may lead to numerical difficulties as overflow, division by zero, stiff system, etc. For example, it might be of interest to study how the elasticity of a robot link degrades the performance by comparing it to that of a robot having a rigid link. We could do that by increasing the stiffness parameter of the link, but this will

unfortunately imply that the ODE system becomes stiff and difficult to solve. To avoid this, we could make the model more sophisticated. The drawback is then that the model may become messy and difficult to understand. The model would be much more transparent if we instead could have two separate descriptions and not one interlaced version. Support of multiple versions is a way of introducing abstraction and multiple views.

We propose a version concept which allows a model to contain a number of versions. To simplify changes, the versions should have a common interface to the environment. This means that the versions should have common terminals and parameters. Since the versions are meant to describe the same component, it is natural to let them have the same terminals. It is also natural to let them share parameters.

When designing a concept for handling multiple versions, it is important to consider the relation to the submodel concept. The intention of the version concept is to allow multiple descriptions of the behaviour of a component, whereas the submodel concept is meant to allow decomposition of a model into submodels; a submodel could contain a number of versions and each version could consist of a number of submodels. The submodel is hierarchical whereas the version concept proposed is flat.

It is desirable to have a mechanism that allows version selection of a model's submodels. This can be achieved by letting a standard parameter of the model define which version that should be used. We then have the power of the parameter concept and can at various levels define proper combinations of versions of submodels. Since the model version concept is basic, it might be a good idea to introduce special user interface facilities for selection of versions besides those available for setting parameter values. It might also be of interest to be able to switch model version during a simulation. Typical applications are simulation of malfunctions and emergency versions where the behaviour of a component changes drastically. When switching models it must be possible define the initial values of the new states in terms of the old states.

In Section 2.1 it was indicated that it is desirable to be have models that are flexible with respect to process design so it is easy to exchange components. The submodel concept and the version concept offer mechanisms that make it possible to introduce flexibility. However, the desire to have this kind of flexibility implies that the model must be designed more carefully. The model developer may have to introduce more levels. As an example, consider the case mentioned in Section 2.1, where we indicated that it could be useful to be able to switch between a synchronous generator and a asynchronous generator. A way to handle this could be to define a model `Generator` that abstracts the common properties of these to types of generators. The model `Generator` should have two versions: an asynchronous generator model and a synchronous generator model with an excitation system. It could have terminals describing driving input torque and electrical quantities at the generator terminal. The parameters could be selected to be more performance related.

2.4 Combined Discrete-Continuous Simulation

The dynamics of a system can be modelled in different ways. There are two basic and well-established modelling styles, namely continuous-system modelling and discrete-event modelling. Continuous-system models describe the behaviour or state at all times, whereas discrete-event models give snap-

shots of the model state at instants of "interest", but say nothing about the behaviour during intervening time periods. Ordinary differential equations (ODE's) and partial differential equations (PDE's) are used to describe behaviour in continuous-system models, whereas sequences of time events and difference equations are used in discrete-event models. Although it is not common today, there is a need to be able to also discuss more qualitative about systems and to handle incomplete and uncertain information.

In many applications, it has turned out that many models can be naturally expressed in a combined framework, combined discrete-continuous modelling. In this section we will discuss how the desire to support combined discrete-continuous modelling influences the design of structuring concepts. The critical point is modelling of interaction between submodels. A basic difficulty is that an input to a continuous-system model must be defined all times, discrete-event models only have their outputs defined at certain time instants. Consequently, if we want to use an output of a discrete-event model to define the input of a continuous model, we must decide what should be done during intervening time periods when the output of the discrete-event model is undefined.

Before discussing the possibilities and difficulties of modelling interaction between submodels in combined discrete-continuous models, we will give some motives for at all supporting combined discrete-continuous modelling.

Motives

ODE's, PDE's, difference equations and sequences of discrete events are all useful in automatic control engineering to describe system dynamics. ODE's are fundamental to describe dynamics in automatic control and a simulation tool must be able to handle them. However, it is a very difficult task to develop a general purpose simulation tool for PDE's. It is not the scope of this report to discuss tools for simulation of PDE's. To get simple and manageable models, a model developer often avoids to use distributed parameter models. He may discretize the model using finite element techniques. He may assume that the distribution over a volume has a certain parametrized shape or even that it is constant. He can then use ODE's to describe the dynamics. Transform techniques could be used to solve linear PDE's.

Difference equations are important in automatic control, since many controllers are implemented digitally.

There are several reasons for using discrete events in what is basically a continuous-system model. First, the purpose of a continuous-system model is to describe how a system behaves transiently with high resolution in time. However, sometimes a transient is so short with respect to the time scale, that we could view the transient phase to be infinitesimally short. Such an idealization often implies that the model could be made conceptually simpler, since we then do not have to describe the transient, but just define how the state after the transient should be calculated from the state before the transient. An example is the classical way of modelling a bouncing ball assuming an infinitesimally short time of bounce. In continuous-system simulation such an idealization cannot be done, since it leads to forces having Dirac delta components. Second, it is inconvenient to count pulses or to detect zero crossings in terms of ordinary differential equations and difference equations. You can use sampled models with high sampling rate, but this is inefficient. Third, there is a "metareason" to use discrete-event concepts. Sometimes, it is convenient to

be able to change models when certain events occur. Examples are modelling of batch processes, malfunctions and emergency situations. In these situation the dynamics may change drastically. The models could be made more readable if we could have separate models and rather than one interlaced version. This need for model structuring was mentioned in Section 2.3, when discussing the use of having multiple model version.

When is a relation valid?

In Section 2.2 is was motivated that equations (relations) containing terminal variables should be used to describe interaction between submodels. Without explicitly mentioning it, we assumed that those relations should be valid for all times, where the time was assumed to be continuous. In discrete models the behaviour or the state is only defined at discrete time instants. Consequently, it is not quite trivial to define the meaning of a connection between a terminal variable of a continuous-time model and a terminal variable of a discrete model.

As an illustrative example of the difficulty, consider a plant that is controlled by a digital controller. A digital controller is conveniently described by difference equations. Let the submodel have the input u_d and the output y_d. Assume that the plant is modelled as a continuous-time system with an input u_c and an output y_c. To model the interaction between the plant and the controller, let us connect y_c to u_d and y_d to u_c. In the framework discussed in Section 2.2 this would generate the "equations" $u_d = y_c$ and $u_c = y_d$. These look very similar, but for us they should have quite different meanings. The normal, idealized, mental picture of the digital controller is that at a sampling instant the input should be read, and the state and the output should be calculated. We also in most cases imagine that the output actually is defined and constant between the sampling instants, as we assume zero-order sample-and-hold.

A critical step in the interpretation of the connection "equations" $u_d = y_c$ and $u_c = y_d$ is how we would like to view the variables of the discrete model between the sampling instants. The example indicate that it is useful to let the output y_d be defined to model zero-order hold. The input u_d need not be defined between the sampling instants, but for the user of the simulator, it may be useful to have access to u_d at any time. For example it may facilitate the debugging of the model. We may remark that the state must be preserved between the sampling instants and it is then very natural to assume that it is also defined between the sampling instants, because it must be stored somewhere.

Let us now assume that the values of the terminals of a discrete model are defined also between sampling instants and that the values are those calculated at the latest sampling instant. The connection "equation" $u_c = y_d$ is then a relation which is valid at all times, but in contrast to connections between continuous-time models the causality is given, since the relation should define u_c. On the other hand, $u_d = y_c$ is valid only at the sampling instant. The input u_d should be set to y_c at the sampling instant and keep that value until next sampling instant. Moreover, $u_d = y_c$ should not put any direct and explicit constraints on y_c. It should merely imply that the value of y_c are read now and then.

The discussion indicates that we have to define the semantics of a connection between a continuous-time model and a discrete model more specificly. You may have hoped that the simulator could sort out these issues itself. Un-

fortunately, this is not the case. To realize this, consider the example discussed above. It would then be possible for the simulator to assume that y_d is the input and u_d the output. This view would give the composite model quite another behaviour.

There are different ways of letting the model developer define the meaning of a connection between a continuous-time model and a discrete model. First, he could do it when making the connection. Second, the model developer of a discrete model could define the terminals as being either inputs or outputs. We recommend the second approach for three reasons. First, discrete models are very often models of digital systems or computer programs. The causality should then be given and it is actually not difficult to define which terminals that are inputs and which that are outputs. Second, on the contrary it must be viewed to be favourable to define the causality, since this allows the simulator to make additional consistency checks of the the connections. Third, a model developer putting together library models may find the first approach laborious. He may have to answer a number of questions when making a connection, since a terminal may be composite. In the second approach, he gets messages only when he makes an erroneous connection.

We have above talked about continuous-time models and discrete models. But what about combined models? They will certainly be created when submodels of the two types are connected. That is no problem when considering the connection semantics. The encapsulation of composite models is only introduced for convenience and the terminals of composite models are just intermediaries which easily could be removed. Assuming that this has been done, then all connections are connections between non-composite models. If we introduce a rule saying that a non-composite model should be of either type, we in fact have no combined models when interpreting the connections. There are other good reasons for not allowing mixed non-composite models. If a non-composite submodel could be of a combined type, we must for each of its variable declare explicitly whether it is a continuous-time variable or a discrete-time variable. We must also indicate the interpretation of each "equation" that involves variables of both types. As you can understand, this could easily result in messy, unreadable and erroneous models. A rule saying that a non-composite model must be of either type enforces structure. Furthermore, we need not introduce a concept for describing the semantics of variables and relations. It is given implicitly by the type of the model.

Sampled models – discrete-time models

We will now consider the problem of defining the sampling instants and the decomposition of discrete models.

It is desirable to have the same structuring concepts when we are using difference equations to describe behavior as when using ODE's. The difficulties discussed above were due to different time concepts. It is possible to decompose the model into submodels and use relations to describe the interactions between submodels if all submodels in the hierarchy have common sampling instants. This is for example the case when we would like to make a model of a complex digital control system and neglect calculation times. The difficulty arises when we want to model components that are not synchronized or interact with parts that are modelled by continuous-time models.

To solve this problem we propose the introduction of two discrete submodel types: discrete-time and sampled. From a structural view, the discrete-

time model is equivalent to the continuous-time model, except that difference equations and not ODE's are used to describe behaviour. Both types can be decomposed into submodels and interaction between the submodels could be described by interactions. A continuous-time model assumes that the time is defined globally. So does the discrete-time model, but it also assumes that the sampling instants are defined at a higher level.

The sampled type is introduced to make it possible to combine continuous-time and discrete-time models. A sampled model should have the external properties of discrete models discussed in the preceding paragraph. The causality of the terminals should be defined by the model developer; the model should have inputs and outputs. At a sampling instant the sampled model should read the inputs and with the internal state calculate the outputs. The encapsulation actually functions as a sample and hold of the inputs and outputs. If the internal description of the behaviour sets the output equal to the input, we have a model of a zero-order hold device. Discrete-time models could be used to describe the internal behaviour of a sampled model. The sampled model then defines the sampling instants of its discrete-time submodels. It is for us an open question whether the use of continuous-time and sampled submodels for describing the behaviour of a sampled system should be supported. To motivate such a support we must have some applications where it would be useful. In the following we will assume that only discrete-time submodels may be used to describe behaviour of a sampled model.

You may think that the introduction of the sampled model type makes the relation between continuous-time and discrete-time models unsymmetrical. Why not let the sampled model type encapsulate continuous-time models instead? When designing digital controller, it is important to study also how the system behaves between the sampling instants. For obvious reasons, it is not a very good idea to try to restrict the description of the behaviour of the continuous-time parts to the sampling instants. Another answer is that the problem of connecting two discrete-time model having different sampling instants is similar to that of connecting a continuous-time and a discrete-time model. The concepts discussed above also solve the problem of interconnecting discrete systems having different sampling instants. In this way we can handle multi-rate sampling in a convenient way.

With discrete systems we must be able to define the sampling instants. A simple, but powerful way to do this, is to let each sampled subsystem have a variable that defines the time of the next sampling instant. If we have several discrete subsystems, it may be important that they are sampled synchronously. If they should be sampled at identical time instants, this is easily handled by letting the subsystems be of discrete-time type, since then the encapsulating sampled model defines the sampling instants. If we want the submodels to be sampled synchronously but with phase shifts, this could be done by making submodels of sampled type. If the different submodels calculate their sampling instants themselves, numerical imperfections may introduce drift so the samplings after a while are not synchronous. This problem is avoided if the calculation of the sampling instants is centralized in one place. The model developer can introduce a submodel of sampled type that models a clock having outputs that defines the samplings instants. The other submodels can read this value and set their sampling times accordingly.

Discrete-event models

Discrete-event modelling is well-established and it has been used in many applications. An overview with many references are given in Kreutzer (1986).

If we say discrete-event simulation and encapsulated submodels, it is natural to come to think about object-oriented programming. You could say that object-oriented programming has its roots going back to discrete-event simulation, since many ideas behind object-oriented programming have their roots going back to Simula, which is a programming language for discrete-event simulation. In the object-oriented approach the submodels could be viewed as active objects and the interaction could be modelled by message sending between the objects. The behaviour is described by procedures.

Let us recall the approach indicated for continuous-time simulation. It was argued that declarative forms (equations) should be used to describe behaviour of submodels and interations between submodels to make it feasible to build model libraries. It is a global problem to make a procedural description or make the algorithm from which the behaviour can be calculated. It cannot be made for the individual submodels independently of how they are used. There are simulators which accept discrete-event models given on declarative forms (Kreutzer, 1986), for example, as sets of rules. There is a close relationship with data and knowledge representation in artificial intelligence, knowledge-based engineering and expert systems. Note also that models given on declarative forms can also be used in other contexts than simulation.

We will not here discuss the support of general discrete-event modelling, but we will focus on how ideas from discrete-event modelling can be used in a basically continuous-system modelling style.

Motivated by such examples as those given in this section under "Motives", we propose a new submodel type called discrete-event model having the following properties. A terminal should be either input or output. An event is defined as a zero-crossing from below of an indicator expression, which may involve time, inputs and other variables of the discrete-event model. A discrete-event model should be able to wait for one or many events. When an event occurs, the waiting discrete-event model should become active. Refering to our examples, we want the discrete-event model to be able to initiate switching of models and to set the states of other models. If we just want to count the number of events this could be handled internally. It is not the topic of this chapter to discuss the internal format of a discrete-event model. However, it could be noted that to support the use indicated by the examples, a discrete-event model could be kept simple.

When a submodels receives an order to switch version, it must be able to calculate the initial values of the new states from the states of the old model version. This need was also mentioned in Section 2.3.

The example of modelling a bouncing ball in the classical way indicates that it should be possible to manipulate the state of a submodel. One approach is to let the submodel owning the states handle the calculation of the new state so the discrete-event model just has to initiate the calculation. This approach could be considered as a special way of switching model version. Another approach would be to let the discrete-event model do the calculation. However, the desire to have encapsulated submodels makes this approach unfeasible.

Let us consider how events could be detected when solving ODE's numerically. Events that are specified by the clock are easy to detect and handle. It can be handled by integrating the equations up to that time and then

restart the integration routine. This means that it is almost straightforward to simulate models consisting of both continuous-time submodels and sampled submodels. If the event depends on the state, the problem becomes more complex, since we must be detect if and when events occur. Cellier (1979) proposes an approach where indicator functions tell the integration when events may occur. The integration goes on until an indicator function indicates an event. The step-size control mechanism of the integration routine is then disabled and the step is repeated with a new step size computed by a special iteration scheme to establish if and when an event has occurred. It is important that the indicator functions are calculated with enough accuracy and resolution in time so events are detected. One way to get some automatic control of the calculation of the indicator functions is to introduce the indicator functions as extra states. Thus making the integration routine explicitly aware of the indicators, so it adjusts the step-size also with respect to the behaviour of the indicators. However, the method does not guarantee that fast multiple switches are detected. If the model is supposed to model something and the behaviour of the model exhibits multiple fast switches, the user must really consider whether important dynamics has been neglected. It may then be important to model the transitions in more detail.

Handling of events is closely related to handling of discontinuities. There are efficient and robust numerical ODE solvers for systems consisting of ODE's on state space form when the state derivatives are continuous. Cellier (1979) shows that discontinuities cannot be handled by step length control, since the integration routines can miss fast switches and give a completely false result. His approach is to view the problem as a continuous problem and to view a discontinuity as an event which means that we should switch to a new continuous problem. Symbolic analysis can be used to set up the indicator functions automatically. If the model is given in symbolic form, it is rather easy to detect discontinuities by scanning through the equations.

Cellier concludes that to handle discontinuities properly, an event concept with indicator functions must be introduced. In Cellier's approach the integration routine must not switch to the new set of equations until it has been established that an event has occured. Note that this implies that the numerical integration routine may want to evaluate the model outside the validity range of the model. Consequently, the derivatives must be defined so this evaluation does not cause numerical overflow or division by zero.

3. Visualization

Graphics plays an important role in control engineering. First, many of the techniques for the analysis of control systems rely on graphics; trend curves, phase planes, Bode plots, Nyquist plots and root locus plots. Second, it is natural for a control engineer to use block diagrams when describing a process or a model. Unfortunately, the use of graphics in CACE has been hampered by lack of feasible hardware for a long time. However, the situation is changing drastically. The new workstations with high performance, real-time graphics offer new possibilities to introduce graphics in CACE.

In this chapter we will focus on the use of graphics to visualize systems and their behaviour. In Section 3.1 we discuss the use of hierarchical block diagrams to describe the model decomposition and interaction structure. Section 3.2 describes an introductory study of a window-based environment for the simulation package Simnon on the Sun workstation. An example of animation to visualize behaviour is given in Section 3.3. Implementational and standardization issues for graphics and window systems are discussed in Section 3.4.

3.1 Hierarchical Block Diagrams

Hibliz

In the project "New forms of man-machine interaction" (STU project 84-5069) (Mattsson, Elmqvist, Brück, 1986) a prototype simulator for dynamical systems was developed and implemented. The purpose was to investigate some possibilities of using graphics to improve the user interface. The structural properties of a system or a model are very important particularly when working with complex or large systems. These structures are, however, difficult to represent in an easily apprehendable way when a purely textual description is used. For example, the interconnection structure of subsystems is much easier to describe graphically.

The simulator supports hierarchical block diagrams to describe the model decomposition and the interaction structure. At the highest level the user sees a diagram of annotated boxes connected with lines. The user can scroll, pan and zoom the block diagram continuously in real-time. Zooming controls the amount of information displayed. When zooming in on a block, it changes from an annotated box into a representation showing internal structure with increasing detail. Since the block diagrams can be hierarchical, it is possible to make the description at each level simple and clear. The simulator is called Hibliz which stands for HIerarchical BLock diagrams with Information Zooming. The user creates and edits his block diagrams in a Macintosh-like fashion. He can also create overview windows which indicate where he is when he pans and zooms. Hibliz also simplifies model development by allowing submodels in the form of ordinary differential and algebraic equations rather than assignment statements for derivatives and algebraic variables.

Assessment of Hibliz

Many demonstrations for people from university and industry indicate that the hierarchical block diagram is a natural and easily understood concept for describing model decomposition. Hierarchical block diagrams also make the model more concrete. The user can view the model as an object. The concept of seeing and pointing is important. For example, to inspect a model the user can just point at it and zoom-in.

An application study of modelling a destillation column in a number of languages has been performed (Nilsson, 1987). The study focused on how the model structure could be described. The model was decomposed into nine trays, one reboiler and one reflux drum. An interesting structural feature of this model is that the nine trays are identical and are connected in series. Today's commercial simulation languages have no graphical interfaces for describing model structure. Nilsson found that the hierarchical block concept of Hibliz allowed a very illustrative way of describing the model structure and made it possible to create nice overviews of the whole model. However, he found it laborious to create the hierarchical diagram. The user interface of Hibliz does not allow the user to explore the fact that he would have nine identical trays in series. The user has to create one model for a tray. When he has done this, he can copy the model nine times and lay out and connect them manually. It is understandable if a user finds this work laborious. However, as stated by Nilsson there is a tradeoff between simplicity and power of the commands. The drawing of block diagrams could possible be facilitated by inclusion of a grid, automatic alignment procedures etc.

Another serious drawback with Hibliz is that it does not support a model type concept. When the user copies the tray model, the copies become completely independent of each other. This means that maintenance of the destillation column model becomes laborious. If the user wants to modify the tray model, he has to edit all nine tray models. The maintenance would be much simpler if there was a type concept that allowed all nine tray models to share a common description.

In summary, we recommend hierarchical block diagrams as a general tool for describing model structure. However, we would also like to stress that there are many open questions concerning their look and the means for creating and editing them. Should symbols (icons) be used to denote different parts instead of annotated boxes? How should keyboard and mouse be used when creating, editing and inspecting models? Continuous scrolling, panning and zooming demand fast graphics. It is not necessary to have these features to implement hierarchical block diagrams. The zooming up of a block can be implemented by creating a new window and displaying the block in this window.

The new approach

To evaluate ideas, when developing user interfaces, you have to implement them to get feedback. This implies that it should be easy to make prototype implementations. Hibliz is written in Pascal and it is laborious to modify it. So we decided to freeze Hibliz and start from scratch. The experiences from other subprojects had indicated that Lisp gave a nice interactive environment. When the project started we had only the IRIS 2400 workstation available. We upgraded it by purchasing another 70 Mbyte Winchester disk and by increasing the CPU memory from 2.5 Mbyte to 6.5 Mbyte. The Extended Common Lisp implementation from Franz Inc. was purchased. This Common Lisp

implementation provides interfaces to C procedures and the IRIS Graphics Library. The EMACS editor was also installed on the IRIS. The IRIS window manager Mex (multiple exposure), EMACS and Common Lisp turned out to give a nice interactive environment.

Our aim was to make the software portable. The use of Common Lisp guaranteed to some extend portability. However, the fact that we would like to use graphics and windowing facilities made the thing worse. Standardization issues are discussed further in Section 3.4. Our approach was to isolate the implementation dependent parts in a graphical front-end. To make it easily portable, we studied a number of workstations and windowing systems to see which graphical operations that commonly were supported or easily could be implemented. The graphical front-end was implemented on the IRIS. It is discussed further in Section 3.4. Here we will only mention that it was more laborious and took longer time than we had estimated when we started. Hopefully, we could avoid such work in the future.

A minor prototype for creating and editing block diagrams has been developed and can be used for rapid prototyping. The support of the window manager make it easy to implement information zooming by open up a new window. Also creation of menus are supported.

3.2 A Window-Based Environment for Simnon

Despite of how you organize information, there will always turn up situations where the user would like to access information from two different places simultaneously. A way to accomplish this is by supporting windowing.

Professor Dean Frederick, Rensellaer Polytechnic, Troy, New York, USA, participated in this project as a guest researcher for two month (May 15 – July 15, 1987). When he arrived, the department had just got four Sun Microsystems 3/50 workstations. It was decided that he should make some initial efforts to develop a modern workstation-based version of the simulation program Simnon on the Sun workstation. The reasons were as follows. First, he had various experiences of using graphics in CACE. Second, it would give us experiences of using the Sun workstation in this kind of applications. Third, the result could be useful. Simnon is commonly used at the department and with four workstations there is a potential to get feedback from many users.

The result of his effort is described in Frederick (1987). To this date six window-related commands have been added to Simnon:

`mkwin`	a graphics window for plotting
`wedit`	an editing window
`wfdir`	a directory facility for models and macros, with editing
`wdisp`	a window for the Simnon `DISP` command
`wprint`	a window for the `PRINT` command
`whelp`	help on the window-related commands

These commands can be issued from the keyboard when Simnon is displaying its prompt. For window management the SunView software has been used. This means that the windows created by Simnon behave and can be controlled as ordinary, native windows on the SUN. The windows can be repositioned, resized, closed to become an icon or deleted. Text windows can be scrolled. In plotting windows created by issuing the `mkwin`, all of the usual Simnon

graphics related commands, like for example `SPLIT` and `AREA`, can be used with the usual arguments. The implementation is usable, but unexpected crashes occur. Frederick (1987) suspects that it is the communication between Simnon and the window system that causes the crashes. Dean Frederick and Tomas Schönthal at our department have plans to continue the work.

3.3 Animation

An exciting idea is to present the results of simulations or data logging as some kind of animation of the process studied. One possibility to visualize results is to simulate an instrument and to present the results in windows as if you were viewing them on real devices. Mimic diagrams is another possibility. The data then controls positions, sizes, colors and visibility of objects in the picture. For example, a mimic diagram can show the level in a tank. Color can indicate the temperature of the content. Warning indicators may turn on when the level becomes too high. Note that animation is also useful for demonstrations for non-experts and customers. It must be pointed out that the design of the animation must be done carefully so it does not become more spectacular than useful.

Since it is a large effort to develop general, interactive tools for defining mimic diagrams or more general stylized representations for animated cartoons, and since there already exist good implementations on the market, we have decided not to develop own such tools.

Our IRIS 2400 has the possibilities to make fancy real-time animations. It is for example capable of real-time animation of a robot in 3-D graphics. Ola Dahl has modified the demonstration program `robot` from Silicon Graphics to display ASEA's Industrial Robot IRB 6/2. The program can read the coordinates of the robot joints from a file, which can be the result of a simulation or a logging of a real experiment. When the robot performs its movements, the user can using the mouse move around the 3-D world and study how the robot performs its work cycles. This animation feels very real.

Dr. Hirzinger, DFVLR Deutche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt e.V., Oberpfaffenhofen, West Germany (close to Munich) has developed a steering ball with six degrees of freedom. It senses the forces and torques you apply to it using your hand. It can be used in various ways. A natural way to use it for controlling objects on a graphical screen is to let a force in the vertical direction mean scrolling, a left or right horizontal force mean panning, and pressing mean zooming. Torques can be interpreted as a desire to have the object on the screen to rotate. With fast visual feedback the steering ball is very natural and easy to use. It is a terrific input device that can make graphics workstations more powerful.

DFVLR have a number of interesting applications of animation. One application is animation of robots in space. It is difficult to control robots in a space station manually from earth since there could be a considerable time delay. The idea is to superpose graphics on the video picture of the robot. The human operator on earth could by controlling the animated robot virtually get rid of some of the time delay effects.

We have purchased a steering ball and used it in the robot program. The user can use the steering ball to modify his position relative the animated robot and in this way view the robot from different distances and angles of view when the robot is performing its work cycle.

3.4 Graphics and Window Standards

Portability is a major concern when designing and implementing software for CACE. Much of the problem is overcome by programming in a language that is supported on many computers. Unfortunately, the use of languages like Fortran, C, Pascal, Modula-2, Ada, Common Lisp etc. does not eliminate all potability issues, since the use of graphics is important in CACE.

Current Status

There is today one generally accepted graphics standard: GKS (Enderle et al. 1984; Hopgood et al. 1983). GKS is rather low-level, but all high-level operations in an integrated engineering environment can probably be implemented on-top of GKS. Regrettably, most of today's implementations are poor. They are unacceptably slow. A basic problem is that most implementations do not support sampled or event driven input, implying that they cannot be used for interactive "workstation-type" graphics. Another problem is that the use of GKS more or less forces you to select a programming language for which GKS has defined standard name bindings. This means for example that you cannot use C. There are no standard C name bindings in GKS, since C itself is not standardized.

Another maybe more difficult problem is windowing. A good overview of relevant issues and some existing window systems is given in Hopgood et al. (1986). Another interesting overview is Rosenthal (1986). Standardization cannot be expected before 1989. One important source of difficulties is the interaction between the window manager and the graphics package. The windowing facilities are normally not accessible from the graphics packages, and user input is handled differently. It is extremely frustrating to have to use two vastly different ways of interaction simultaneously. Another aspect is that a user does not like to learn several window systems. The advantages of having a standard window system for all CACE programs are quite obvious and uncontroversial, but it should be noted that CACE programs are not the only use of a workstation. The user will use the native, vendor-supplied window manager, and would therefore prefer that one also in CACE programs. The same also applies for text editors.

To sum up we can say that current status is a bit chaotic. Each vendor of workstation has there own graphics and window systems. There are also emerging third-part systems that could be used on a number of workstations.

Future Trends

An extended and improved ANSI standard, PHIGS (Programmer's Hierarchical Interactive Graphics System) (SIS 1985; Shuey et al. 1986; Brown, 1985), is proposed. PHIGS seems to have valuable high-level features, in particular a dynamic, hierarchical structure for graphics. It should be noted that PHIGS is very strongly promoted by IBM.

One problem area (which may eventually solve itself) is the lack of experience of using PHIGS. We do not yet know what "programming style" best explores the strengths and features of PHIGS. Neither is the relationship to object-oriented programming understood. To get experience of using PHIGS, we have ordered a PHIGS implementation called FIGARO for our IRIS 2400 (FIGARO, 1986; Plaehn, 1987). FIGARO is developed by Template Graphics Software Inc., San Diego, California, USA.

We are not aware of any prosed standard for window systems. However, X-windows owned by MIT appears to become a de facto standard.

Our approach

When the project started, we anticipated that we had to run the software on at least two different workstations. Not having a standard to rely on we decided to isolate graphics and window system dependent issues in a graphical front-end module to facilitate porting to another workstation. We studied a number of workstations and their graphics and windowing systems to see which graphical operations that commonly were supported or easily could be implemented.

The design and implementation of the graphical front-end is described in Brück (1987). It handles input from keyboard, pointing device (mouse) and stored files and can draw on a graphical screen. There are 34 operations divided into eight groups:

1. Drawing primitives: `move`, `draw`, `rectangle`, `fillrectangle`, `circle`, `fillcircle`, `setcolor`, `setlinewidth`
2. Text: `drawstring`, `requeststring`
3. Local, hierarchical coordinate systems: `scale`, `translate`, `pushmatrix`, `popmatrix`
4. Hierarchical segments, which can be appended to but not edited: `open`, `close`, `delete`, `call`, `highlight`
5. Picking (multiple hits has to be resolved by the application): `pushmarker`, `popmarker`, `requestpick`
6. Rubberbanding: `requestline`, `requestrectangle`, `requestshape`
7. Reshapable and overlapping windows: `create`, `title`, `bind`, `limits`, `erase`, `redrawwindow`, `redrawall`
8. Definition of menus, which at selection returns text commands. `newmenu`, `addtomenu`, `requestmenu`, `mainmenu`

The current version runs on the IRIS 2400 workstation. The IRIS window manager Mex imposes certain requirements on the implementation. A basic question is "who should be responsible for redrawing the windows when the user moves or resizes windows: the window manager or the application program?" The problem is discussed in Hopgood et al. (1986). From the application programmer's point of view, the redraw request should be hidden where possible. However, it must be recognized that redraw requests to the application program are an inescapable fact of life, for example, when there is insufficient memory to hold off-screen copies of windows; the screen of our IRIS has a resolution of 768*1024 pixels and 24 bitplanes implying, that a copy of the screen requires 2.4 MB of memory. Another problem is that resizing a window might imply that the content of the window also should be scaled and in that case the old raster copy is not useful. The picture must then be regenerated. This is easily done when a picture is stored in a high level format (a segment).

The IRIS window manager Mex does not handle redraws itself, but puts a special `redraw` token on the same queue that is used for mouse input. These requests must be served as fast as possible since all processes owning windows may be blocked until all redraw requests are served. In our implementation, this is handled by letting the graphical front end be a separate Unix process.

The graphical front-end stores the content of each window as a segment and can in this way handle redraw requests itself. An invokation of a graphics routine from the application means that a message are sent from the application process to the graphical front-end process.

4. Conclusions

The purpose of this work has been to study various aspects of system representation, which is an important issue in CACE. The work has been a continuation of earlier CACE subprojects in this area: "New Forms of Man-Machine Interaction" (STU project 84-5069) (Mattsson, Elmqvist and Brück, 1986) and "High-Level Problem Solving Languages for Computer Aided Control Engineering" (STU project 85-4808) (Åström and Mattsson, 1987).

A basic idea is that the user interface should be separated from the processing parts to get flexible tools allowing adaptation of the user interface to the needs and the desires of various users. The representation of systems is an important and critical part of a CACE system, since it should be unified to integrate the various. We believe that it is possible to agree upon a common set of modelling concepts, but that it would be useful to allow the concepts to have different textual and graphical representations. Basic modelling concepts were outlined in Chapter 2 and five complementary modelling structuring principles were proposed.

Another basic idea is that that declarative forms (equations) should be used to describe the behaviour of submodels and the interactions between models. Declarative models on symbolic forms are flexible, since they can be used in various contexts. They can be manipulated automatically to generate efficient code for simulation, code for calculation of stationary points, linear representations, efficient control code, descriptions which are accepted by other existing packages etc. Furthermore, it is a global problem to make a procedural description or make the algorithm from which the behaviour can be calculated. This cannot be made for the individual submodels independently of how they are used.

Chapter 3 focussed on the use of graphics to visualize systems and their behaviour. First, hierarchical block diagrams and Hibliz were discussed. The conclusion is that it is a good idea to use hierarchical block diagrams to describe model structure, but there are many open questions concerning their look and the means for creating, and editing them. We are currently testing a number of ideas. Second, a prototype window-based environment for Simnon on the Sun workstation has been developed. This gave us new experiences of graphics and window systems. The prototype is usable and can thus give us user feedback. Third, a program for real-time animation in 3-D graphics of ASEA's Industrial Robot IRB 6/2 has been developed. The animation feels very reel. Fourth, we have discussed graphics and window standards.

The work has been influential in shaping the future of the CACE project (Mattsson, 1987), which will be focussed on development of prototype tools for model development and simulation.

Acknowledgements

This project "Representation and Visualization of Systems and Their Behaviour" has been a part of the project Computer Aided Control Engineering (CACE) at Lund Institute of Technology. We are grateful to the National Swedish Board of Technical Development (STU) who has supported this project under contract 86-4049.

I would like to thank Professor Karl Johan Åström, Mats Andersson, Dag Brück, Ola Dahl, Bernt Nilsson and Tomas Schönthal for their participation in various parts of the project. Many thanks also to Professor Dean Frederick, Rensselaer Polytechnic Institute, Troy, New York, for his participation as a guest researcher.

References

ÅSTRÖM, K.J., and S.E. MATTSSON (1987): "High-Level Problem Solving Languages for Computer Aided Control Engineering," Final Report 1987-03-31, STU project 85-4808, STU program: Computer Aided Control Engineering, CACE, Report CODEN: LUTFD2/TFRT-3187, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BROWN, M.D. (1985): *Understanding PHIGS – The Hierarchical Computer Graphics Standard*, Template, The Software Division of Megatek Corporation, San Diego, CA, USA.

BRÜCK, D.M. (1986): "Implementation of Graphics for Hibliz," Report CODEN: LUTFD2/TFRT-7328, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D.M. (1987): "Design and Implementation of a Graphical Front-End," Report CODEN: LUTFD2/TFRT-7367, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H. (1978): *A Structured Model Language for Large Continuous Systems*, Ph.D-thesis, Report CODEN: LUTFD2/(TFRT-1015). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H. and S.E. MATTSSON (1986): "A Simulator for Dynamical Systems Using Graphics and Equations for Modelling," *Proceedings of the IEEE Control Systems Society Third Symposium on Computer-Aided Control Systems Design (CACSD)*, Arlington, Virginia, Sep. 24–26, 1986.

CELLIER, F.A. (1979): *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*, Ph.D-thesis, Diss ETH No 6483, The Swiss Federal Institute of Technology, ETH, Zürich.

ENDERLE, G., K. KANSY and G. PFAFF (1984): *Computer Graphics Programming (GKS—The Graphics Standard)*, Springer-Verlag.

FREDERICK, D.K. (1987): "An Introductory Study of a Window-Based Environment for Simnon on the SUN Workstation," Report CODEN: LUTFD2/TFRT-7366, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

FIGARO (1986): *FIGARO Reference Manual*, Part I–II, Megatek Corporation, San Diego, CA, USA.

HOPGOOD, F.R.A., D.A. DUCE, J.R. GALLOP and D.C. SUTCLIFFE (1983): *Introduction to the Graphical Kernel Standard (GKS)*, Academic Press.

HOPGOOD, F.R.A., D.A. DUCE, E.V.C. FIELDING, K. ROBINSON and A.S. WILLIAMS (Eds.) (1985): *Methodology of Window Management*, Proceedings of an Alvey Workshop at Cosener's House, Abingdon, UK, April 1985, Eurographic Seminars, Tutorials and Perspectives in Computer Graphics, Springer-Verlag.

IEEE COMMITTEE REPORT (1968): "Computer Representation of Excitation Systems," *IEEE Trans. on Power Apparatus and Systems* **PAS-87**, June 1968, 1460–1464.

KOENIG, H.E., Y. TOKAD and H. KESAVAN (1967): *Analysis of Discrete Systems*, McGraw-Hill.

KREUTZER, W. (1986): *System Simulation: Programming Styles and Languages*, Addison-Wesley.

MATTSSON, S.E. (Ed.) (1987): "Programplan för ramprogrammet Datorbaserade hjälpmedel för utveckling av styrsystem (Computer Aided Control Engineering, CACE)," Final Report CODEN: LUTFD2/TFRT-3193, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E., H. ELMQVIST and D.M. BRÜCK (1986): "New Forms of Man-Machine Interaction," Final Report 1986-09-30, STU project 84-5069, STU program: Computer Aided Control Engineering, CACE, Report CODEN: LUTFD2/TFRT-3181, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1987): "Experiences of Describing a Distillation Column in Some Modelling Languages," Report CODEN: LUTFD2/TFRT-7362, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

PLAEHN, M. (1987): "A high performance PHIGS interface," Vendor Forum, *IRIS universe – the IRIS community magazine*, Summer 1987, Silicon Graphics Inc., Mountain View, CA, USA.

ROSENTHAL, D.S.H. (1986): "Window System Implementations – Denver Usenix Course Notes," Sun Microsystems, Mountain View, CA, USA.

SHUEY, D., D. BAILEY and T.P. MORRISSEY (1986): "PHIGS: A Standard, Dynamic, Interactive Graphics Interface," *IEEE Computer Graphics and Applications* **6**, No. 8, August 1986, 50–57.

SIS (1985): *Datorgrafi—PHIGS, Programmers Hierarchical Interactive Graphics Standard*, Technical report no. 306, SIS—Standardiseringskommisionen i Sverige, Sweden.