# Modelling of Control Systems with C++ and PHIGS

Brück, Dag M.

1988

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

CODEN: LUTFD2/(TFRT-7400)/1-010/(1988)

# Modelling of Control Systems with C++ and PHIGS

**Dag M. Brück**

Department of Automatic Control
Lund Institute of Technology
August 1988

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

*Document name*
INTERNAL REPORT

*Date of issue*
August 1988

*Document Number*
CODEN:LUTFD2/(TFRT-7400)/1-010/(1988)

*Author(s)*
Dag M. Brück

*Supervisor*

*Sponsoring organisation*
The National Swedish Board of Technical Development (STU project 87-2503)

*Title and subtitle*
Modelling of Control Systems with C++ and PHIGS

*Abstract*

This paper describes an interactive tool for modelling of control systems. The focus is on practical experiences with C++ as a development tool, and the need for multiple inheritance, parameterized types, and exception handling, in this application. Experiences with a new graphics standard, PHIGS, using an object-oriented programming style, are briefly covered.

This paper will be presented at the USENIX C++ Conference, October 17–21 1988, Denver, Colorado, U. S. A.

*Key words*
Computer Aided Control Engineering, Modelling, Programming languages, Programming tools, Data abstraction, Object-oriented programming, Exception handling, Computer graphics

*Classification system and/or index terms (if any)*

*Supplementary bibliographical information*

*ISSN and key title*

*ISBN*

*Language*
English

*Number of pages*
10

*Recipient's notes*

*Security classification*

# Modelling of Control Systems with C++ and PHIGS

**Dag M. Brück**

Department of Automatic Control
Lund Institute of Technology
Box 118, S-221 00 Lund, SWEDEN

E-mail: dag@control.lth.se

**Abstract**

This paper describes an interactive tool for modelling of control systems. The focus is on practical experiences with C++ as a development tool, and the need for multiple inheritance, parameterized types, and exception handling, in this application. Experiences with a new graphics standard, PHIGS, using an object-oriented programming style, are briefly covered.

## 1. Introduction

Modelling has traditionally been one of the main topics in control engineering. Control systems are complex and require careful design and analysis, in particular, as errors in control system design can become expensive. There exists today a great need for computer aided design of control systems.

Our research is centered around tools for model development and simulation. The objective is to design the basic concepts needed for structuring models, and to design the internal computer representation of control system models. An experimental tool for modelling and simulation has been developed in KEE, an expert system shell.

The experimental tool will form the basis of an engineering tool for the designer of control systems. In such a product, flexible, efficient and affordable system software must be used. We have therefore evaluated C++ as the future implementation language, and PHIGS as the main graphics system. A simplified experimental tool has been implemented in C++. Whereas the KEE version supports all essential parts of an engineering tool, the C++ version only provides graphical interaction; the internal structure is quite similar, in order to meet future needs.

## 2. Modelling of control systems

The model of a control system can be regarded as a hierarchy of components. One of the fundamental ideas is to build libraries of component models, ranging from basic items (for example, a pump) to more complex objects (for example, a distillation column). The designer has the option of working bottom-up, putting predefined components together to form a new component, or top-down, decomposing a complex object into manageable pieces,

or most likely, a combination of bottom-up and top-down design [Nilsson, 1987]. The key word is reuseability — of earlier designs and of standard components.

A single component can be described in many ways: graphically, textually, using block diagrams (describing its structure), or mathematically (for example, in state-space or transfer-function form). It is also necessary to use models with different degrees of detail and complexity, for example, an efficient simulation model for normal operation, and an extended model for analyzing error conditions. All these models are needed in different stages of the design, and should be available in a model development tool. It should be noted that the common "machine" view may be replaced by a "materials" view. For example, a chemical compound may carry all knowledge in the model, while the stations in the refinery only signal changes of state.

With our set of basic concepts, a model has three properties: it has terminals which provide an interface to the outside world, parameters for adapting its behaviour, and at least one realization that defines its behaviour. Only data in the terminals are available to other components; there are no global data, except a time reference for simulation.

We currently support two types of realizations: primitive realizations using ordinary differential equations, and structured realizations using block diagrams. A structured realization consists of submodels and connections (between submodels, and between submodels and the terminals of the enclosing model). Interaction between components is defined only by connections.

Simulation is often used to analyze control systems, and the designer should be able to simulate his/her model using this tool. Simulation introduces a number of interesting mathematical problems, which will not be covered further in this paper [Mattsson, 1988b]. The connection concept also raises interesting questions: for example, what is a legal connection, and how do you define compatibility between terminals [Mattsson, 1988a].

According to current trends, it is also necessary to throw in an expert system and a couple of knowledge bases.

## 3. Direct model representation

Modelling of control systems maps nicely to the ideas in object-oriented programming. It is natural to represent a model with a class in the programming language used for implementing the design tool. It is then possible to develop new models using inheritance and specialization of classes.

Inheritance is not suitable for describing all kinds of relationships between models. Multiple representations of a single model (textual or mathematical), and specialization (a car is a special kind of vehicle), can be described with inheritance. Decomposition of a model into its components is different. For example, that a car has tyres does not mean that the car can be inflated, so inheritance is not the right mechanism; components are represented by class members (Listing 1).

The direct way of representing models with classes is used in the experimental tool developed in KEE. Instantiation is used, for example, to create objects that contain simulation data. A necessary key feature of KEE (and object-oriented systems like Loops) is the possibility to dynamically define new classes while the program is running.

```
class vehicle {
  char*  owner;
};

class car : public vehicle {
  tyre   fl, fr, rl, rr;
  engine e;
};
```

**Listing 1.** Direct representation of a car model, derived from vehicle.

# 4. Model representation in C++

If interactive model development is presumed, direct representation is not possible in C++, simply because classes cannot be defined at runtime. Consequently, components cannot be represented directly with class members, and inheritance cannot be used to derive new models. To be able to interactively create models, we must implement a dynamic framework for representing models, realizations, etc. This framework is similar to the class systems commonly based on Lisp, but the implementation task is simplified by the structure of control systems.

It should be noted that the engineer developing control systems will see an interactive modelling tool; C++ is used only to implement the dynamic framework, not as a control system description language. One can also say that the object-oriented aspects of model representation have been separated from the object-oriented aspects of C++. Still, object-oriented programming effectively supports the design and implementation of the framework.

### Internal data structures

Now, let's plunge straight into the internal data structures of the C++ program. The code listed below is slightly simplified; constructors and destructors are not listed, and most general purpose routines have been omitted. An example will be given below.

All objects are components; they have a name, and they can be inserted into lists (Listing 2).

```
class component {
  char* name;
  link  next;

public:
  virtual void menuaction();
  virtual void redraw();
};
```

**Listing 2.** Definition of the basic component class.

Method `redraw` is a schoolbook virtual function in C++: every component has a graphical representation, so all components must implement `redraw` in some way. Graphics will be described further in Section 5.

When the user points at a component and presses a mouse button, some components (e. g., models and realizations) will respond by displaying a menu. Other components (e. g., terminals and connections) are not associated with a menu. In C++, which in its present shape only supports single inheritance, method `menuaction` must be declared as a

virtual function in the base class, component. When multiple inheritance becomes available in C++, menuaction would more naturally be the property of a class associated purely with the user interface; models and realizations would be derived from this class, but not terminals and connections [Stroustrup, 1987a].

Generally speaking, multiple inheritance enables us to separate the user interface and the modelling structure more effectively. There will be one "thread" of inheritance for the user interface (drawing block diagrams, and menu actions when applicable), and one thread of inheritance for the modelling of control systems (components, models, etc.). The development of class libraries, in particular, will benefit from multiple inheritance. For example, functions provided by the operating system and the window manager, will be easier to describe and use in an object-oriented fashion with multiple inheritance.

The model contains terminals and realizations, in C++ represented with linked lists (Listing 3). General purpose lists of components are used, which effectively corrupts the type security in C++. In addition, the programmer must bother about explicit type conversions. Alternatively, generic lists could be faked with macros. Future versions of C++ may incorporate true generics, also called parameterized types [Stroustrup, 1987b]. The need is evident, even in this small example.

```
class model : public component {
  list terminals;
  list realizations;

  void new_terminal();
  void new_realization();

public:
  void menuaction();
  void redraw();
};
```

**Listing 3.** Definition of the model class.

There are two different kinds of model realizations: primitive realizations based on equations, and structured realizations based on hierarchical block diagrams (Listing 5). There is no "one-of" concept (for example, allowing a pointer to a set of classes) in C++, so an additional class realization is needed (Listing 4). In this case, there are no real problems; in other cases, an awkward data structure might be forced upon the programmer. The one-of concept is available with full type checking in KEE, and has reduced the need for common base classes.

```
class realization : public component {
};
```

**Listing 4.** The common part of all realizations.

A submodel establishes a relation between two models, one fully enclosed in the other (Listing 6). With a structured realization, a model is described by the behaviour of its submodels and by its connections. The submodel also has a graphical meaning. When a model is simulated, the submodel must be "instantiated" by the model representation framework. Although many submodels may refer to a single model (e. g., a pump), every submodel requires a private data area to hold simulation variables.

```
class eqn_realization : public realization {
  list equations;

  void new_equation();

public:
  void menuaction();
  void redraw();
};

class struct_realization : public realization {
  list  submodels;
  list  connections;

  void new_submodel();
  void new_connection();

public:
  void menuaction();
  void redraw();
};
```

**Listing 5.** Primitive and structured model realizations.

```
class submodel : public component {
  point  position, size;
  model* parent;
  model* sub;
  void*  data;

public:
  void move();
  void scale();
  void instantiate();
  void redraw();
};
```

**Listing 6.** Definition of the submodel class.

### An example

A small example will demonstrate the data structures above: a servo built from a regulator and a motor. On the screen, the engineer will see a block diagram as in Figure 1. Input to the servo is the reference value, also called the setpoint. Output from the servo is the actual position of the actuator. The regulator controls the motor, but the common feedback loop has been left out to simplify the example.

The textual representation in Figure 2 reveals the most important C++ objects needed for the servo. The servo object has two terminals and a realization (terminals and connections will not be described in more detail). The realization is of course structured, and contains two submodels. It also contains three connections: the reference value imported to the regulator, the control signal from regulator to motor (shown in Figure 2), and the exported actuator position.

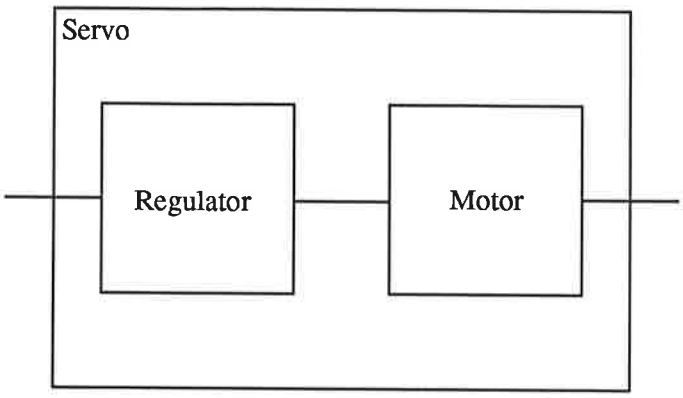


**Figure 1.** A servo with two submodels.

The submodel objects (for example, **MotorSub**) serve two purposes in this example. Firstly, the graphical appearance of a structured realization is determined mainly by the position and size of the submodels. This information cannot be stored in the model object; a certain kind of motor can be used as a submodel in many different models. Secondly, the submodels establish a relationship between the enclosing model (the servo), and the model objects used as components (e.g., the motor). The two pointers in the submodel object are used, for example, when defining connections. The references between models, realizations and submodels are shown graphically in Figure 3. The role of the submodel when simulating the control system is not discussed here.

The C++ objects used for representing the regulator and the motor are similar to the servo objects. The main difference is that the regulator and the motor have primitive realizations, probably expressed with differential equations.

### Exception handling

Handling of exceptions (errors and similar uncommon events) is a problem in all software systems. Ordinary programming techniques, using status flags and if-statements, lead either to bad program structure and cluttered code, or to programs that take proper behaviour for granted. A well designed exception handling mechanism (as in Ada), is an invaluable asset in practical software development. Exceptions increase the readability of the program and indicates the programmer's assumptions about expected and unexpected events [Ghezzi and Jazayeri, 1982, page 22].

The model development tool is quite complex, and many inconsistencies must be checked step-by-step, at different times. Exception handling is useful for restoring the internal data structures to a previous well-defined state. Storing as little redundant information as possible makes this task easier, but may increase complexity in other areas.

The absence of exception handling is a serious flaw of C++. Ada style exception handling, which is also available in C [Lee, 1983], is very effective, but a more flexible scheme may be called for in C++. Some people say that exception handling is needed for developing good class libraries.

Finally, it should be noted that friend functions have been used sparingly (for example, a connection needs free access to terminals and submodels), and proved to be extremely useful. By bending the rules a little, a natural data structure has been maintained; ever-expanding modules because of too strict encapsulation is often a problem with Modula-2 and Ada.

*Model:* **Servo**
*Terminals:* **[Ref, Pos]**
*Realizations:* **[ServoRealiz]**

*Struct-realization:* **ServoRealiz**
*Submodels:* **[RegSub, MotorSub]**
*Connections:* **[RegSub.u — MotorSub.u, ...]**

*Submodel:* **RegSub**
*Position:* $(-0.6, 0)$
*Size:* $(0.5, 0.5)$
*Parent:* →**Servo**
*Sub:* →**Regulator**

*Submodel:* **MotorSub**
*Position:* $(0.6, 0)$
*Size:* $(0.5, 0.5)$
*Parent:* →**Servo**
*Sub:* →**Motor**

*Model:* **Regulator**
*Terminals:* **[Ref, u]**
*Realizations:* **[RegRealiz]**

*Eqn-realization:* **RegRealiz**
*Equations:* $[\cdots]$

*Model:* **Motor**
*Terminals:* **[u, Pos]**
*Realizations:* **[MotorRealiz]**

*Eqn-realization:* **MotorRealiz**
*Equations:* $[\cdots]$

**Figure 2.** Textual representation of the servo; terminals, connections and equations are not shown. Square brackets denote a list, an arrow (→) a pointer reference.

## 5. Using PHIGS

PHIGS (Programmer's Hierarchical Interactive Graphics Standard) is a new 3D graphics standard, aimed at interactive CAE/CAD applications [Brown, 1985]. PHIGS should be regarded as an extension and a complement to the Graphical Kernel Standard [Hopgood et al., 1983], but not as a replacement.

The basic unit in PHIGS is the structure (cf. segment in GKS). A structure contains elements for drawing, graphical attributes, and transformations. It is possible to build hierarchies of structures (i. e., one structure may call another), and to edit the contents of a structure; this is not possible in GKS. Application data may also be stored in a structure, possibly a useful feature.

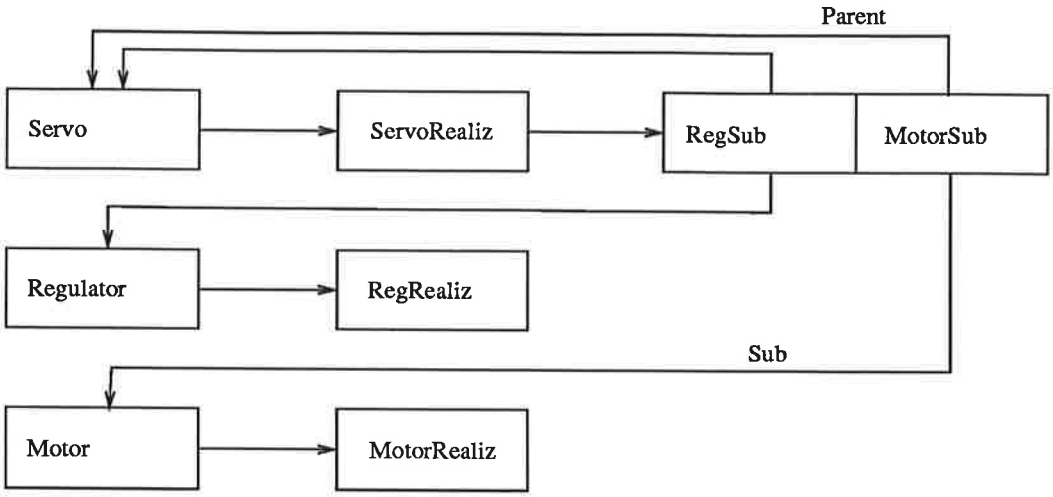


**Figure 3.** References between models, realizations and submodels of the servo. Terminals, connections and equations are not shown.

In order to take maximum advantage of the hierarchical structures in PHIGS, one structure is associated with every object in the C++ program. This one-to-one correspondence is very convenient; changes are normally localized to a single PHIGS structure, and complete regeneration of the graphics can be avoided. As a typical example, consider changing a pump model: the structure associated with the pump must be changed, but models using the pump as a submodel only refer to a structure identifier, and need no changes. The fine granularity of the graphics hierarchy causes an extra overhead at redraw, which is quite tolerable in this application, though. It can be noted that the model development tool is not a typical PHIGS application, in the sense that it uses the hierarchical features of PHIGS, but not the 3D capabilities.

The correspondence between the object hierarchy and the PHIGS structure hierarchy is shown in Figure 4. The object structure on the left is the same as in Figure 3, but the regulator objects are not shown. A PHIGS structure is associated with each object, as indicated by dashed arrows. The PHIGS structures on the right form a parallel hierarchy, logically connected with "execute structure" primitives. The graphical representation of a model is determined by the realization and its associated structure. The PHIGS structures are in reality more complex, for example, to control picking (see below).

The problem of associating a C++ object with a structure, was solved by some fancy programming. A C++ object can easily refer to a structure by storing the structure identifier, but a problem arises when control must go from a structure to the associated C++ object (for example, when the object's menu action should be invoked). The solution is to use the object's `this` pointer as pick identifier, after conversion to an integer. When the PHIGS system returns a pick identifier, the identifier is converted back to a "pointer to component." The exact nature of the object is not known, but all components implement method `menuaction` (Listing 2).

PHIGS can display graphics on multiple "workstations," which in a workstation environment corresponds to multiple windows. By using so called filters, different graphical representations can be displayed with a single structure hierarchy. Regrettably, multiple workstations are not yet supported by some PHIGS implementations. Event mode input and rubberband lines may also be missing in current implementations. Window management is not available in the PHIGS standard, and may therefore cause considerable practical problems.

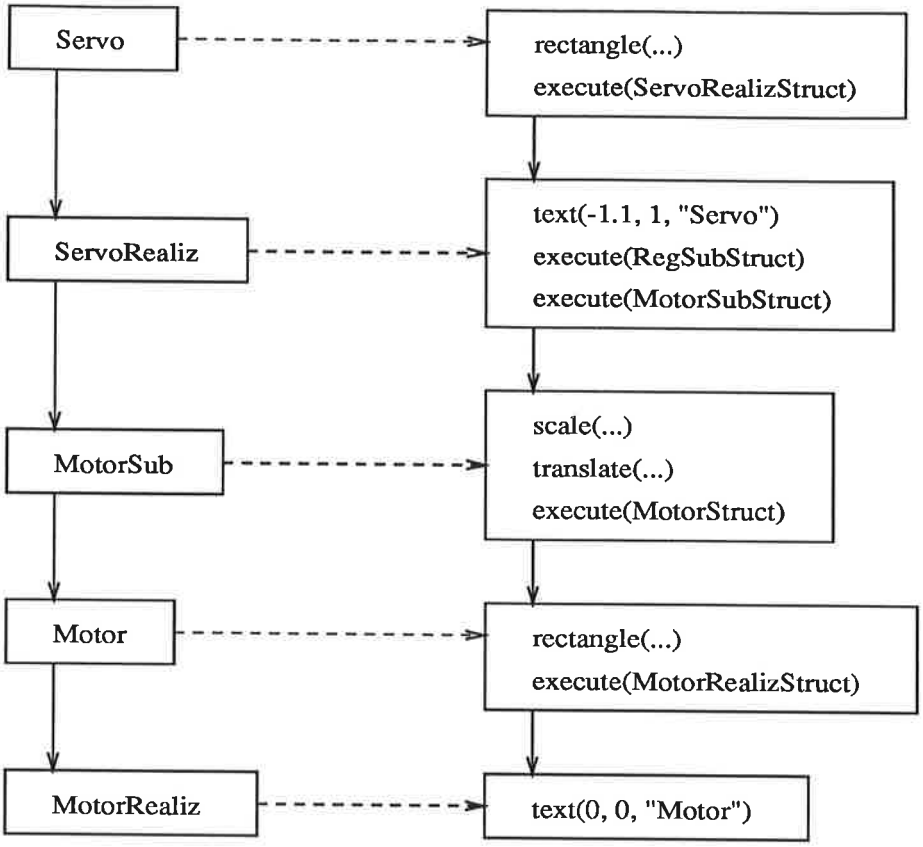


**Figure 4.** Parallel hierarchies of C++ objects (left) and PHIGS structures (right).

## 6. Conclusions

In our experience, a dynamic environment like KEE is the best choice for research and rapid prototyping. An engineering tool requires a less expensive and more efficient implementation tool that is available on many computers; in this case, C++ is superior. We have not made a detailed evaluation of KEE versus C++, but the current work shows that programs and data structures using the object-oriented parts of KEE can be implemented in C++ with reasonable effort.

The major difficulty is that C++ does not support dynamic creation of classes. For this reason, models of control systems cannot be directly expressed as classes in C++, so an object-oriented framework must be implemented. The data abstraction and object-oriented programming aspects of C++ provide good support for this framework, and a good programming environment in general. Multiple inheritance, parameterized types and exception handling are much needed extensions to C++.

PHIGS is a powerful new graphics standard, but current implementations need improvement. Window management remains a problem area.

## Acknowledgements

I am grateful for many interesting discussions with Sven Erik Mattsson and Mats Andersson, and for comments on the manuscript by Mats Andersson, Ola Dahl and the reviewers. This work was supported by the Swedish National Board for Technical Development (STU).

## References


BROWN, MAXINE D. (1985): *Understanding PHIGS*, Template Graphics, San Diego, CA, USA.

GHEZZI, CARLO and MEHDI JAZAYERI (1982): *Programming Language Concepts*, John Wiley & Sons.

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

LEE, P. A. (1983): "Exception Handling in C Programs," *Software — Practice and Experience*, **13**, 389–405, May 1983.

MATTSSON, SVEN ERIK (1988a): "On Model Structuring Concepts," *Proc. 4th IFAC Symposium on Computer-Aided Design in Control Systems*, Beijing, P. R. China.

MATTSSON, SVEN ERIK (1988b): "On Modelling and Differential/Algebraic Systems," *Simulation*, Accepted for publication.

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

STROUSTRUP, BJARNE (1987a): "The Evolution of C++: 1985 to 1987," *Proc. USENIX C++ Workshop*, Santa Fe, NM, USA.

STROUSTRUP, BJARNE (1987b): "Possible Directions for C++," *Proc. USENIX C++ Workshop*, Santa Fe, NM, USA.