Application of Object-oriented Methods in Process Visualisation

L. Szirmay-Kalos, G. Márton, T. Fóris, J. Fábián

email: szirmay@fsz.bme.hu

Department of Process Control, Technical University of Budapest

Abstract

This paper presents an approach to building process visualisation systems. A visualisation system must update a model of the visualised process from the measurements and provide its graphics presentation to the user. The proposed method defines both the model and its graphics presentation simultaneously and using graphics techniques. The only step which requires programming is the description of the behaviour of component types. Using object-oriented approach, the behaviour is described by rules determining how an object reacts to messages from the process, other objects and from the user. From this definition and from the structure derived from the graphics schemes, the global system behaviour is generated automatically. This allows for fast visualisation system generation and requires only graphics editing if the underlying system structure changes.

Keywords: process visualisation, object-oriented design, model-view-controller paradigm.

1. Introduction

Process visualisation means the monitoring of the current state of complex industrial processes such as electric power supply systems, train and flight traffic, chemical reactors, etc. Process visualisation is not simply the display of measured data since usually only a subset of variables are measured in these systems, while other variables must be computed using the measured data and some knowledge of the visualised system.

In an electric system, for example, the voltage is measured at the output of the transformer station, while the actual state of the power transmission lines, describing whether or not they are energised, must be calculated by the process visualisation program from the measured data including the output voltage of the transformer and the actual state of the switches.

This means that a process visualisation program must have a *model* representing the structure of the system and the actual state of the components. Then this model is visualised on the computer screen and allows the user to intervene in the process operation. The graphic representation of the actual state of the model, called the *view of the model,* usually consists of one or more screen structures, called *schemes*, and dialog windows popped up onto the schemes. Should the user want to give control commands, he can interact with the controller elements of the graphics representation. This separation of the model and its view is usually called the *model-view-controller paradigm* [2].

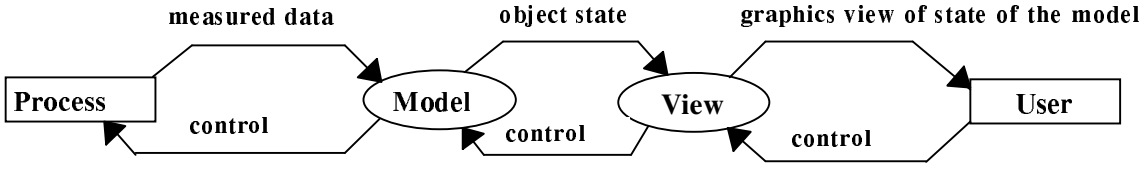

Figure 1: Relation of process, model, view and user

If the model could be defined by a set of differential equations or other well defined mathematical formalism, the system would be called a *scientific visualisation* system [5].

Models of general process visualisation systems, however, are not restricted to mathematical structures, but a model can be anything that can be simulated on a computer.

A visualisation development system should provide methods to define both the model and its view. In addition to the initial definition, the modification of the model and the view is also required, since the structure of real systems is usually not static, but is modified regularly by construction of new subsystems or by tearing down the obsolete parts. These tools should be easy to use and should not allow *inconsistent modification* of the model or its view, nor should they require deep knowledge of programming or of the design of the visualisation program. The design of the view is straightforward and usually accomplished by interactive and graphical methods in the available process visualisation systems. The model, on the other hand, must be specified by an application program that is connected to the interactively designed view. This approach, however, does not allow for the easy definition of the model nor is it capable to guarantee the consistent modification of the model and its views.

Obviously, a method is needed that uses interactive and graphics techniques for both the view and the model and which guarantees the consistency between the two aspects. Since the schemes of the view provide a graphical representation of the system structure, an approach meeting these requirements would be graphics editing of these schemes and the automatic update of the model from these new schemes. Since only the view is modified while the model is automatically generated from the view, the consistency between the model and the view is guaranteed. The modification requires only the use of a "drawing package", thus it can be done by non-programmers too.

Although this idea looks promising, its implementation raises the following problems:

1. The drawing package used to draw the schemes should allow the definition of both the displayed components and their interconnections (topological structure).
2. The model is not only the structure of its components but also implies an "algorithm" which describes the behaviour of the system. This behaviour would determine the not-measured data from the measurements, from the structure and from the behaviour of the components. Behaviour is also responsible for the reaction to the user inputs and their corresponding changes in the model and interventions in the process.

The solution of the first problem requires a special *scheme editor* where the topology is also defined in addition to the screen layout (geometry). The scheme editor connects the model objects to graphics symbols that are responsible for the display of the state of those objects. This solution also requires transformation tools that convert the topological data of the schemes to the structure of the model. Since in this method the scheme is the only place where objects can be defined (scheme editor is rather a model definition tool than simply a view definition program), scheme editors should make it possible to hide those objects which take part in the model, but are not seen on any scheme.

The second problem challenges the methods of object-oriented software engineering, which makes possible to describe the operation of complex systems by individual actors (objects) having their own state, identity and behaviour and communicating with each-other to generate the global behaviour of the system.

In this paper the principles of an object-oriented process visualisation system applying these concepts are discussed. In order to define the specification model of the various parts of the program, the analysis models and notation proposed by Rumbaugh et al. [1] are used. The examples are taken from the field of electric power supply (systems) and train traffic control systems. The applicability of the concepts, however, is not limited to these application areas.

2. Object-oriented definition of process visualisation

It is obvious that the behaviour of a system can be made adaptive to the dynamically evolving system structure if the behaviour is defined on the component level. This means that the operation of individual component types (switch, transmission line, track, instrument, etc.) must be defined without knowing anything about the global system structure. This guarantees the validity of the behaviour in any structure.

The behaviour of a single component type means the description how a component of this type reacts to events or messages from its environment. The environment of a single component consists of other components, the underlying process in the case of measurements and the user himself in the case of user controls. The reaction of a component may be the modification of its internal state and it may send messages to other objects or directly to the controlled process. In this way, the global behaviour of the system can be defined by the behaviour of individual objects and their communication. The structure of the system is defined by the communication paths, that is by the information describing which objects can communicate with each-other. Since the structure is dynamic and the same type of components can be used in many different points of the system, the definition of the behaviour should not contain the communication partners. Instead, so called *connection points* are defined for each object type and the communication is defined by sending messages to or receiving messages from these connection points. The actual partner of communication would be that object which is connected to the "other end" of the connection point.

The state of an object holds the information needed to determine the effect of future messages. From the programmers point of view, state is decomposed into variables. In order to assure flexibility, these variables are not limited to the built-in types of a programming language. In the actual implementation, the variables have dynamic type, that is, the type of a variable is determined by the value which was assigned to it. The value can have the following types:

- **digital**, if the value has finite domain (enumeration type),
- **analog**, if the domain is a subset of real numbers,
- **text**, if it is a (non-limited) array of characters,
- **pointer** that can hold the address of an object,
- **FIFO, List**, etc. of those types.

Concerning practical situations a "real object type" has just a few connection points and its behaviour can be summarised by a couple of rules. An electric wire, for example, has two connection points. The rule describing its operation is the following: if one end (connection point) is energised (the object connected to this point has sent a message that this point is energised), then the other end is also energised, which requires a message to be sent to the object connected to it. If the actual voltage is irrelevant and we need to know only whether or not the wire is energised, then the wire will have a single digital variable describing if it is energised or not. An electric switch, on the other hand, propagates the "energised" message from one of its connection points to the other connection point only if the switch is closed. Thus the internal state of a switch consists of a digital variable showing whether the switch is closed or open, and two other variables for the (two) endpoints describing if they are energised. Let us take other examples from the application area of train traffic control. A track has two connection points. Concerning the behaviour of the track, it simply passes the train arrived at one of its connection points to the other. The track has a single variable of type FIFO of text containing the identifications of the trains travelling on it. A (track) switch has three connections, and the point where the train leaves the switch depends on the state of the switch (a single digital variable) as well as the point where the train arrived at the switch.

Now let us examine the view part of process visualisation. The view must reflect the actual state of the system, that is the actual state of the objects of the system. For each model object, *dialog-windows* and display objects called *symbols* can be defined. Dialog-windows provide detailed, usually textual information about an object and its variables. Symbols are organised into screens called *schemes*. A scheme thus visualises a greater part of the system. A symbol reflects just the most relevant properties of the actual state of an object or optionally of a variable of an object.

Symbols have different subtypes, including, among others:

- *Colour-symbol* that is capable of visualising a value of finite domain. It consists of graphics primitives including polyline, polygon, rectangle, ellipse, text, etc. For each primitive, the colour must be defined for each value. By using ”non-visible” colour we can create symbols modifying their shape due to the change of the displayed value.
- *Text-symbol* which can display text or any other values converted to text.
- *Widget-symbol* is a widget of the underlying Window system (currently OSF/Motif). Its capabilities are determined by the type of the widget.

Although a symbol type can only display a few value types, this does not introduce any restriction into the system, since the variables are of dynamic type and their conversion is automatic. It means that an object's variable that has currently digital type can also be displayed by a text-symbol, since having received the variable the text-symbol converts its value to text without examining its original type. For a digital value, the result will be the logical name of the value (e.g. "switch open").

The connection between the model and the view is realised by associations of symbols and dialogs with the objects of the model. An object can have dialogs and symbols located on different schemes. Each symbol corresponds to exactly one object, while some of the symbols are directly connected to a single variable of that object. For these symbols, the update of their graphics appearance is automatic. If the value of the variable changes, the new value will update the symbol automatically. The new look of the symbol will depend on the type of the symbol (Colour, Text or Widget) and its definition including the graphics primitives and their state space.

The model, view and their connection are shown in the following OMT object-diagram [1]:

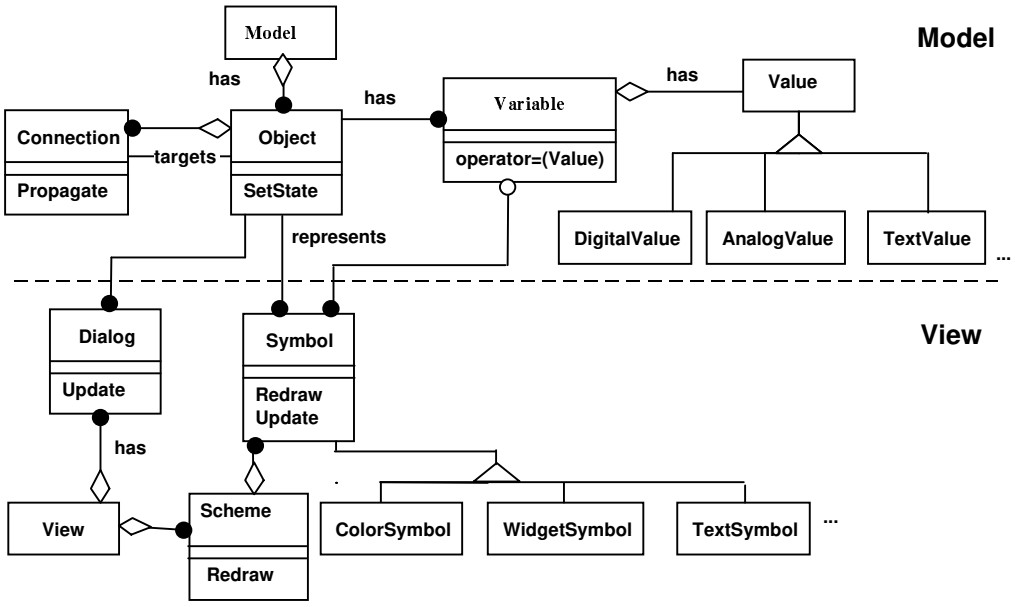

Figure 2: Object model of the visualisation program

The object model provides a static view of the visualisation system. Concerning its dynamic behaviour, we must consider how the objects react to events coming from the outside world. In the simplest case, these external events can come from the process informing the model about a new measurement and from the user. The communication sequence generated by an event from the process is summarised in the following communication diagram:

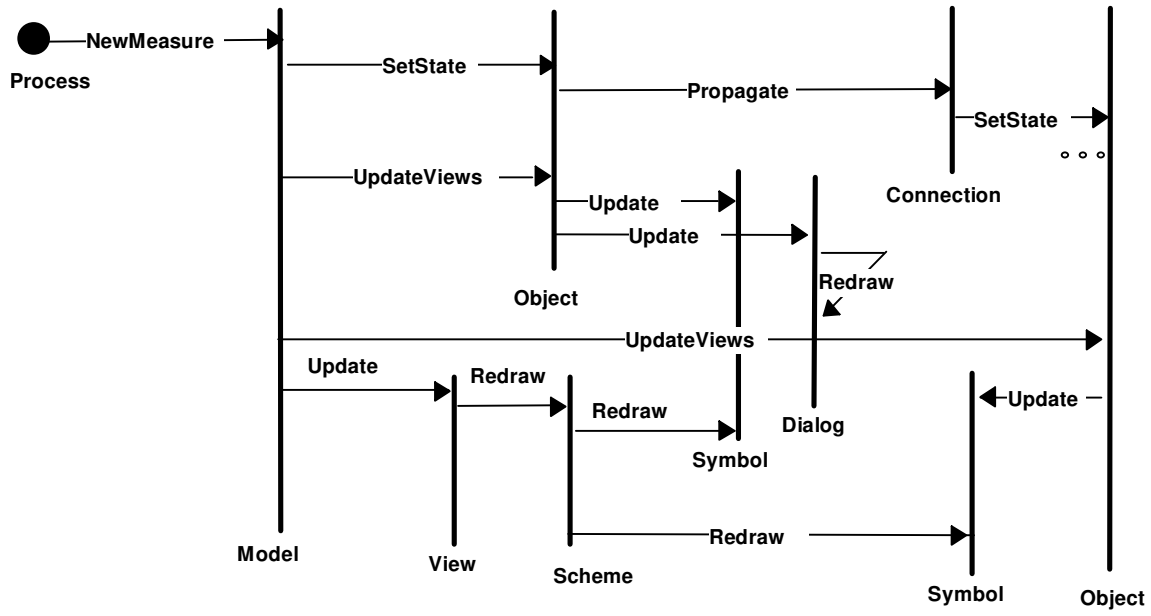

Figure 3: Communication diagram describing the reaction to a new measurement from the process

3. Application definition

The visualisation system discussed so far provides a general framework to operate the model and its view to reflect the operation of the monitored process. The operation of model objects as well as the operation of symbols and schemes, however, depend on the actual application domain which must be defined to complete the visualisation program. This section introduces the concepts of application definition. Again, the application definition should heavily rely on graphics representation. The main flow of application definition is shown in figure 4.

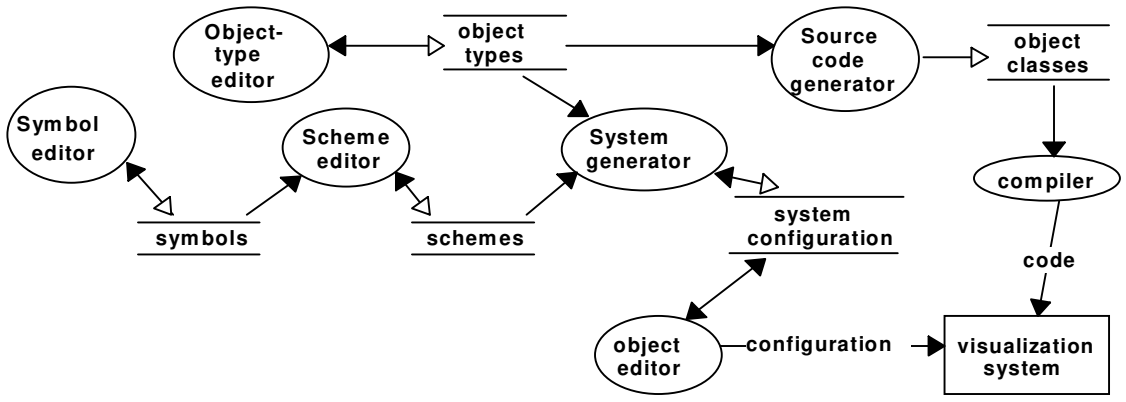

Figure 4: Dataflow of application definition

Symbol editor is a graphics tool for defining *colour-symbols*. In order to reflect the practical experience that systems usually consist of components of fixed size and having fixed number of connections (resistor, instrument, valve, building, etc.) and components used to interconnect the fixed components (wire, pipe, track, road, etc.), two types of colour-symbols are distinguished: *fixed-symbol* and *link-symbol*.

To define a fixed-symbol, first the graphics primitives forming this symbol must be drawn. Since a colour-symbol will reflect the state of a model object by modifying the colour of its graphics primitives, the designer must also define the states of the symbol and order colours to the primitives in each state. Finally the connection points (which reflect the interface points of the real object visualised by this symbol) must be specified.

Link-symbols, on the other hand, have no fixed geometry, but it will be determined by those fixed-symbols that are connected by this link-symbol and, of course, by the "routing" possibilities between those fixed-symbols. Thus the definition of a link-symbol includes only the definition of the state space and in each state the graphics attributes (e.g. colour) of the polyline representing this link-symbol.

The object model describing the symbols created by the symbol editor is shown in figure 5.

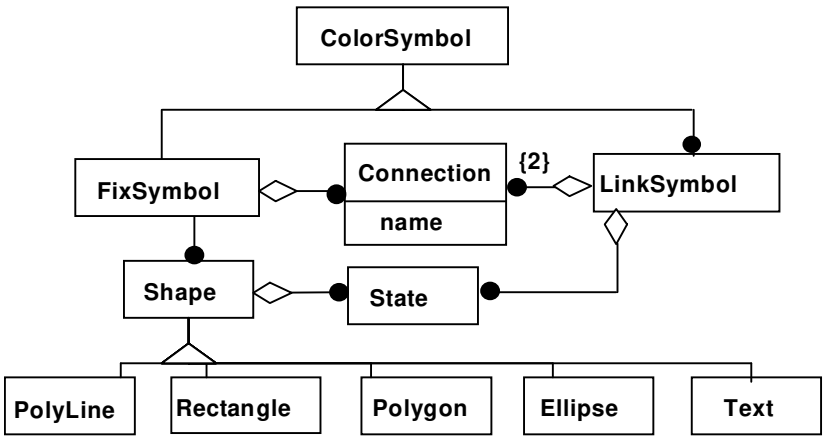

Figure 5: Object model of symbols in the symbol editor

In practical symbol design the accepted symbols of engineering drawings are usually used, which are animated according to the value of the represented value.

Having defined the symbols, these can serve as components to build up screens with the *scheme editor*. *Scheme editor* requires not only the drawing of the screen layout, but the topological structure of the symbols and the correspondence between symbols and model objects. To derive the topological structure, the *scheme editor* takes advantage of the distinction between fixed- and link-symbols. Fixed-symbols can be placed anywhere on the scheme, while the placement of a link-symbol requires the identification of two connection points of two fixed-symbols that will be connected by this link-symbol. In addition to the defined starting and ending points, link-symbols can have unlimited number of internal points which determine the path how the link-symbol connects the two fixed ones.

Scheme editor also allows the placement of *text-symbols* and *widget-symbols* onto the screen.

When a symbol is placed, regardless of its type, the name and type of the object that will be represented by this symbol must also be specified. Optionally, a variable of that object can be identified. Thus the correspondence between the view (symbols of the scheme) and the model (name of model objects and optionally variables of model objects) is provided here.

Summarising, the object model of the structure built-up by the scheme editor is:

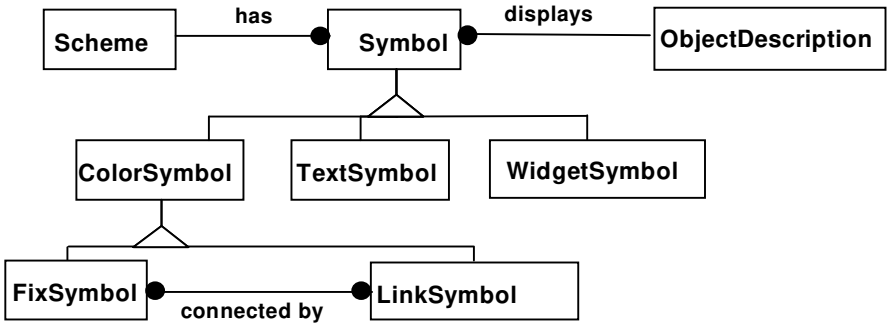

Figure 6: Object model of schemes in the scheme editor

Schemes are usually drawn by examining the design drawings of the system. First fixed- symbols are placed, then connected by link-symbols. Finally, invisible objects are identified. Invisible objects are those objects that cannot be seen in any schemes. However, it is worth defining them in this stage of the design because they take part of the model and their role, place and topological interconnections can be unambiguously defined by the scheme editor.

The topology oriented approach of the *scheme editor* has two advantages. The scheme can be edited easily and safely, because dragging a fixed symbol will automatically drag the connected link symbols. On the other hand, the model can automatically be generated from the graphical representation of the schemes, provided that the scheme meets the requirement that two fix-symbols are connected by a link-symbol if and only if their represented objects are also connected by the object represented by the link symbol. The system generator will derive a list of objects that were given as base objects of the symbols and a connection list derived from the connections of fixed and link symbols. This approach does not require parallel, and therefore dangerous editing of view and model representations. Only the view must be restructured if the underlying system structure changes, the model will automatically be updated.

The automatically generated object list must be extended with the individual object properties, including the physical address (if the given variable is measured), technological and emergency limits of analog values, etc. The representation generated by the *system generator* and extended by the designer serves as the *configuration* of the visualisation program.

The other branch of application definition focuses on the definition of *model object types* (Fig. 4). Using the *object type editor*, the application developer defines the variables and connection names of individual object types. For example, the electric switch type may be defined by the following statements:

```
TYPE Switch
VARIABLE state DIGITAL VALUES switch_on, switch_off
VARIABLE voltage1, voltage2 DIGITAL VALUES non_defined, energised, ground
CONNECTS end1, end2
```

Listing 1: Type description of an electric switch

From this description the source code generator creates a C++ class with default behaviour to react to process and user events. This C++ class must be altered by the application programmer to reflect the specific properties of the object class. A strongly simplified version of the file generated form the above definition and completed to reflect the behaviour of the switch, can be the following (statements that are not automatically generated are shown with boldface):

```
//*********************************************************
//                          SWITCH                        *
//*********************************************************
enum  Switch_connections { end1, end2 };              // CONNECTIONS

//          VARIABLES
#define state    variables[0]
enum  state_values { switch_on, switch_off };
#define voltage1 variables[1]
#define voltage2 variables[2]
enum voltage1_values { non_defined, energized, ground };

//=========================================================
class Switch : public ActiveObject {
//=========================================================
public:
    Switch( ) : ActiveObject( "Switch" ) {       // default object name
        state = DigitalValue( switch_on );
        voltage1 = DigitalValue( non_defined );
        voltage2 = DigitalValue( non_defined );
    }
    Response SetState(Connect src, Message ms, Variable* pv, Value* value);
};

//-----------------------------------------
// Function responsible for object behaviour
//-----------------------------------------
Response Switch :: SetState(  Connect source,      // message source
                              Message mess,        // message identifier
                              Variable * pvar,     // destination variable
                              Value * value ) {    // new value
//-----------------------------------------
    switch ( source ) {
      case PROCESS:           // Message from the process (measurements)
          switch ( mess ) {
            case M_SWITCH_CLOSED:
                state = switch_on;
                switch ( voltage1 ) {
                  case energized:    Propagate(end2, M_ENERGIZED);  break;
                  case ground:       Propagate(end2, M_GROUND);     break;
                  case non_defined:  Propagate(end2, M_NOT_DEFINED); break;
                }
                switch ( voltage2 ) {
                  case energized:    Propagate(end1, M_ENERGIZED);  break;
                  case ground:       Propagate(end1, M_GROUND );    break;
```

```
        case non_defined:  Propagate(end1, M_NOT_DEFINED); break;
      }
      return OK_RESPONSE;
  case M_SWITCH_OPEN:
      state = switch_off;
      Propagate( end1, M_NOT_DEFINED );
      Propagate( end2, M_NOT_DEFINED );
      return OK_RESPONSE;
}
```

```
      case end1:              // Message from "end1" connection points
          switch ( mess ) {
            case M_ENERGIZED:   voltage1 = energized;   break;
            case M_GROUND:      voltage1 = ground;      break;
            case M_NOT_DEFINED: voltage1 = non_defined; break;
          }
          if ( state == switch_on ) Propagate( end2, mess );
          return OK_RESPONSE;
      case end2:              // Message from "end2" connection points
          switch ( mess ) {
            case M_ENERGIZED:  voltage2 = energized;   break;
            case M_GROUND:     voltage2 = ground;      break;
            case M_NOT_DEFINED: voltage2 = non_defined; break;
          }
          if ( state == switch_on ) Propagate( end1, mess);
          return OK_RESPONSE;
      default: return INVALID_RESPONSE;
    }
}
```

Listing 2: Class definition for the switch object

As shown in the above example, a member function, called *SetState*, must be filled up to describe how an object of this type reacts to messages coming from the user, process or from the connection points. The reaction may involve the update of the internal variables and sending new messages to other objects through the connection points. The update of symbols is automatic, thus this burden is taken from the shoulder of the programmer. The development system compiles this class and builds it into the visualisation program which can be started with the *configuration* defining objects and their connections.

A characteristic view of the running visualisation program is shown in figure 7.

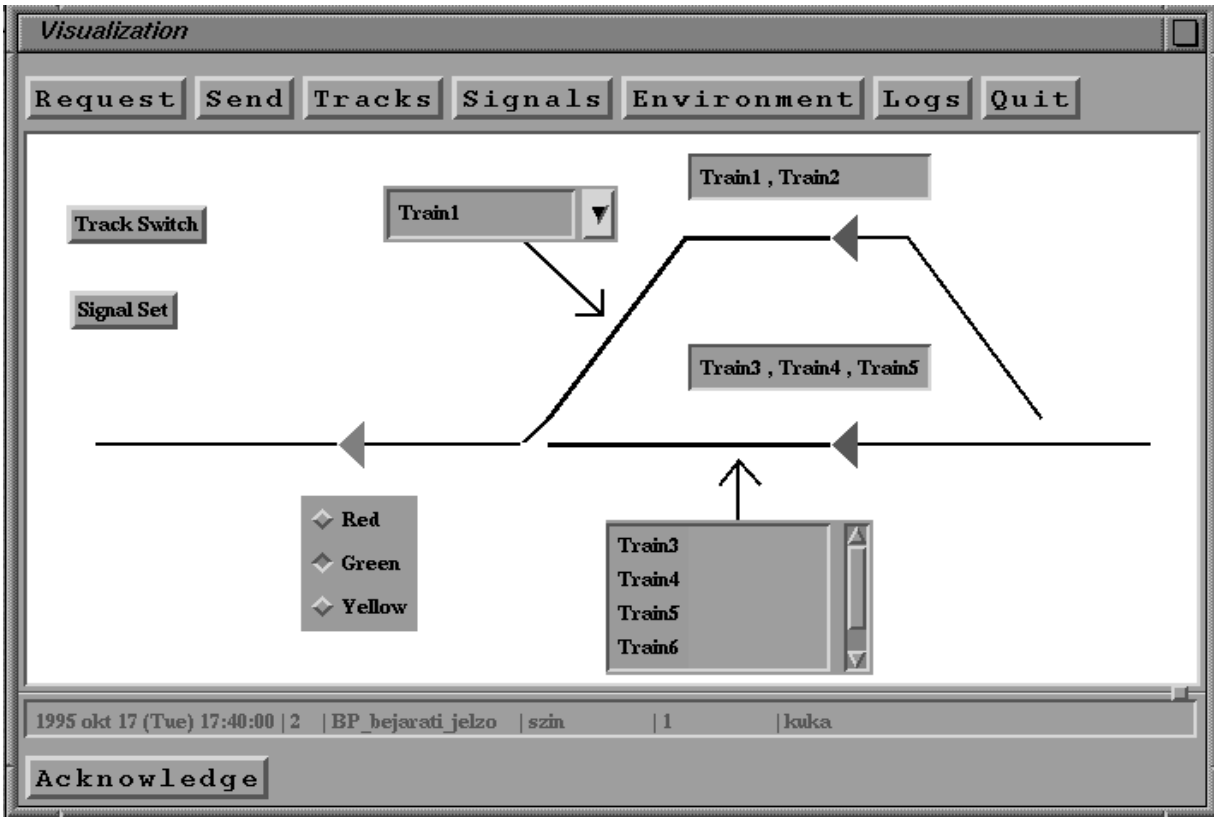

Figure 7: Snapshot of the visualisation program

4. Conclusions

This paper proposed a method for process visualisation taking advantage of object oriented methodology. In order to develop an application, the graphics representation of the symbols must be drawn, the screen layouts representing the topological structure and the underlying model objects must be provided and the behaviour of each object type must be programmed. The behaviour is defined by rules reflecting how the object reacts to messages coming from the physical process, the user or from its connection points connected to other objects. Since this behaviour definition is independent of the environment of the object, it is valid in any system structure and will remain valid if the structure is altered. According to our experience, even complex systems are composed of just a few object types (e.g. an electric lighting network may have the following components: supply, switch, lamp, wire, wire-connection, grounding). The definition of their behaviour requires just a couple of C++ statements. Thus the only problem which requires significant effort is the scheme definition. If the schemes change, but no new object types appear, the modification of the model requires no programming at all, thus the visualisation program can be quickly and easily adapted to evolving systems.

In fact, the definition of object operation as a response to messages coming from its connection points follows the natural behaviour. The modelling power of this approach is very promising. We have tried it in the application areas of train traffic control, electric power distribution, lighting networks, logic circuits and linear electronic networks consisting of resistors and switches. The last example is a challenging one, since unlike the previous examples, in resistor networks there is a very strong coupling between the state (voltage and current) of different objects. Thus, if the voltage of one connection point of a resistor changes, the components should start a very intensive communication to decide the value of the voltage at the other connection point. These problems related to strong coupling can be resolved by iteration. The voltage of the other endpoint is estimated from the variables of only those objects which are directly connected to this point. Then a message is sent to the other endpoints to do the same. This iteration will converge to the real solution. This is similar to the progressive radiosity method of computer graphics.

5. Acknowledgement

This work has been supported by the Hungarian Scientific Research Fund (OTKA) under the reference number F 015884.

6. References:

[1] Rumbaugh, Blaha, Premerlani, Eddy, Lorensen: Object-oriented Modeling and Design
Prentice-Hall, 1991

[2] W. Lalonde, J. Pugh: Inside Smalltalk (Volume II)
Prentice Hall, 1990

[3] N. Knolle: Why Object-oriented User Interface Toolkits are better

Journal of Object-oriented Programming, Vol. 2, 1989
[4] B. Shneiderman: Designing the User Interface
Reading Mass., Addison Wesley, 1986
[5] R.A. Earnshaw, N. Wiseman: An Introductory Guide to Scientific Visualization
Springer-Verlag, 1992