Object-Oriented Framework and Methodology to Process Visualisation System Development

László Szirmay-Kalos
e-mail: szirmay@fsz.bme.hu
Department of Control Engineering and Information Technology,
Technical University of Budapest

Abstract

In this paper a process visualisation development system and its associated development methodology are presented. This methodology is optimised to systems that have complex structure and are built of large number of components belonging to relatively small number of types. In order to handle the complexity, the input requirements of the method is as close to the ”native language” of the application as possible. The elements of the ”native language” are assumed to include engineering drawings and manuals describing the operation of component types the system is built of. Graphics techniques are used to supply the engineering drawings into the development system while not only the required visual appearance is described but the structure of the underlying system is also defined. The elements of engineering drawings are dynamized to animate the graphics presentation to reflect the current state of the monitored system. Component manuals are transformed to interface and state definitions from which a code generator generates a C++ class for each component type. This C++ class must be tuned to reflect the operation of a single component type. From these definitions the development system automatically builds up the complete visualisation program, providing easy and fast application development.

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

Introduction

Process visualisation systems are used to monitor the operation of industrial processes and to provide a graphics representation of the current state to the user. Furthermore, they also allow the user to intervene in the process operation. A process visualisation system is informed about the changes of the underlying process by messages encoding events or results of measurements. In order to derive the current state of the process from these changes, a process visualisation system incorporates a model of the monitored system. The objects of the model correspond to physical or logical components of the monitored system. The level of abstraction on which components are defined is determined by the level of detail the visualisation system must provide. If all important variables of physical components were measured and the results were sent to the related object, then the current state of the process - that is the current state of every component - could be determined easily without requiring the model objects to communicate with each other. In real systems, however, the majority of components are not connected directly to measuring instruments, thus the state of their model objects must be calculated from the state of those model objects which are updated by messages from the process. This is possible if the model incorporates not only the objects reflecting components, but also the structure of the underlying system and ”knows” how the process operates. This means that the model is expected to simulate the operation of the real system.

In an electric supply system, for example, the voltage is measured at the outputs of the transformers, from which the voltage of all transmission lines should be calculated and presented to the user. This can be done if the topology of the system is known and the

visualisation program is aware of the rules of operation: a transmission line is equipotential and the two endpoints of a switch have the same voltage if the switch is closed.

In addition to simulating the real process to keep the complete model updated, a process visualisation system should also provide graphics presentation to the user and interact with him.

For complex processes, both the model management and the design of the graphics presentation requires a great amount of programming effort. Since process visualisation systems are demanded by all sectors of industry, the need to support the development of such software products created a family of tools called *Process Visualisation Development Systems*, including, for example, Fix DMACS [1], Vision [2], Visual Designer [4], PVSS [5], Sammi[6] and SL-GMS [7].

General purpose process visualisation systems have to provide solutions for two different problems. On the one hand, they have to support the definition of the model and its interfacing to the process. On the other hand, they should provide the features of a user interface development package and help interconnect the generated user interface with the model. Systems that support only the user interface definition (e.g. DataViews [3], IlogViews [8]) can also be used to develop process visualisation, but the developer has to add all the functionality of the model in a separate code.

Current process visualisation development systems provide the developer with WYSWYG style editors to define the user interface interactively and to specify how the graphics is animated. The generated user interface consists of widgets and dynamic graphics elements. In some visualisation development systems the set of available user interface elements is fixed, while in others this set is extensible. The behaviour of user interface objects is controlled by setting the properties or by adding program segments in triggers.

Model management is solved through the use of so called databases where model objects are stored, and the records and fields from these databases are visualised. The behaviour of the model and the co-operation between the model and the view should be defined by a separate program written in a ordinary or graphical programming language. The applied graphical programming languages describe the functionality as dataflow or eventflow diagrams that are usually built up from fixed transformation elements. This approach is very popular in *scientific visualisation* systems [8] where the model could be defined by a set of differential equations or other well defined mathematical formalism. For process visualisation, however, where a model can practically be anything that can be simulated on a computer, a restricted set of building blocks might make the model definition cumbersome. Therefore, advanced development systems allow user defined transformation blocks.

We can conclude that for the user interface part, process visualisation development systems are clearly object oriented, but for the model part, they either do not provide any framework or force the developer to follow a functional approach.

In order to evaluate a process visualisation development system and its associated development methodology, the properties of the process to be visualised must also be taken into consideration. This paper focuses on a large and important family of physical systems and processes described by the following properties:

1. The physical system and its the required graphics presentation are very complex, where the number of model objects and view elements can be thousands or even tens of thousands.

2. The structure of the system is also complex. Furthermore, the components and the structure of the real processes often vary even after the installation of the visualisation system due to reconstruction.
3. The number of component types from which the physical system is built of, however, is usually small (tens). The operation of a single component cannot be modelled by mathematical structures, such as differential equations, real functions, digital networks, etc. Instead, the operation is defined by rules describing how the component reacts to changes on its inputs. Unlike the structure of the system, component types do not tend to change during the life cycle of the physical system.

These features are common in the majority of the physical systems, such as in electric or water supply; drains; train, car or flight traffic; analogue or digital networks; telecommunication systems; etc. All of these systems are very complex, since they are built of many components interconnected in complex ways, although they incorporate just a few component types of relatively simple operation. The complex structure of these systems results in very complicated global behaviour that is impossible to describe by formal rules. The component level behaviour, however, can be defined easily.

If we used commercial process visualisation development packages to visualise these systems, complexity would pose severe problems. Commercial development packages usually provide sophisticated techniques to define and modify the user interface, but they do not really support the easy definition and modification of complex models. Their model definition ”language” may be very far from the natural concepts of the system to be visualised. The functional approach offered by these systems for the description of the behaviour of the model require the structure of the model to be ”wired” into the code, which makes it very complicated to maintain for complex systems. Thus the definition and validation of the model and global behaviour of such systems could be a nightmare, and it might be necessary to start the definition from scratch if the structure of the system changes.

The basic requirements of a visualisation development system that can overcome this problem of model complexity are summarised, as follows:

1. The input requirements of the development system should be close to what is usually available about the underlying system or process, thus development does not require intensive transformation of complex system descriptions and long design cycles. The underlying system is assumed to be defined by:
    - Engineering drawings based on the symbols and the notations used in the given branch of industry.
    - Manuals of component types used in the process. These manuals contain information about the interface or connection points of the component and its operation. For simple components, such as pipes, switches, lamps, etc., no such manual is necessary.
2. The model objects and view elements should be generated safely and easily, and should not allow *inconsistent modification* of the model or its view. The safe generation is further emphasised by the observation that the components and the structure of real processes often vary even after the introduction of the visualisation system due to reconstruction. The visualisation system must be maintained on these occasions, which is a potential source of errors. Safe and easy definition can be provided by graphics techniques. From the graphics representation, model objects and view elements must be generated automatically and

without programming. Since component types do not tend to change during the life cycle of the underlying process and their number is relatively small, programming is allowed on the level of component types, but it must be supported by automatic code generation.

3. Since the complicated global behaviour of systems under consideration is a result of great number of components and complicated structure, the functional properties should be defined on the component level, thus the definition of the behaviour is separated from the definition of objects and the structure. This makes the development clearly object-oriented. The separation of the behaviour from the structure also makes the visualisation system adaptive to evolving system structures. If the structure changes, the program code need not to be altered.
4. The required programming effort should be minimised and the user of the development system should not be expected to posses deep knowledge of advanced software engineering and user interface methods. For user interface design no coding should be required. However, for the sake of flexibility, the available user interface elements should also be extended by user developed ones.
5. The generated visualisation program is optimised for speed, since process visualisation systems are real-time systems where the response time is a key factor.

Using a development system meeting these requirements, we can start with engineering drawings describing the process to be visualised and with manuals of component types. Applying only graphics techniques, engineering drawings are supplied into the visualisation development system while making design decisions on the graphics interface. Design decisions concern how the engineering drawings are imitated by graphics windows and how these drawings are made dynamic to reflect the actual state of the process. Note that engineering drawings affect not only the layout and the elements of the user interface, but also determine the objects and the structure of the model. In order to preserve this information, engineering drawings are input to the system by a special *scheme editor* where the topology is also defined in addition to the screen layout (geometry). The scheme editor also requires the identification which model object is represented by a graphics element on screen. This solution also requires transformation tools that convert the topological data of the schemes to the structure of the model. Manuals of component types, on the other hand, are converted to the description of the interface (connection points) and of the internal state, from which the development system generates C++ classes that must be tuned and extended by the application developer to reflect how the given component operates.

From the supplied information, the process visualisation development system generates an executable process visualisation program that can be connected to the process interface module and then be started to monitor the operation of the underlying process. The generated visualisation program includes an executable code linked from a visualisation kernel and the compiled C++ classes tuned by the user, and data files that describe the graphics elements and the layout of the views, list the objects representing physical components and define the structure of the underlying system. All of these are generated automatically, thus the development is easy and works mainly with concepts native to the underlying process.

The fundamental ideas behind this process visualisation development system are the application of reusable MVC (Model-View-Controller) classes and the automatic generation of the instances of such MVC classes from graphics definitions.

The MVC paradigm is a standardised concept to separate the user interface part from the rest of the application [9], [10]. The MVC approach divides the application into three parts:

1. The model part represents the problem domain and has no user interface elements.
2. The view part is responsible for the outer layer that is visible to the user.
3. The controller part controls the interaction between the model and the view.

There is a one-to-one correspondence between view objects and controller objects. A model object, however, can have many controllers and views. Generally, a controller and its view may be associated with multiple objects. Controllers connect the separated model and the user interface formed by view objects. Every controller object both instantiates and controls its view object. On the other hand, controller objects maintain control and route external events, such as user actions like mouse clicks or keyboard hits, to that model object which is responsible for reacting to it. To do so, that controller object must be found which is the target of an external event, then this controller can deliver the event to its model object. Controllers carry out this search by communicating with each other and with the help of associated view objects. The co-operation of controllers is co-ordinated by tree structures formed by controllers and views. The trees are created during initialisation when a controller object instantiates its view object and its subcontrollers. The top level controller is called the main controller and is usually instantiated by the main program. When a controller receives an external event, it may ask its view object to determine whether the event is within its view, or it may ask its subcontrollers whether or not the event is for them. If the event is within the view of a controller, then the controller sends a message to the model object. The model may alter its internal state and then informs its views to update themselves accordingly. In addition to delivering user action to model objects, controllers also convert the initial event to a message that is meaningful in the application domain.

It should be noted that in our approach the elements of MVC layers are implemented in the visualisation program kernel and are mainly hidden from the application developer. The application developer need not even know what the MVC concept is when he uses the development system. MVC objects and their interconnections are created by the kernel while it is reading data files that describe the given application. Thus, unlike most of the interactive programs which determine the majority of model, view and controller objects in programming time, our approach creates them in run time.

In the following chapters of this paper, we analyse the model, view and controller layers of the visualisation programs generated by the proposed development system and then the development methodology is discussed. It should be noted that this paper does not intend to invent yet another modification of the MVC paradigm, but simply discusses how the MVC paradigm can be applied to create a process visualisation system effectively and how the model layer should be elaborated to allow the automatic generation of the objects of this layer from graphics definitions. In order to present the specification of various parts of the program, the analysis models and notations of the OMT method proposed by Rumbaugh et al. [12] are used. The examples are taken from the fields of electric power supply and train traffic control systems. The applicability of the concepts, however, is not limited to these application areas.

The architecture of the generated visualisation program

In this chapter the model, view and controller layers of a program generated by the proposed development system is discussed.

Model layer

In a process visualisation system the model is an abstraction of the visualised system or process. The model reflects the components and the structure of the real system and is also responsible for simulating its operation. The objects of the model are abstractions of the physical components and their interconnections. As stated, the definition of the behaviour of the model can be greatly simplified and be made adaptive to evolving system structures if the behaviour is defined on 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 model class representing a component type describes 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 and the user himself. The reaction of a model object may involve the modification of the internal state and the transfer of messages to other model objects representing connected components 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 communication paths, that is by 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 identified for each component type and the communication is defined by sending messages to or receiving messages from these connection points. The actual partner of communication will be that object which is connected to the ”other end” of the connection point.

The state of a model object holds information needed to determine the effect of future messages. From programmer’s 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 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 arbitrary data,
- **FIFO, List**, etc. of those types.

Concerning practical situations a ”real component 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 that describes the operation of the wire 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 identifiers of the trains currently travelling on it. A track-switch has three connections. The point where a 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.

The classes of objects corresponding to components are specialisations of the abstract *Object* class. This *Object* class incorporates all general characteristics that are application independent. In order to introduce application specific features, inheritance is employed. Thus, an application object type (switch, track, etc.) is derived from the general *Object* class. In the subclasses the behaviour of the object - encoded by the *SetState* method - is redefined.

The application independent and application dependent parts of the model are shown in the following OMT object-diagram [12]:

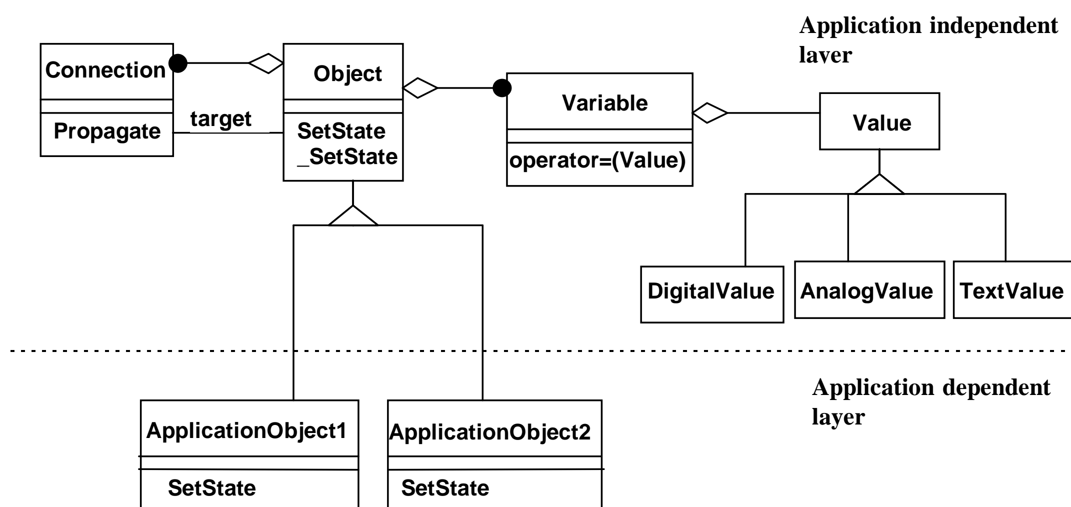

Figure 1: Object model of the model layer of the visualisation program

As shown in figure 1, the class called *Object* has central role in the model section. The most important method of *Object* is *_SetState* which is responsible for informing the object that something happened that might affect this object as well. *Object*s are connected to each-other by *Connection*s. An *Object* can have many *Connection*s, each of them represents a single direct communication path between two *Object*s. If an *Object* has been affected by the underlying process, user or connected objects, then it may want to update the state of other *Object*s connected to it. To do so, the *Object* must *Propagate* a message through its *Connection*s which call the *_SetState* method of that *Object* which is the *target* of this *Connection*.

An *Object* may have many *Variable*s representing its actual state. A *Variable*, in turn, has its actual *Value*. *Value* is an abstract base class for the *DigitalValue*, *AnalogValue*, *TextValue*, etc. classes. The actual type of a *Variable* depends on the actual type of its *Value* received through the overloaded assignment (=) operator. This means that each time a *Value* of different type is

assigned to a *Variable*, the *Variable* changes its type. This is the way how dynamic typing is implemented.

The object model of figure 1 provides a static view of the model layer of the visualisation program. In order to introduce its dynamic behaviour, we must consider how the model layer reacts to events coming from outside of this layer. According to the applied MVC paradigm, only controllers can send messages to the model objects directly. These messages deliver user actions and events occurred in the underlying process.

The communication sequence initiated by a message from the controller of a model is summarised in the following event-trace diagram:

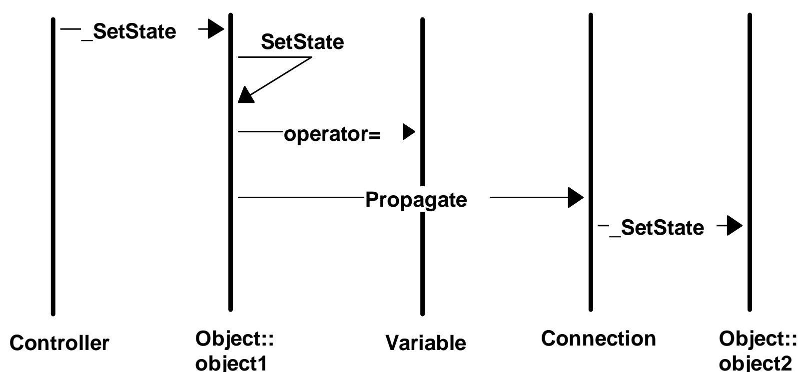

Figure 2: Event trace of the reaction of the model layer to external events delivered by a controller

If the underlying process changes or a user action occurs, the model layer of the visualisation system should be informed. The controller layer identifies that logical *Object* which the new data belongs to and gets this *Object* to react according to its behaviour. This is shown as a *_SetState* message in figure 2. Currently, a two-level message scheme has been implemented in this reaction, since reacting to external events usually consists of general and application dependent parts. First, the controller sends a *_SetState* message to the selected object. This method is responsible for the general reaction. For instance, having called the application specific part this method checks whether the internal state of the object changed and have those views updated which are associated with this model object. On the other hand, the general reaction method activates the specific reaction method called *SetState* which is responsible for the application specific features. *SetState* is a virtual method that is redefined on the application dependent level.

Receiving a *SetState* message, an *Object* may alter its *Variable*s (through *operator=*) and *Propagate* changes to other objects connected to it through *Connection* objects. The connected object will also receive a *_SetState* and indirectly a *SetState* messages. These are processed similarly, updating *Variables* and sending new messages to connected objects. This means that the effect of the external event will spread in the model and finally all affected objects will change appropriately. It is like a room where somebody turns the light on. The primary object is the switch which alters its state then propagates the ”energised” message to the wires connected to it. The wires propagate other messages to other wires and finally a lamp also receives an appropriate message causing it to alter its state and to start throwing light.

Each object may be both a receiver of a message and an initiator of a message subsequence.

Examining the communication diagram of figure 2, we should realise that it is almost independent of the actual application classes, except for the implementation of the *SetState*. Application programmer can thus be relieved of interfacing the model with the external world and interfacing different model objects.

View layer

The view must reflect the actual state of the underlying process, that is the actual state of the objects of the model. The top-level element of the view is the *MainWindow*. There can be several main-windows if the system uses multiple screens. A *MainWindow* instance incorporates a control menu, message subwindow, clock and lower level view objects responsible for graphics presentation. These objects are instances of the *SchemeWindow* class. A *SchemeWindow* may include dynamic *Drawing* objects, *Widgets* and *DialogWindows* that may be popped up in front of it. *DialogWindows* provide detailed, usually textual information about a model object and its variables. *DialogWindow* is the top of an inheritance hierarchy which include many different dialog window types, such as dialogs showing the values of a variable set, time-charts, question dialogs, etc., that are useful in process visualisation interfaces.

A *Drawing* or a *Widget* object reflects the most relevant characteristics of the actual state of a model object or optionally of its variable.

Drawings have different subtypes, including among others:

- *ColorDrawing* that consists of graphics primitives including polyline, polygon, rectangle, ellipse, text, etc. The controller of this view element can alter the colour of the primitives before causing it to be redrawn on the screen. This type of dynamization seems to be restrictive, but using a ”non-visible” colour we can create drawings which modify their shape or even provide a completely different appearance to reflect the change of the state of the visualised model object. The controller sets the colours of the graphics primitives by telling the drawing which variation of a finite set it should use. The identification of this variation is called the state of the drawing. The number of possible states are finite, thus a drawing is capable to visualise a digital value.
- *TextDrawing* that can display dynamic text.

The *Widget* class has also many subtypes that are common in the actual windowing system (currently OSF/MOTIF). Its capabilities are determined by the type of the widget. For example, a textfield is capable to display textual data, while a scale or a scrollbar can visualise numeric data.

All elements of the view are visually dynamic, since each time the actual state of a model object changes, the visual appearance of the affected view elements may also change. A single *Drawing* or a *Widget* reflects a single model object or a variable of that object. A *SchemeWindow* usually has many *Drawings* and *Widgets*, thus visualises a greater part of the underlying system.

Controller layer

The controller layer connects the model layer to the view and to the process interface.

The class of the main controller is called *VisualSystem* and is instantiated by the main program. Unlike in usual MVC designs, *VisualSystem* is not associated with the main-window of the program. There are two reasons for that. The visualisation program can use multiple screens, thus it may have multiple main-windows. On the other hand, a controller object should be assigned to the process interface which can also instigate actions (it means that the process interface is handled as a special view). We use the main controller for this purpose. *VisualSystem* communicates with the process interface and deliver its messages to the appropriate model object.

Unlike traditional MVC applications where the controller of the active window maintains control, here *VisualSystem* should always have the main control since events from the process can be expected anytime. *VisualSystem* parses the incoming messages and if a message encodes user input, then it gives the control temporarily to the respective subcontroller.

According to the MVC paradigm, each view object corresponds to a controller object. It also means that view classes should correspond to controller classes. The *MainWindow, SchemeWindow, DialogWindow, ColorDrawing, TextDrawing* and *Widget* view classes correspond to *Screen*, *Scheme, Dialog, ColorSymbol, TextSymbol* and *WidgetSymbol* controller classes, respectively. *ColorSymbol, TextSymbol* and *WidgetSymbol* controller classes are parts of an inheritance hierarchy having the *Symbol* class at its top.

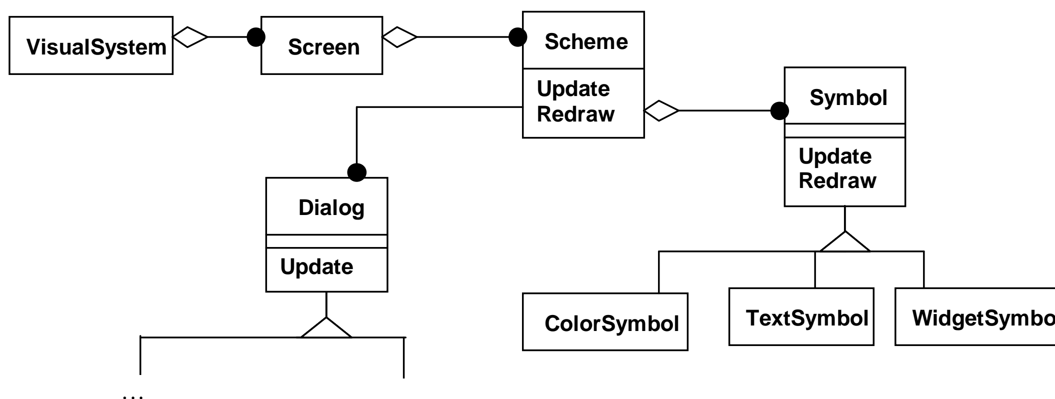

Figure 3: Object model of the controller layer of the visualisation program

Although the dialog view-controller pairs that are readily available in the development system seem to be sufficient in the majority of the applications, the application developer is allowed to add his own dialog types to the system. The controller class of such a new dialog should be defined as a subclass of the *DialogWindow*, thus the interface methods providing the connection between the model and the controller can be inherited.

Connections of model, view and controller layers

The connection between the model and the view is realised by controller objects. An object can thus indirectly have dialogs, widgets and drawings located on different *SchemeWindows*. Direct connections, however, exists only between controller and model objects and between controller and view objects. Note that this is similar to the MVC++ variation [13] of the original MVC paradigm, since we do not allow direct connections between the model and the view.

In addition to interconnecting view and model objects, the controller layer has another function as well. It translates user-interface concepts to application domain concepts and vice-versa. For example, a user event like a ”mouse click” may have a meaning ”close the switch” on the application level. On the other hand, the colour of a drawing (state of a view element) must be determined from the state of the model object, which may require computations.

In process visualisation systems, this translation process is mainly determined by the model object and is independent of the view and controller layers. Thus, this translation functionality may be migrated to the model classes where a default translation is provided which might be altered using inheritance. The method that converts user actions to application concepts is called *UserAction*, and the method responsible for determining the state of the views from the state of the model object is called *UpdateViews.*

The controller passes and also translates messages between the model and the external world interfaced by the view and the process interface. In a process visualisation system, message sequences can be instigated by the process and the user. The reaction to process events slightly differs from the reaction to user interventions, since view objects are involved only in processing the user interventions. In the following chapter the dynamic behaviour of the two types of reactions are discussed.

The connection of model, view and controller layers is shown in figure 4.

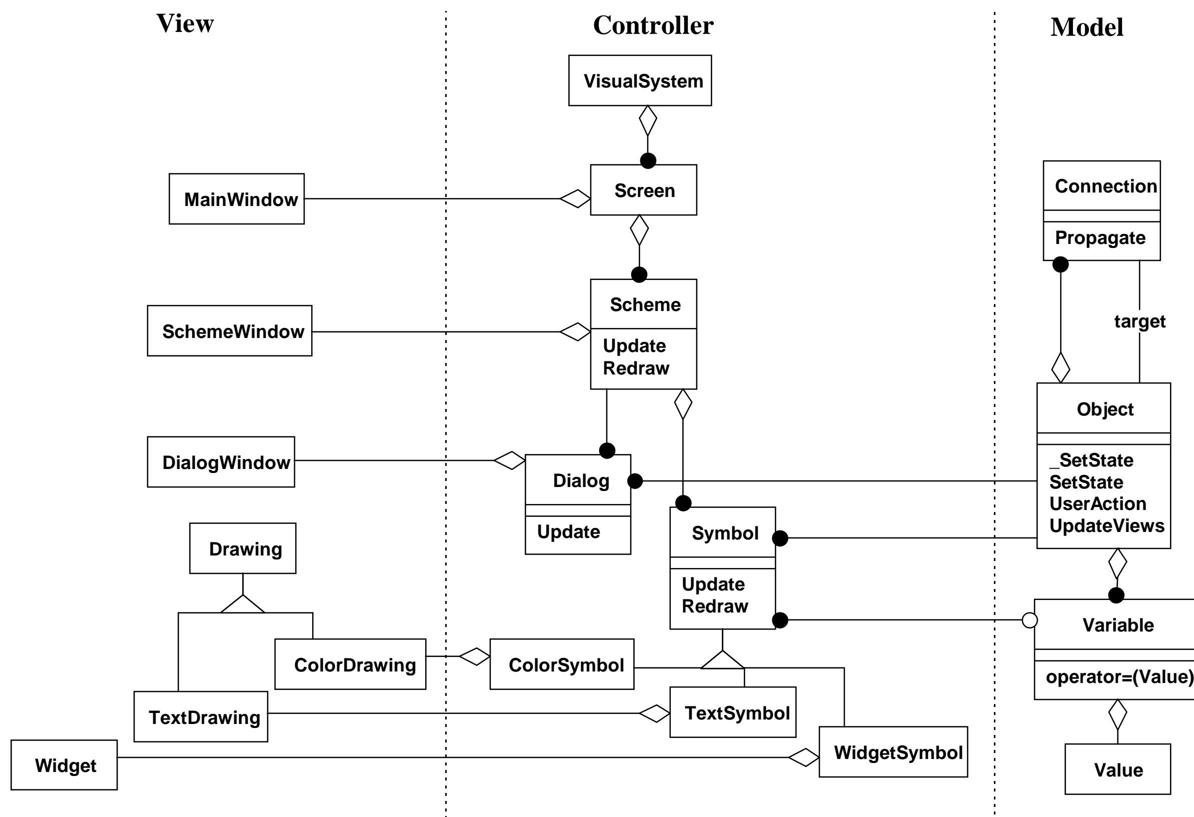

Figure 4: Connections of model, view and controller layers

Reaction to events and measurements in the process

When an event occurs in the underlying process or the result of a new measurement has to be transferred to the model, the process interface sends a message to the main-controller (*VisualSystem*). The main-controller identifies which model object this message belongs to and sends a *_SetState* message to that object. The model objects start communicating through their connection points and finally the complete model is updated.

As a part of processing the *SetState* message, an *Object* may intervene in the process. In order to do that, the *Object* simply sends an *Intervention* message to the *VisualSystem* object which transfers it to the process interface.

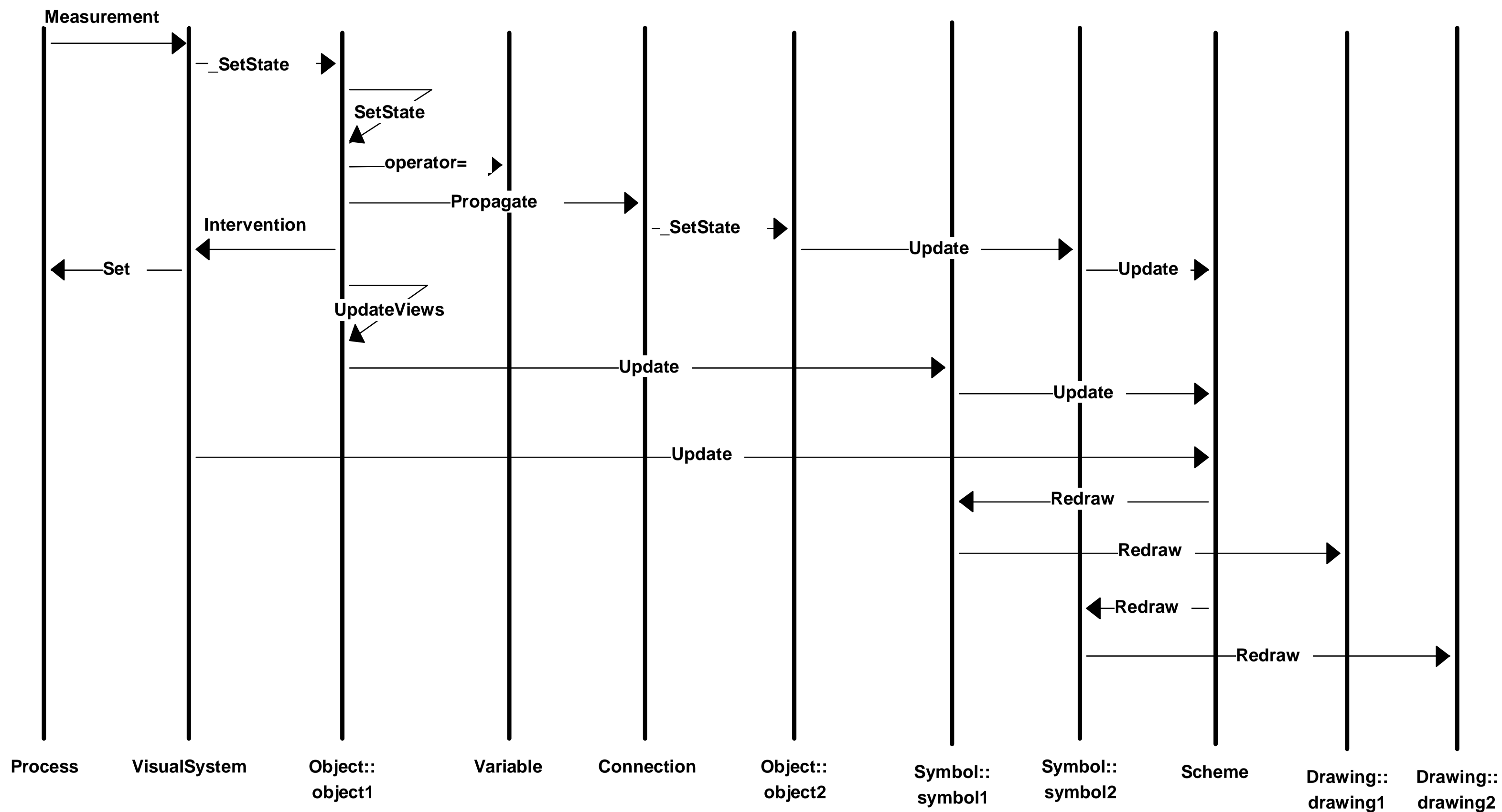

Figure 5: Event trace of reaction to process events

When an *Object* processes a *SetState* message, it may alter its state by modifying the value of its variables. Thus, the view layer should be informed to change accordingly. The model layer tells the controller layer to make the necessary modifications in the view.

Each symbol and dialog controller object corresponds to exactly one model object, while some of the symbols are directly connected to a single variable of that object. Whenever the *Variable*s of an *Object* change, the controls including the *Dialog*s and *Symbol*s must be *Update*d to ask them to update their respective view objects.

This update is straightforward for *Dialogs* since a dialog ”knows” what it needs from the state, thus it can poll the required information from the *Object* in its *Update* method. Similarly, a *Symbol* directly connected to a *Variable* is updated automatically by the new *Value* of the *Variable*. Although a symbol type can only display a single value type, this does not introduce any restriction into the system, since the variables are of dynamic type and their conversion is automatic. It means, for example, 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 gets it to convert 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”).

In the case of *Symbol*s not directly tied to *Variable*s, however, the application programmers should provide information how the set of *Variable*s determine the single *Value* which updates the *Symbol*. The virtual method where it must be described is called the *UpdateViews*.

Note that unlike in the traditional MVC approaches where the model does not know its controllers and views directly, here a model object does know its controllers due to performance reasons. Unlike other interactive programs where there can be many model objects but significantly less view and controller objects, in process visualisation programs the number of view and controller objects even exceeds the also large number of model objects. For example, the state of the same switch may be displayed in many different graphics schemes and dialogs. In this case, if all controller objects were asked to poll their model objects, or the dependant controllers and views of a model were searched in association tables, then the overhead would result in poor performance.

When a controller object is informed that its model has changed its state, it is expected to update its view. Dialogs and widgets can be updated independently since the screen - which is a resource shared by different widgets - is managed by the native windowing system. Drawings, however, are located on a single drawing-area widget of a *SchemeWindow*, thus their independence could not be provided by the windowing system. Let us assume that a drawing should be invisible to reflect some change in the model. If different drawings overlap - which is not exceptional - then this drawing cannot be made invisible by simply redrawing it with the background colour, since it would destroy other drawings. One obvious solution to this problem is the complete redrawing of the *SchemeWindow* when a single drawing changes. However, it is not feasible due to performance reasons. A *SchemeWindow* may consists of hundreds of drawings. An external event, such as a measurement from the process or user action, may alter many model objects and thus many drawings on the screen, which would get all drawings to be redrawn many times as a reaction to this external event. This would result in bad flickering and very poor response times. To overcome this problem, those regions of the *SchemeWindow* should be identified which need to be updated and these regions should be redrawn only once at the end of reacting to an external event. However, the view layer is not aware of events coming from the monitored process, thus the time of redrawing of the user

interface is determined by the controller layer. At this time, all drawings to be modified should be known in order to identify the regions of the *SchemeWindow* that must be redrawn.

Thus, when a model object tells its controller that its state has changed, the controller does not update its the view immediately. The controller just registers that its view will have to be updated and tells its supercontroller that its state has changed and it wants to get the control back when the reaction of the model to the external event is over. When the sequence generated by an external event is over, the control returns to the main controller (*VisualSystem*). The main controller can now recursively ask those subcontrollers, which have active views and have been registered, to update the views right now. When a *SchemeWindow* gets control, it first asks the registered symbols to transfer the bounding box of their respective drawings, then sets the clipping region of its view object, finally asks all - registered and non-registered - symbols to redraw their drawings. Drawings are redrawn in a descending order of the priority of their symbols.

Reaction to user actions

User actions are directed to widgets by the underlying windowing system using a callback mechanism. In our system all actions are directed first to the main controller which starts a search for the subcontroller to which this message belongs to. For dialogs and widget symbols, this search is simple since the callback parameters can include the address of the required control object. For colour and text symbols, however, the drawings of the active *SchemeWindow* must be polled one-by-one to decide within which controller's view the event is. Note that unlike redrawing, here the symbols are taken in ascending order of their priorities. If the symbol is identified, then the model object connected to this symbol is informed about the user action.

First, a method called *UserAction* is activated that translates user actions to events meaningful in the application domain, which, in turn, may activate the *_SetState* method. The transformation of the user intervention into the parameters of the corresponding *_SetState* message is pretty straightforward and can be defined independently of the application. In special cases, however, it seems to be advantageous to overrule this default transformation. Suppose, for example, that in a train traffic control system the user can drag-and-drop trains from track to track if the measuring instruments are broken down and the traffic is followed manually. In this example the implementation of the track's *SetState* method could be simpler if the drag-and-drop operation were transformed to a message that simulates the event that the measuring instrument detected the train movement from one track to the other. To support these special cases, although the visualisation system provides a default transformation, it also allows this transformation to be redefined by the application programmer by reimplementing the *UserAction* method (or a part of it).

From this point the operation is similar to that of the reaction to process events. When the reaction is over, the *VisualSystem* initiates the view update phase.

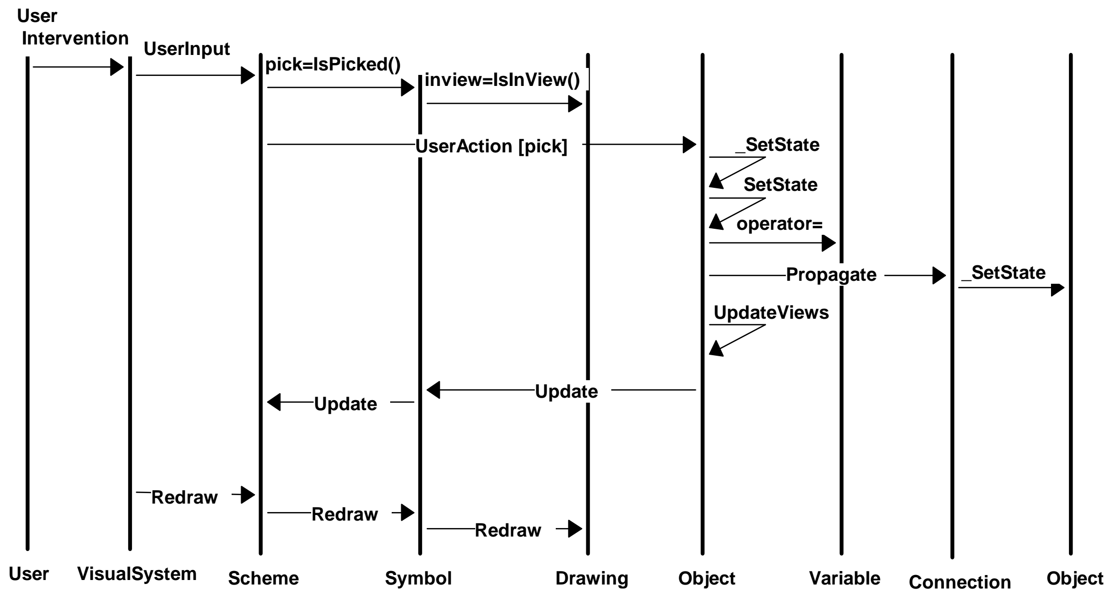

Figure 6: Event trace of reaction to user interventions

The method of defining process visualisation applications

So far, a general visualisation system has been discussed that can be adapted to solve concrete problems. In this section, we review the methodology used to create concrete visualisation programs and the tools supporting the various steps. This process is called the application definition.

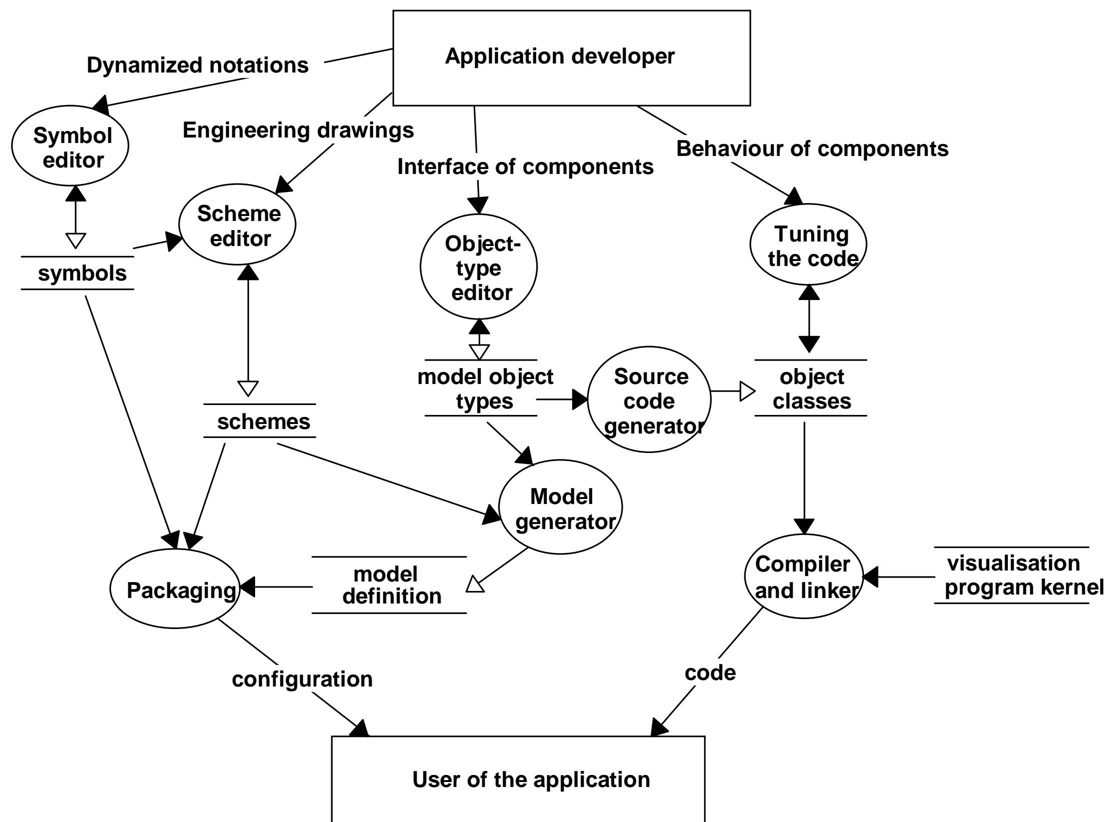

Figure 7: Dataflow of application definition

Recalling the requirements of a process visualisation development system, the application definition should heavily rely on graphics techniques and should require programming only on component level. The main flow of application definition is shown in figure 7.

Symbol editor

Symbol editor is a graphics tool for defining dynamic versions of engineering notations that are called *colour-symbols*. The kernel of the visualisation program will generate a *ColorDrawing* and *ColorSymbol* view-controller object pair from this definition. 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 assign colours to the primitives in each state. Since "non-visible" colour is also allowed - that can be made visible in the symbol editor, but is invisible in the running visualisation program - symbols that dynamically modify their shape can also be constructed. Finally, the connection points (which reflect the interface points of the real component visualised by this symbol) must be specified.

Link-symbols, on the other hand, have no fixed geometry. Their layout 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 8.

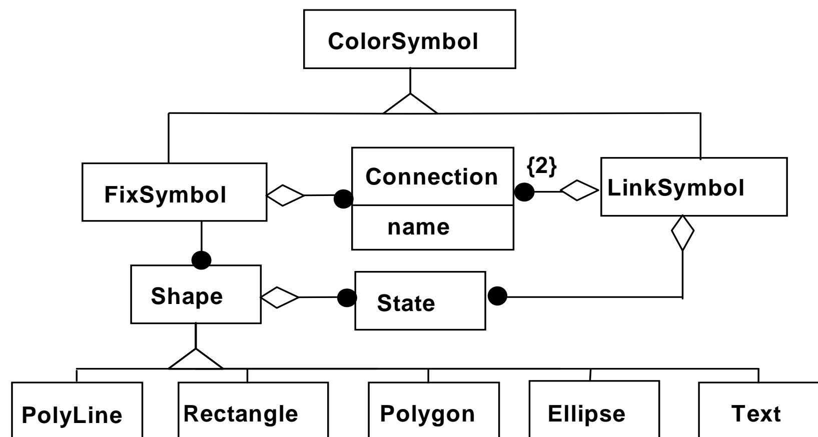

Figure 8: Object model of symbols in the symbol editor

In practical cases the standard symbols of engineering drawings are usually used, which are animated according to the actual represented value.

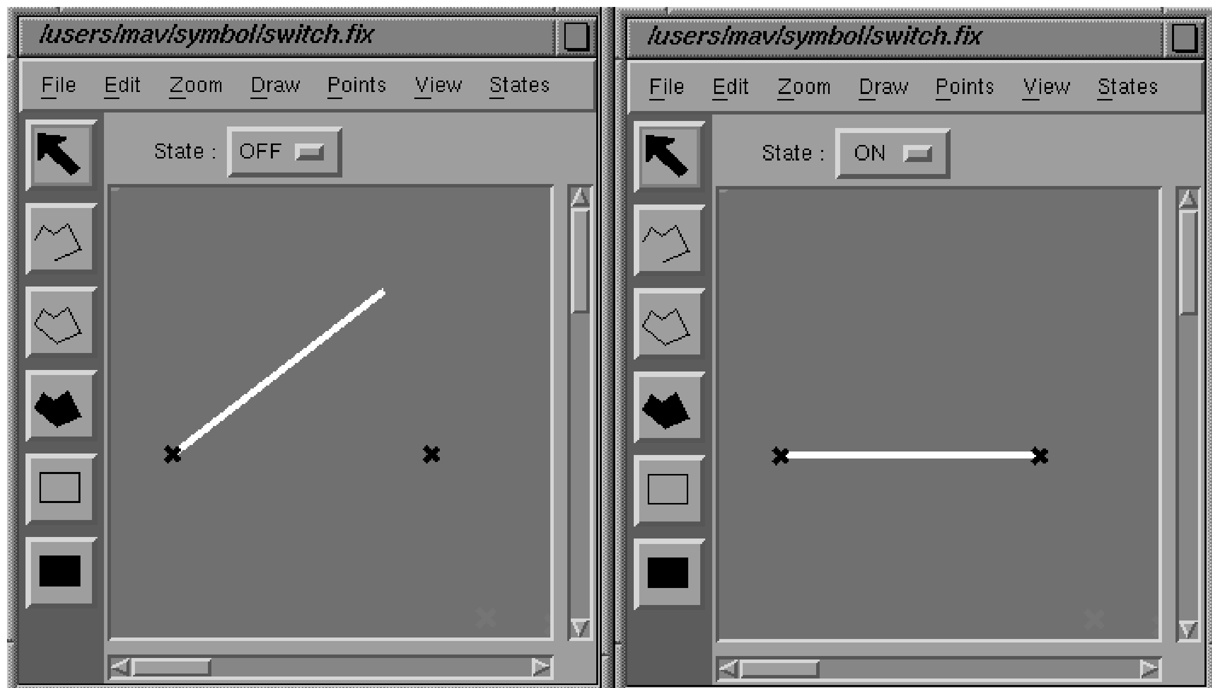

Figure 9: Two snapshots of the symbol editor showing a switch symbol in open and closed state

Scheme editor

Having defined the symbols, they will 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 as well. 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. 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), the scheme editor should allow to hide those objects which take part in the model, but are not visible on any scheme. Such hidden objects can be made visible in the scheme editor, but will be removed from the screen during the run of the visualisation program.

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

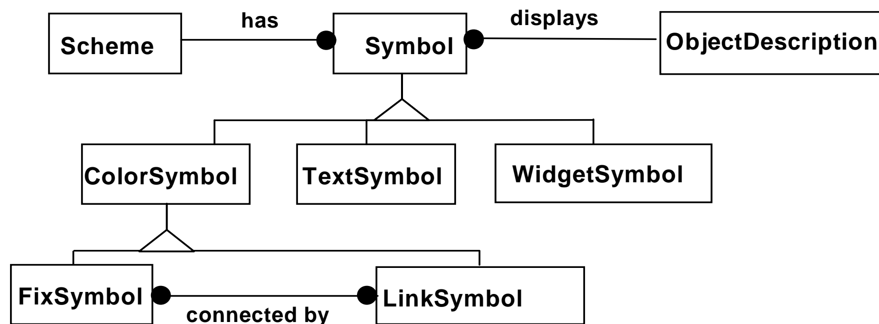

Figure 10: Object model of schemes in the scheme editor

Schemes are usually drawn from engineering 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 scheme. However, it is worth defining them in this stage of the design, because they take part in 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 requirement that a link-symbol can connect exactly two fix-symbols may seem to be restrictive, but it is not because of the following reasons. If we look around in practical systems, we will realise that connection components, such as wires, pipes, tracks, routes, or to be more abstract, telephone conversations do connect exactly two other components. This is also true when, at the first glance, the connection seems to connect more than two components, as for example, if wires or pipes are connected with each other, or we have a conference talk on telephone. In these situations, a special component appears, such as ”connection of wires”, ”connection of pipes” or ”telephone switching centre organising the conference talk”. Thus, connections still connect just two other symbols. Clearly, such a special component is not abstract, but has an important role in the physical operation. A ”connection of wires”, for example, forces all connected wires to have the same voltage, and makes the sum of incoming current flow equal to the sum of outgoing current flow. Summarising, link-symbols connecting two fix-symbols are usually enough. Should we need more general linking, new fix-symbols must be introduced, which tie link symbols together. Not only does not the introduction of such new components confuse the model, but it usually makes it clearer and reduce the number of required component types.

Model generator

The model generator will derive a list of objects that were specified as base objects of the symbols and will generate a list of object connections 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 be automatically updated.

Object type editor

The other branch of application definition is based on the definition of *model object types* (Fig. 7). This branch of application definition is based on declarative programming and automatic source code generation techniques. Using the *object type editor*, the application developer defines the variables and connection names of individual object types (this is a fully declarative step). 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

The definition expresses that a Switch has three variables, all of them of DIGITAL type. The possible values of variable *state* are ”switch_on” and ”switch_off”. The value set of *voltage1* and *voltage2* (the voltages of the two connection points), on the other hand, are ”non_defined”, ”energised” and ”ground”. Finally, a Switch has two connections called ”end1” and ”end2”.

Source code generator and tuning the generated code

From the description of Listing 1 the *source code generator* creates a C++ class with default behaviour. This C++ class must be altered by the application programmer to reflect the specific behaviour and properties of the object class. Thus procedural programming is required in the phase of ”tuning” the automatically generated code.

A strongly simplified version of the file generated from the above definition and completed by the application programmer 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 voltage_values { non_defined, energised, ground };

//============================================================
class Switch : public Object {
//============================================================
public:
    Switch( ) : Object( "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 energised:    Propagate(end2, M_ENERGISED);  break;
                     case ground:       Propagate(end2, M_GROUND);     break;
                     case non_defined:  Propagate(end2, M_NON_DEFINED); break;
                  }
                  switch ( voltage2 ) {
                     case energised:    Propagate(end1, M_ENERGISED);  break;
                     case ground:       Propagate(end1, M_GROUND );    break;
                     case non_defined:  Propagate(end1, M_NON_DEFINED); break;
                  }
                  return OK_RESPONSE;

              case M_SWITCH_OPEN:
                  state = switch_off;
                  Propagate( end1, M_NON_DEFINED );
                  Propagate( end2, M_NON_DEFINED );
                  return OK_RESPONSE;
           }

       case end1:                  // Message from "end1" connection point
           switch ( mess ) {
              case M_ENERGISED:     voltage1 = energised;   break;
              case M_GROUND:        voltage1 = ground;      break;
              case M_NON_DEFINED:   voltage1 = non_defined; break;
           }
           if ( state == switch_on ) Propagate( end2, mess );
           return OK_RESPONSE;

       case end2:                  // Message from "end2" connection point
           switch ( mess ) {
              case M_ENERGISED:     voltage2 = energised;   break;
              case M_GROUND:        voltage2 = ground;      break;
              case M_NON_DEFINED:   voltage2 = non_defined; break;
           }
           if ( state == switch_on ) Propagate( end1, mess);
           return OK_RESPONSE;
       default:
           return INVALID_RESPONSE;
    }
}
```

Listing 2: Class definition of the switch

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.

In the above example, a Switch may receive messages from the underlying process (M_SWITCH_CLOSED, M_SWITCH_OPEN), which determine whether or not the switch is closed or from its two connection points (M_NON_DEFINED, M_ENERGISED, M_GROUND) which inform the switch about the voltage of the object connected to its two connection points.

The operation is simple:

> If the switch is opened, then the voltages of objects connected to this object are not affected by this switch, that is, the voltages are either determined by something else or undefined. The connected objects are informed about this event by an M_NON_DEFINED message. If the switch is closed, then the object connected to one end of the switch is forced to have the same potential as the object on the other end. If the switch is closed and the potential of the object connected to one end changes, then the object connected to the other end is also instructed to follow the change.

If we want to develop a complete visualisation program of an electric energy distribution system, we need just five other component types in the simplest case: wire, connection-of-wires, transformer, grounding and consumer. Their operation and thus their C++ classes are even simpler than that of the switch. A wire or a connection-of-wires simply passes all messages that are received from one of its endpoints to the others. In the meantime it decides whether it is energised for presentation purposes. A transformer and a grounding may send M_ENERGISED or M_GROUND messages to connected components, respectively. A consumer updates its state upon receiving messages from its connection points.

We have to mention that tuning automatically generated code fragments raises the problem of what happens to additions and modifications if the code needs to be regenerated later. There are different alternatives but none of them is perfect. A possible solution uses special comments in the generated code, that are used to separate automatically generated and manually input program lines. These comments control the regeneration process and only those lines are replaced that are in between comments which designate automatically generated parts.

Compiling and packaging

The generated and tuned C++ classes are compiled and linked with the kernel of the visualisation program resulting in an executable code. When running the visualisation program, this code inputs data files describing symbols, schemes and the definition of the model, then builds up the model, view and controller layers accordingly and interconnects them. The code segment providing this input is the part of the kernel, thus, from application programmer's point of view, all objects are created automatically and without requiring programming. This approach has obvious benefits but also raises a problem. The kernel cannot be aware of the type of model classes since model classes, such as switch, wire, etc. are created during application definition. Creation of objects, however, requires the definition of the class to be instantiated. In order to overcome this problem, the kernel assumes a method called

```
Object * AllocateObject(char * class_name)
```

and uses this method to instantiate a class whenever it is expected to do that during the input of the configuration. AllocateObject, however, is not implemented in the kernel.

Before compiling the application, the user should certainly specify what the new object types are and what application files must be compiled and linked with the kernel. Using this information, the development system "writes" a program source file that includes the declarations of the new object types and implements the AllocateObject method. This file is also compiled and linked with the kernel to make the generated code complete.

Files describing symbols, schemes and the model are in text format and apply a formal language defined for this special purpose. Thus, it is possible (but is not recommended) to define the complete visualisation without using the graphics editors. In order to speed up the initialisation of the visualisation program, these text files are pre-compiled in a *packaging* phase, providing a set of data files called the *configuration.*

A characteristic view of the running visualisation program displaying a part of an electric energy distribution system is shown in figure 11.

Conclusions

This paper proposed a method for developing process visualisation programs, taking advantage of the MVC paradigm and OO methodology, and discussed a development system supporting this method. In order to develop an application, the graphics representation of the symbols must be drawn, the screen layouts that also represents the structure of the underlying model 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. Since the definition of the behaviour 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 and the definition of their behaviour requires just a couple of C++ statements. Thus the only problem which requires significant effort is the scheme definition.

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 was a challenging one, since unlike the previous examples, in resistor networks there is a very strong coupling between the states (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 of 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.

We have to admit that this paper discusses only the relevant aspects of the implemented visualisation development system, especially detailing the method of model definition and interfacing between the model and its graphics presentation. A practical process visualisation program and its development system, however, should have other features as well, which we

did not address here because of space limitations and because of the fact that these features can be separated from the discussed key concepts. Among others, these features include:

- Automatic logical-event generation, alarming, user acknowledgement and logging.
- Distribution of the model in a networking environment.
- Distribution of control rights of the users in a multi-user environment.
- Interfacing with the process.
- User identification and security considerations.
- Remote installation and configuration.
- Robustness and fault tolerance.

The largest project in which this development methodology and the development system itself have been used was the visualisation of the railway line between Budapest and Vienna. In this project the train traffic, state of the safety equipment and the electric energy distribution system had to be monitored and controlled. The following table summarises the most important data of this project and the size of the development system itself.

Hardware	40 Sun 4 workstations connected in a TCP/IP network (500 km long fibre optic cable)
Operating and windowing system	Solaris 2.1/OSF-Motif
Number of component types	82
Number of model objects	~25.000
Number of symbol types	96
Number of view objects	~27.000
Number of schemes	53
Number of C++ lines in the application dependent part, including both automatically generated and manually entered lines	~20.000
Number of C++ lines in the kernel	~40.000
Number of C++ lines in the complete development system	~75.000

Table 1: Characteristics of the train traffic and electric power control application

Acknowledgement

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

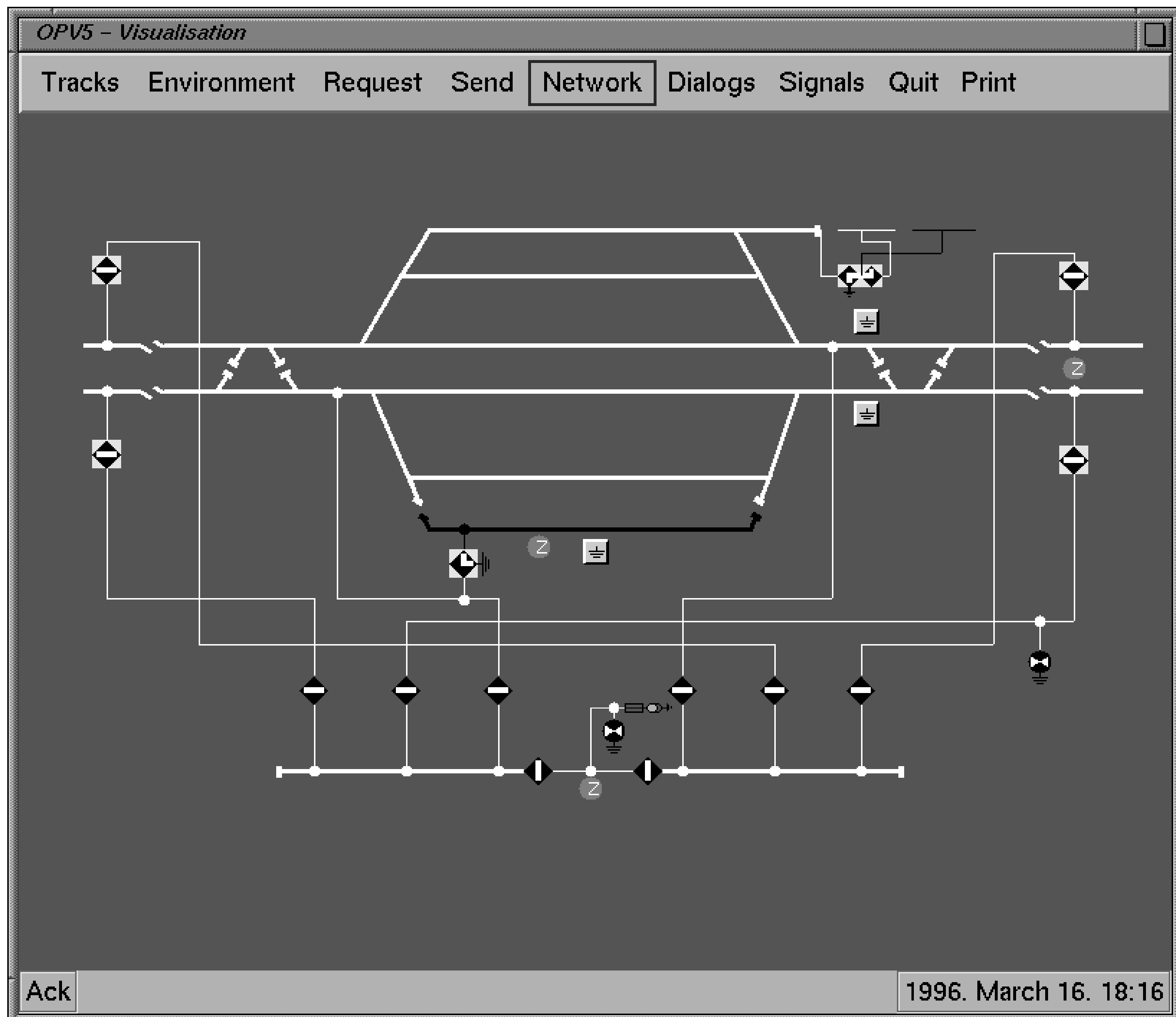

Figure 11: Snapshot of the visualisation program

References:

[1] *FIX DMACS - System development*, Display development - Intellution Inc. 1992-1994
[2] *Vision -Process Visualisation system*, DIVICON Ltd.
[3] *Powerful Tools to Monitor and Control Live Processes,* URL: http://www.dvcorp.com/mktg
[4] *Visual Designer - Intelligent Instrumentation*
[5] *PVSS - Prozess-Visualisierungs- und Steuerungssystem*, EDV-Technik Mühlgassner GesmbH
[6] *Sammi, Graphical framework for real-time command and control*, Kinesix/Scientific Software-Intercomp
[7] *Points to consider in evaluating Dynamic Data Visualisation Tools*, URL: http://www.telsa.hl.com.au
[8] P. Mégard: *Criteria for Selecting a good GUI Development Tool,* URL: http://www.ilog.fr/Products/Views
[9] R.A. Earnshaw, N. Wiseman: *An Introductory Guide to Scientific Visualization,* Springer-Verlag, 1992
[10] W. Lalonde, J. Pugh: *Inside Smalltalk* (Volume II) Prentice Hall, 1990
[11] G.E. Krasner, S.T.Pope: *A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80* Journal of Object-oriented Programming, August/September, 1988
[12] Rumbaugh, Blaha, Premerlani, Eddy, Lorensen: *Object-oriented Modeling and Design,* Prentice-Hall, 1991
[13] A. Jaaksi: *Implementing Interactive Applications in C++,* Software -Practice and Experience, Vol. 25(3), 271-289 (March 1995)
[14] N. Knolle: *Why Object-oriented User Interface Toolkits are better,* Journal of Object-oriented Programming, Vol. 2, 1989
[15] B. Shneiderman: *Designing the User Interface,* Reading Mass., Addison Wesley, 1986