A Procedure for a Unified Approach to Analysis and Development Projects

Bernd Eichenauer, IBE

1. Preface

Below we describe a procedure for analyzing and developing systems with discrete concurrent processes. Its basic elements have been known since the sixties [1], but because of its awkwardness it has hardly been used in practice until a few years ago. Now that fast and user-friendly computers are available at almost every workplace, we have the prerequisites for an efficient use of the procedure, which numerous authors have extended in the last three decades.

Recently the procedure has been implemented in practice and adapted to today's user interfaces (e.g., [2]). The result is development tools that are easy to use and that can be employed wherever technical or commercial processes need to be planned and implemented.

This article was written to show that previous objections to the Petri net method no longer apply, and to demonstrate the enormous advantages this method provides in working on analysis and development projects.

2. The Three-Phase Model in Processing Analysis and Development Projects

If one gave a system developer the task of developing a product, for example a computer program or a vehicle, and of delivering it without any tests, the developer would rightly reject such a project. We know that we make mistakes during system development, and that most of these errors can be eliminated through the use of follow-on tests. No one doubts today that a product should be delivered only after extensive quality assurance measures [3]. Normally one can assume that, depending on the application area, one must allocate between 30% and 50% of the total development costs for component and system tests of technical products.

While testing procedures can be applied without problems during development of technical products, (since prototypes of the products are available for testing), until recently this was not possible when designing or changing operational processes, aside from the rarely used and expensive practice scenarios. As a result, modeling was often concluded after the specification phase, and experience with the dynamic behavior of the specified model had to be obtained with "real life" tests.

Removing the planning errors and weak points which are unavoidable with this approach often results in enormous additional costs which could have been at least partly avoided, had appropriate planning aids been available. Surely the numerous corporate reorganizations about which we hear so much today can be traced back in part to earlier planning errors.

In recent years, due to rapid progress in computer technology, development systems have become possible which allow us to economically create realistic prototypes of general technical and/or commercial systems before they are implemented. This allows us to gain early experience with the dynamic behavior of a planned system, and to use the model to study and optimize the real system to be implemented.

A prerequisite here is a suitable specification method for model design, one that lets us consistently describe both the static and the dynamic behaviour of the system under design. For this we need a descriptive language which can describe all aspects of the model to be developed, and which has none of the flaws we know from the descriptive means in the many available CASE tools. We know that consistency in descriptive methodology means efficiency in development and alteration, a better overview and fewer errors.

The first two phases of a project are the specification phase and the simulation phase. For many applications that are restricted to analyses (which occur especially in the commercial field), only these first two phases are significant. In a third phase, which becomes more interesting in technical applications, the model can be connected to its environment and then, where applicable, be implemented directly as an automation program. With increasing interweaving of operational and technical processes (e.g., with decentrally organized production systems) and with faster and faster hardware, this third phase is becoming ever more important.

3. The Specification Phase

To implement the three-phase model, the frequently used, more informal descriptive procedures (flow diagrams, hierarchy diagrams, etc.) do not suffice. Rather, we need an exact system description, without which a meaningful simulation and, even more important, the program implementation, is not possible. But at the same time the modeling process should be so simple and practical that even users without extensive programming experience can apply it successfully.

3.1 Extended Petri Nets

One methodology which fulfills both the demand for exactitude and that for simple formulation was already proposed in the early sixties by Petri [1]. The Petri nets named after him (e.g., [4,5,6]), while they do allow an exact graphical model specification, were previously so impractical in their use that initially they were of only scientific interest. But through the possibilities of modern computer technology, the method, with numerous extensions in recent years, has become much more attractive, and it is currently one of the few practically applicable methods which are suitable both for specification and for the simulation of discrete concurrent processes.

Superficially, a Petri net is similar to a flow diagram. Both are based on graphs, and both depict the current system status graphically. But in contrast to flow diagrams, Petri nets are strongly structured. They show an exactly defined graphical syntax and clear semantics.

Petri nets are event-oriented, and can be constructed from a few simple graphical elements. They consist of two different node types (places and transitions), of connectors and tokens (Fig. 1).

Tokens represent the objects to be processed, such as information, materials, accounts, tools, warehoused goods, transport vehicles, etc. Status changes in the system are reflected in the movement of tokens from places to transitions and on to other places.

Places are the passive elements of a Petri net. They serve as storage areas for information and tokens. The location of tokens on the places of a Petri Net represents the current status of the net.

Transitions are the active elements of a Petri net. They execute actions and process information. A transition removes tokens from entry places, changes their characteristics according to procedures specified by the user, and places new tokens on output places.

Connectors connect places and transitions.

We say that a transition "fires" when the conditions are met for removing tokens from their input places and subsequently allocating other tokens to their output places. Conditions for firing could include having a token on all input places of the transition, or that during firing the predefined capacity of an output place is not exceeded.

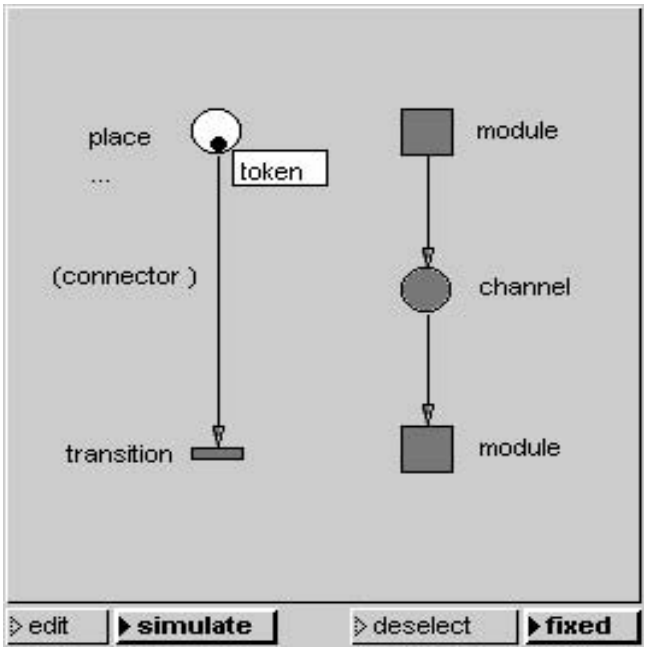

Fig. 1: Net elements of generalized Petri nets

Over the years the original Petri nets, like every other technical development, were improved and extended numerous times. This included significant expansion of the original Petri net concept, so as to increase user acceptance. The goal was to make available efficient tools which allow the rapid and intuitive design of complex systems. Below we discuss a representative sample of the extensions which we make use of during the modeling process.

3.2 Object-oriented Model Design

Petri nets are excellently suited for object-oriented design of applications. After all, the different elements of Petri nets abstract the general building blocks of event-oriented, parallel, discrete systems. The dynamic object flow is formulated independently of what the components of the Petri net mean in the real world. That is why Petri nets can be used to design the most various applications.

What meaning the individual elements of a Petri net have in a specific application area is determined by the user himself, from case to case. Thus, as mentioned above, the dynamic objects that move within a Petri net (tokens) can represent a message in the model of a communications system, while in the model of a production system they represent some product or material on a conveyor belt. Since tokens are stored in places, the places serve to model the senders and recipients of messages, or to model theproduction line. And transitions indicate how messages are transmitted, and what processing measures should take place, or they model the processing machinery for the material on the production line.

How flexible and simple the correlation between the components of a Petri net and real objects is, can be seen in the intersection model depicted in Fig. 6. Here the tokens represent the different vehicles, the places represent the intersecting streets, and the transitions are used to depict, among other things, the traffic lights.

3.3 Naming and Attribution of Petri Net Elements

To describe processing within transitions and for attribution of tokens and connectors we can in principle use any desired programming language. But any programming language to be used to process Petri nets should meet the following requirements:

- As already mentioned, the various net elements must be viewed as objects to which the user assigns real objects during modeling; therefore the programming language to be used should also be object-oriented. If the individual objects of the Petri net are instances of appropriately defined classes, we can consistently formulate an object-oriented model.

- The language should allow rapid switching between the editor and the simulator during development. This can speed up the development process; e.g., partial nets can be tried out without problems, and errors can be quickly eliminated during debugging.

- The language should be available for all current computers and operating systems, and should make it possible to transfer nets effortlessly from one computer type to another.

- The language should be simple and intuitively understandable, and should be easy to learn and implement, even for users with little programming experience.

One programming language that fully meets all these requirements is Smalltalk-80 as implemented by ParcPlace Systems Inc. [7,8]. Smalltalk-80, combined with the class library, provides a powerful object-oriented language, and can be put to use without extensive programming experience. In our examples further below Smalltalk-80 code was blended in everywhere.

Most of the elements in our expanded Petri nets can be labeled with text. While comments can be formulated as free text strings, the actual inscriptions are Smalltalk commands and must therefore meet the Smalltalk conventions.

The behavior of transitions during the firing of tokens is defined via inscriptions. The interaction occurs as soon as messages are exchanged with the incoming tokens. Messages can alter the attributes of tokens, or test their value, if e.g. the execution of the transition code depends on the current status of the net (the value of a token attribute).

To every initial token we can assign any desired number of attributes. Each attribute is defined by Smalltalk commands which are executed when the token is produced. The value of the last command provides the value of the attribute. The inscribed Smalltalk code can consist of a very simple object, for example an integer or another literal; but it can also be a complex program. Fig. 2 shows a window in which the initial tokens for the place "place1" are defined. "place1" consists of four initial tokens. The third token has two attributes. The second attribute is the integer 22.

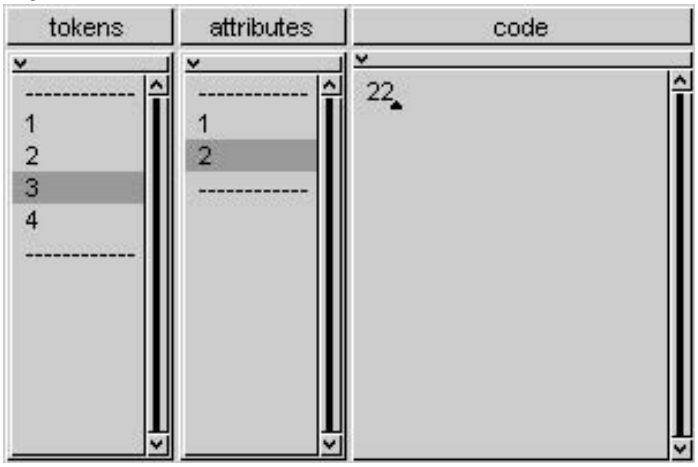

Fig. 2: Declaration of initial tokens and their attributes

As with tokens, we can assign attributes to connectors as well. Through such attributed connectors only those tokens can flow whose attributes are compatible with the connector attributes. This allows, among other things, the selection of a specific connector for the flow of a token.

The inscriptions of transitions are subdivided into the three components: condition code, delay code and action code.

The condition code, which must produce one of the boolean values, "true" or "false," allows us to make activation of a transition dependent on the values of the input tokens. If processing of the condition code results in the value "false," the transition cannot be activated.

Classical Petri nets operate without reference to time. All transitions are processed simultaneously if their required conditions are met. This allows modeling of the "simultaneity" of parallel processes. This of course is inadequate for modeling real systems and for the process-dependent real time behavior of a target system. For this reason some Petri net extensions permit assignment of time-related behavior to each transition, by delaying the triggering of a transition. By way of the delay code, which must produce a non-negative number, the firing of a transition can be delayed by as many simulator time units as the number states. We will come back to the concept of a "simulator time unit" later.

The action code, finally, is processed as soon as the transition has fired. Action code is usually employed to assign values to variables, which are used elsewhere in the net, to change attributes or to produce side effects. For example, we can use the action code to update a list displayed in a window, to show its latest status, or to read in a value from a control input device.

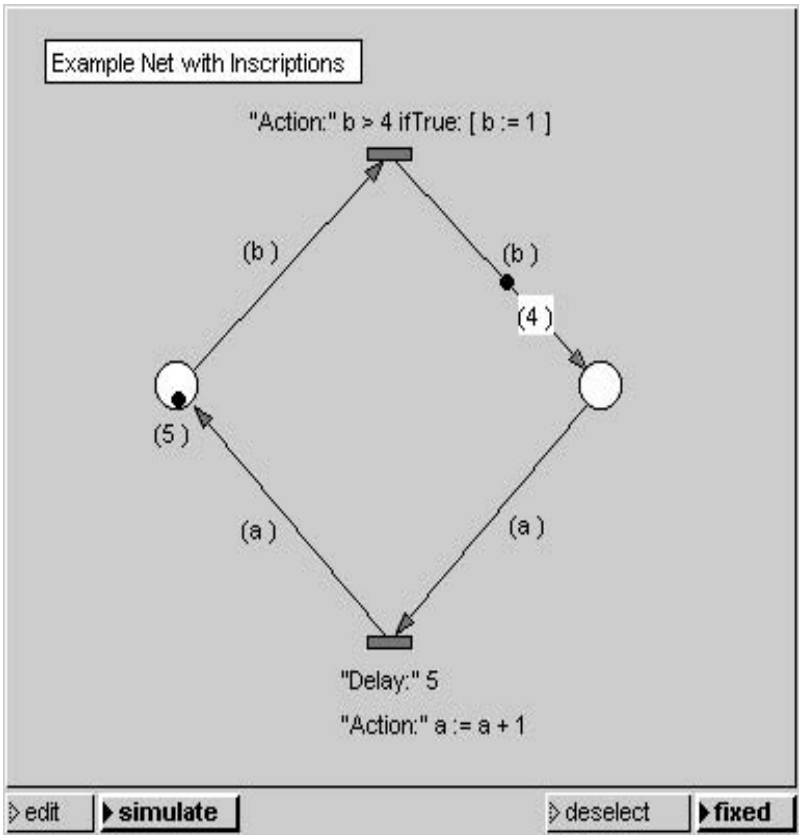

Fig. 3: Simple example net with Smalltalk inscriptions and tokens

Fig. 3 shows a simple net in which Smalltalk inscriptions are used. The tokens carry values as attributes; by means of the action codes these values are increased, or, if a boundary is exceeded, the values are set back again. A token with the attribute value 4 runs directly from the upper transition to the right place. A second token with the attribute value 5 is waiting in the left place.

3.4 Start- and End-Code

We can assign a start- and end-code to every net. The start-code often serves to initialize global variables, and is executed when the net is started. With the end-code we can, for example, set back external devices used in the simulation or during execution of the net.

3.5 Hierarchical Structuring

For the sake of a clear overview, complex nets are structured hierarchically, i.e., subdivided into a primary net and subnets. The subnets to be assigned to the individual levels can be freely defined at any time, through selection of network nodes of higher-lying levels. The number of hierarchical levels may not be restricted. This makes it possible to develop a system top-down or bottom-up, as desired. Sometimes we even use a middle-out development.

Essentially there are two types of subnets, namely "modules" and "channels" (see Fig. 1).

Modules are active elements which may contain all types of net elements. Each module can be stored individually, and can be re-loaded, or inserted into an existing net. For reasons of transparency the interfaces to modules in the upper net level should also be shown in the refined lower net, so that the integration of the module is clearly visible in the lower net as well.

Channels are refined passive net elements which themselves consist of places and channels. Here also, to provide a clear overview, the interfaces to the active elements in the upper net should be clearly recognizable in the lower net

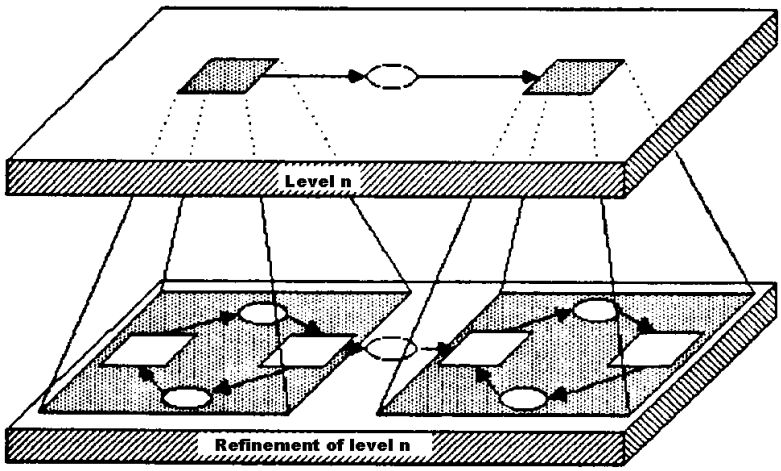

Fig. 4: Establisching hierarchical levels for Petri nets - refinement of levels

While the importance of modules during net development is self-evident, that of channels, whose use is advantageous in creating complex nets, is not immediately clear. Therefore, below we want to clarify the channel concept with the following analogy:

> Places behave like individual connection lines between hardware modules. Channels behave like pipes which encase bundles of connection lines, and they connect modules.

3.6 Several Tokens Per Place

In the original Petri nets, a place was able to store only one token. But in practice it quickly becomes clear that nets which allow only one token per place tended to become very bloated even with small models. To provide an improved overview and to allow reality-proximate formulation of models, many Petri net extensions provided for use of any desired number of tokens. The maximum number of tokens at a place can usually be specified by the user.

Beyond that, many extensions allow specification of a deterministic or random processing sequence for tokens, if a transition is to fire several tokens simultaneously.

3.7 Statistical Functions

As a rule, Petri net tools contain online statistical functions in the form of freely definable line and bar diagrams. Diagrams with statistical information can often be connected to the various net elements with a few mouse clicks. We will discuss this further in Section 4.

3.8 Freely Definable Graphics for Net Elements

The user interface described above has the same advantages and disadvantages as the user interface of many other development tools. On the one hand, the user interface can be easily learned because of the limited number of elements. On the other hand, even small specifications, and large ones all the more so, quickly lose a clear overview, even if they are organized hierarchically. Anyone who has ever reviewed or produced a larger graphical specification, organized, e.g., according to "structured analysis," will know exactly what we mean.

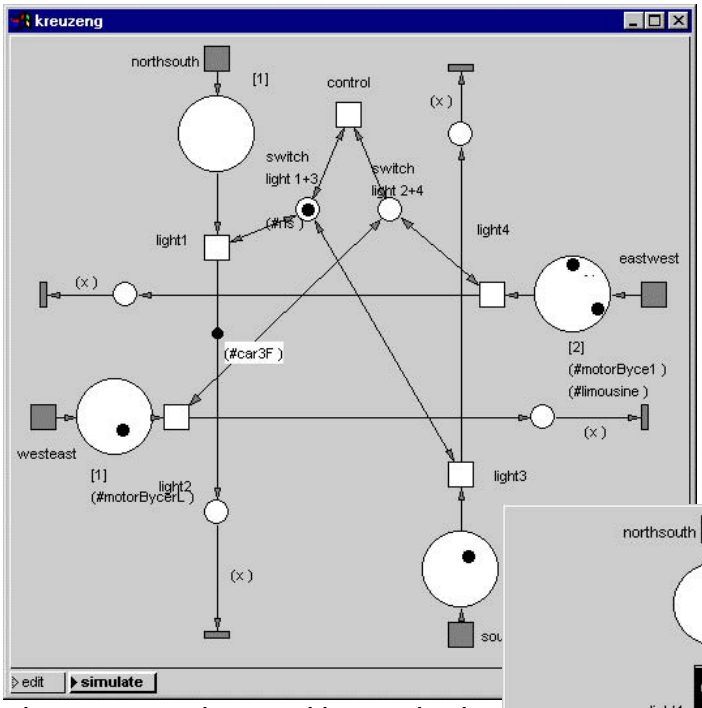

Fig. 5: Example net with standard icons

Since our design method is object-oriented, we have the option of assigning graphic representations to the individual net elements. After all, the user of extended Petri nets associates his objects with the various net elements. Therefore it is useful to replace the previously described standard icons with graphics that clearly depict the association of a net element with an element of physical reality.

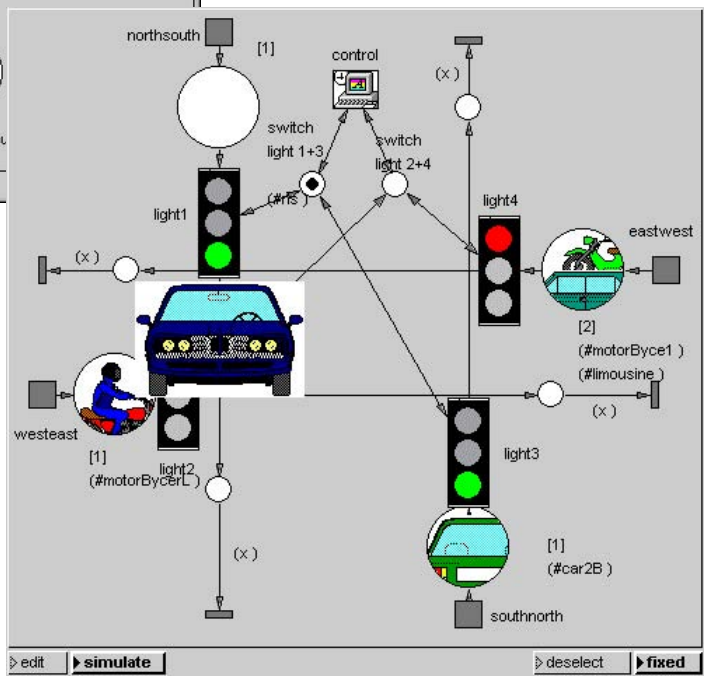

Fig. 6: Example net from Fig. 5 with user defined icons.

For this reason, several Petri net tools offer the option of replacing the standard icons shown in Fig. 1 with icons or bitmaps. This option is very important, because when one is using or evaluating models, one is usually not interested in the most minute implementation details, but rather in the user interface that the tool offers. Just as with conventionally developed program systems, the user usually cares only about functionality and the user interface. He doesn't care about the implementation process or the associated source code. At least the uppermost hierarchy levels of models used for daily work should be intuitively understandable and usable.

Comparison of the two example versions shown in Fig. 5 and Fig. 6 shows how dramatically we can improve understanding of a net by using icons for the net elements. Both figures depict the same net. Although it is rather simple, the standard icons in Fig. 5 make us struggle to see what application is depicted. Fig. 6 makes the application evident at first glance.

To create the user icons we can use any of the numerous paint programs, or an icon editor. Most icon editors are shipped with huge icon libraries, allowing easy insertion of icons into nets (assuming that the development tool being used provides for icon import).

3.9 A Complete Example

Let's examine the simple transport problem shown in Fig. 7. It consists of two operating robots and a turntable with three positions. If the container in Position1 is empty, then Robot1 can put a part in it. If the container in Position3 is full, Robot2 can remove the part from it. The turntable requires a time unit for turning the containers from one position to the next.

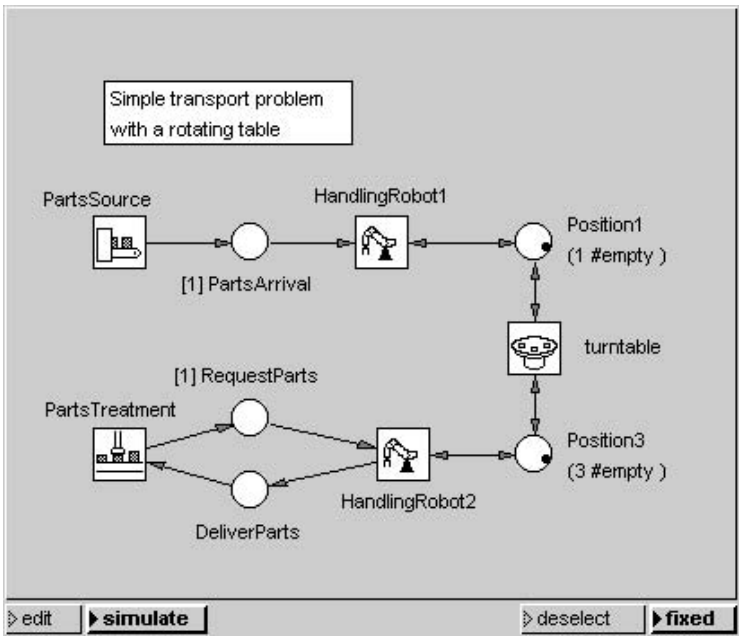

Fig. 7: Uppermost hierarchy level of a simple transport problem

To conserve space, we won't discuss the source of the parts to be moved (module "Parts-Source") and their further processing (module "Parts-Treatment") here. We'll simply assume that the parts will arrive and will be processed statistically.

Fig. 8 and Fig. 9 show the two modules, "PartsSource " and "PartsTreatment." Here we clearly assume a random distribution of the points in time when a part is delivered or used. Delivery of parts is as follows (Fig. 8): the token located at a given place is delayed by the random value of the time delay. Then the transition fires, and delivers a token to each of the places connected to it. One is used to plan the next part delivery. The token which runs to the place "PartsArrival" represents the part delivered from outside the system.

Fig. 10 and Fig. 11 model the work of the processing robots. Here again we're looking only at the delivery of parts. In the container at "Position1" there is a token whose attributes indicate which of the three containers on the turntable is in position, and that this container is empty. The transition "HandlingRobot1" can fire only if, along with this token, there is also a token (a part) in the place "PartsArrival." If the transition fires, it

collects both tokens and places a token with the attribute value "full" (a part) in the container at "Position1."

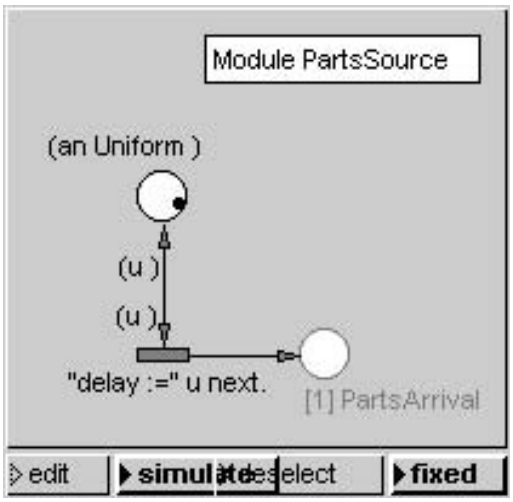

Fig. 8: Production of parts

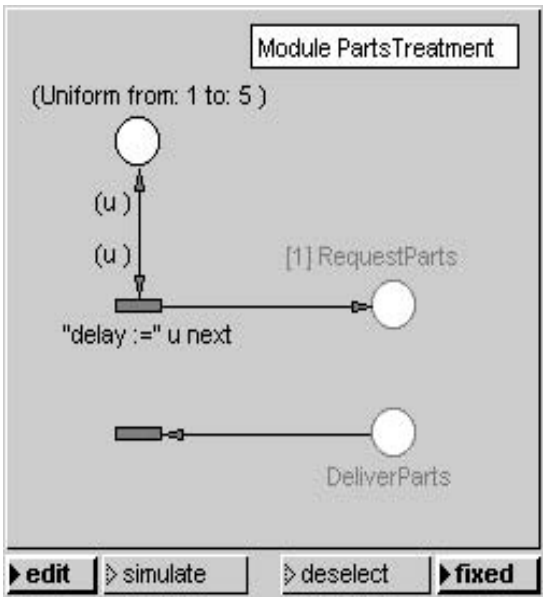

Fig. 9: Use of parts

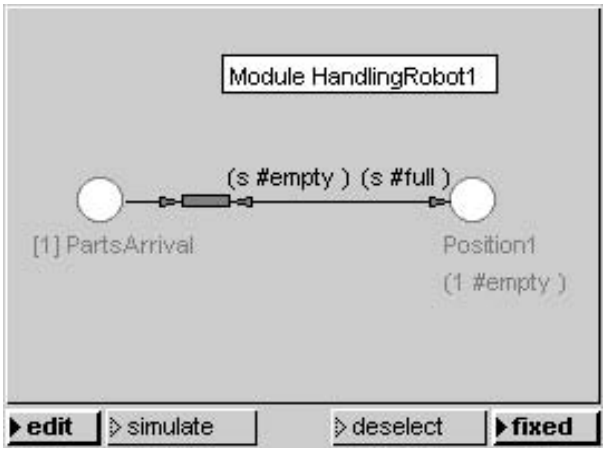

Fig. 10: Placing a part in a container

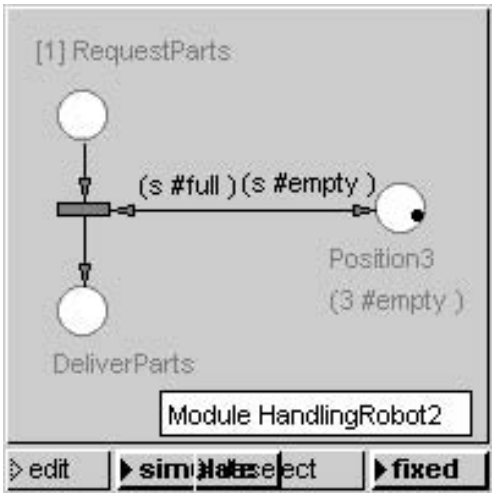

Fig. 11: Removing a part from a container

The model of the turntable shown in Fig. 12 is only a little more complicated. Here we see, a bit more subtly, the input/output interfaces "Position1" and "Position3" from the next-higher level (Fig. 7) with its initial values and the container "Position2" which is not accessible to either handling robot. The partial net shown here rotates the current attributes regarding the three stated positions.

Since the turntable switches the positions according to a time unit, the transition "tr4" fires once in each time unit and places a token in each of the places, pl1 to pl4. The token inserted in pl1 plans firing up to the next time unit. Until that point, transitions tr1 through tr3 fire, since there is a token in each of the input places of all these transitions, and they transport the tokens lying in the three positions on to the next position in each case.

This example shows on the one hand how simply and clearly event-oriented, parallel, discrete processes can be visualized with Petri nets. On the other hand it shows that the modeling tools used have general application.

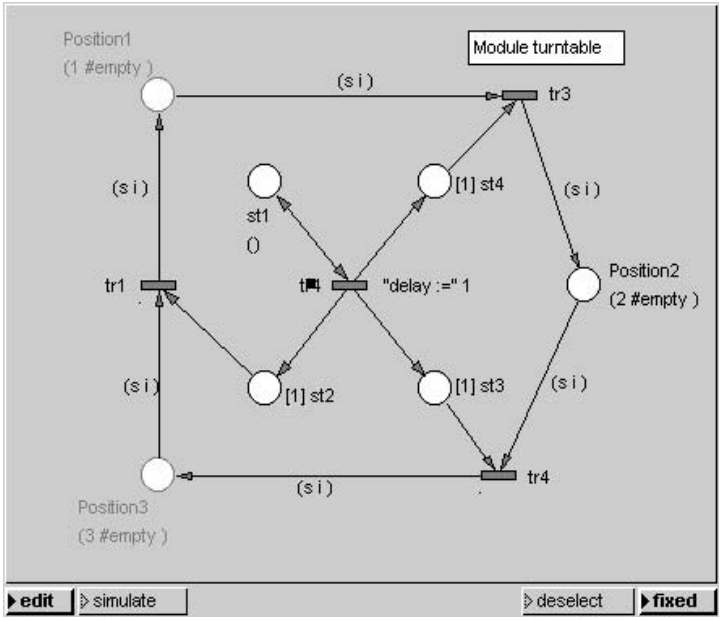

Fig. 12: Model of a turntable with three containers

4. The Simulation Phase

In the simulation phase, a correct net, even if still incomplete, can be activated to analyze its dynamic behavior. By "correct" we mean a net for which all statically possible tests (test for appropriate net construction, syntactic and semantic tests of the inscribed code) have proven successful.

4.1 Simulation Modes and Debugging Aids

The current Petri net tools usually provide two modes for executing nets.

Animation Mode

Nets can be tested efficiently if one can observe the net's execution at an adjustable speed, pausing as needed, making changes, and then resuming. During animation one can see how the tokens (or the user-defined application-oriented icons) flow from place to place. Switching occurs automatically between the individual partial nets displayed in individual windows.

For analyzing critical net sections, e.g., delays in partial processes which delay the overall process, many tools can switch to stepwise processing. This option allows a pause after each step in the simulation, so we can look at the current status of the net (contents of the places, values of attributes and variables, etc.). Some development tools also offer the option of letting the simulation run stepwise in reverse.

Background Mode

Once we have verified the principal functioning of a net, we can optimize it by varying parameters, possibly making some changes of its structure in the process. In this case it is useful to let the net run at optimal speed in the background, without animation, gathering statistical data as it runs. These data are provided to the user in tabular and/or graphical form, so he can determine whether the model can meet his requirements.

We can define interruption points for both simulation modes. Most tools permit definition of node interruption points. If a node with an active interruption point is reached during

simulation, the simulation is halted. The window containing the interrupting node is opened and activated.

Petri net tools with time modeling also allow temporal interruption points. Here a simulation is interrupted as soon as a specified simulation time point is reached. Temporal interruption points are used, e.g., to evaluate nets which must be processed within a specified time span.

4.2 Important Concepts

Although the user need not know many details of the structure of Petri net simulators, some background knowledge is useful for understanding the behavior of models during simulation. Therefore we will briefly describe a few important concepts and their background.

Simulation Time
The original definition of Petri nets prescribes the timeless switching of transitions. Thus, as soon as the conditions for firing a transition are met, the change in assignments should take place without reference to time. In classical Petri nets, the temporal sequence of events plays a role only insofar as events take place in a specific sequence, and the principle of causality applies.

Extensive articles have been written about procedures to introduce the concept of time into Petri nets (see, e.g., [4] and the literature cited there). Often the model of so-called "activation time" is used. Each transition can be assigned an activation time which indicates how many simulator time units should pass between the activation of a transition and its earliest switching. The model described in section 3.9 used activation times repeatedly.

During simulation it normally doesn't matter what the relationship is of the simulator time unit to physical time units, since only the relative behavior of the sequences is of interest. Therefore this approach allows us to simulate both very rapid or very slow processes.

Events
An event occurs when the conditions for firing a transition are met. An event consists of a transition, its input tokens and, where applicable, the current values of further simulator data which must be considered in firing the transitions. The simulator stores all events which are already completed, or which are planned for a later point in time, in an event list. This event list is a chronologically sorted listing of all events.

Each entry in the event list consists of a time indication and a quantity of events which are to occur at that time.

Simulation Algorithm
The lack of dependence on real time described above is also the reason why the time required to execute a simulation is held within reasonable limits. In his mind, the user

can assign a physical time unit appropriate to the modeled system, but this of course does not affect the temporal execution of the simulation.

All transitions that are ready to fire at a specific time fire in parallel. The simulator processes the simultaneously active firings sequentially. Once there is no longer any fire-ready transition available, the simulation time is set to the next entry in the event list.

The course of the simulation is concluded once no transition can be fired any longer, i.e., as soon as the event list is exhausted.

Simulation Protocol

The transitions fired last are noted in a list which is used during reverse runs of the simulator. Many tools allow the user to define the length of the list, and thus also the number of possible reverse steps.

4.3 Diagrams

Diagrams are useful only in nets that include time modeling. They permit graphical illustration of data generated during simulation. As a rule, tools offer bar and line diagrams which are bound to places and connectors (see Fig. 13).

The function which calculates the data to be displayed is activated each time before the state of the associated net element changes. The element itself, and the time span elapsed since the last activation, are provided by the input parameters to be evaluated for calculation of the diagram values.
The user can introduce his own functions for depicting the characteristics which change during simulation. Standardly we use the number of tokens or the value of a token attribute as markers.

Bar Diagrams (Histograms)
For every interval of the X-axis, bar diagrams show the duration on the Y-axis during which the number of tokens defined by the X-interval occurred in the net element.

For places, the change in condition of the associated net element is calculated to see how long a specific number of tokens remained at that place. The number of tokens situated on the place is shown on the X-line, while the Y-line shows the duration for which this number of tokens was stored at that place. In the case of a queue model we can thus determine, e.g., what queue lengths there are and how they relate to one another (see Fig. 17).

In the case of a connector, we can show for each interval of the X-axis how often a token whose first attribute value lies in the X-interval has passed over the connector.

Line Diagrams
In the case of line diagrams, a user-defined entity is specified as a function of time, and connected with straight lines.

For places, we often use the total number of tokens located at a place as a marker. As a default entity for connectors we often use the value of the first attribute of the tokens passing through the connector.

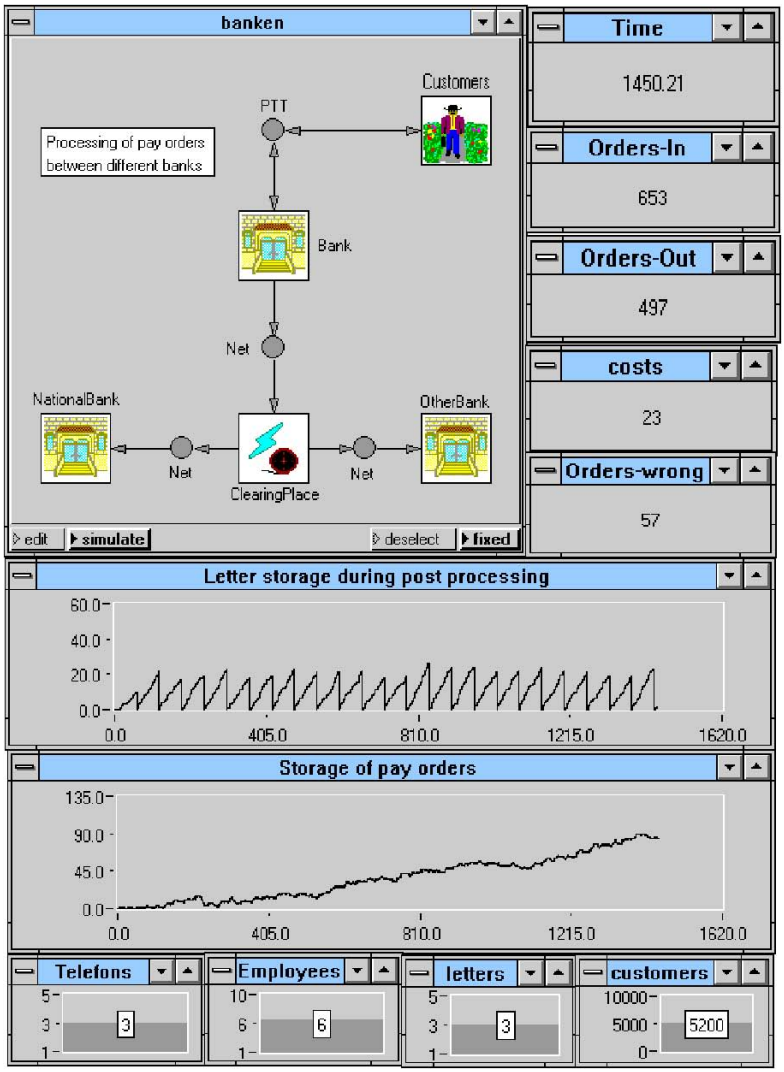

Fig.13: Uppermost level of a banking model with line diagrams and sliding bar guides

4.4 Data In- and Output During Simulation

Using inscriptions, we can easily perform data input/output via an input/output window. Smalltalk-80 from ParkPlace Systems Inc. allows comfortable and efficient input and output with predefined classes and methods which are functionally identical for all computer platforms. As an example, let's examine a Yes-No query via an input window. With the command,

```
returnValue:= DialogView
                confirm: "Do you want to continue the simulation?
                initialAnswer:true.
```

we display the input window illustrated in Fig. 14. The question is answered by clicking one of the buttons, and this assigns the selected boolean result to the variable returnValue.

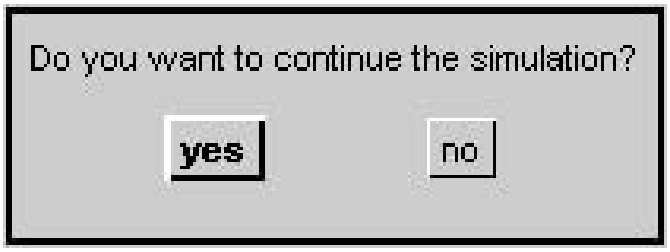

Fig.14: Example of an input window

In addition to conventional data input, some tools also offer graphical data entry by means of a bar gauge. For the net shown in Fig. 13, which depicts the

uppermost level of a banking model, there are four bar gauges under the line diagrams; with these gauges and a mouse or other pointer device, the user can vary the simulation parameters.

4.5 Simulation Examples

Let's first examine the example shown in Section 3.9 and ask the question whether, given the model's conditions, "parts-processing" will be supplied with sufficient parts. Here we could link a histogram to the place "request-parts" and then start the simulation. We see the result in Fig. 15. Most of the time, no request (token) is stored at the place "request-parts"; thus "parts-processing" is receiving an adequate supply of parts.

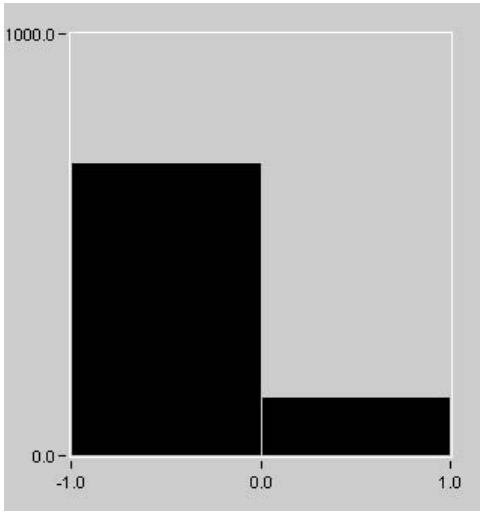

Fig.15: Histogram for „request-parts“

Fig. 16 shows a simple queue problem with one servicing location. If we assume that a simulator time unit corresponds to one minute, then on average a "customer" should arrive every five minutes. As a mean processing time, we assumed 3.33 minutes; thus a server is occupied, on average, 66% of the time. The arrival and processing times are exponentially distributed.

We are interested in seeing whether acceptable queue lengths occur normally, and therefore attach a histogram to the queue. The result appears in Fig. 17. The Y-coordinate shows the portions of the total time during which the individual queue lengths occur.

5. The Implementation Phase

Production and testing of a specification is usually followed by its implementation. For commercial applications, this consists of the adaptation of workflows and other organizational measures to the developed model. Here the development system can be of help only in those cases where it is clear that certain model characteristics cannot be realized or implemented, and that therefore deviations from the original concept must be considered. But such changes of the specification are like iterations in model development, and should therefore be assigned to the two phases we have already described.

In technical applications, implementation consists of the construction or alteration of technical installations and their associated automation programs. Here too we might require model iterations, based on what is do-able. The net specification itself, however, can additionally be used to automatically generate the automation programs.

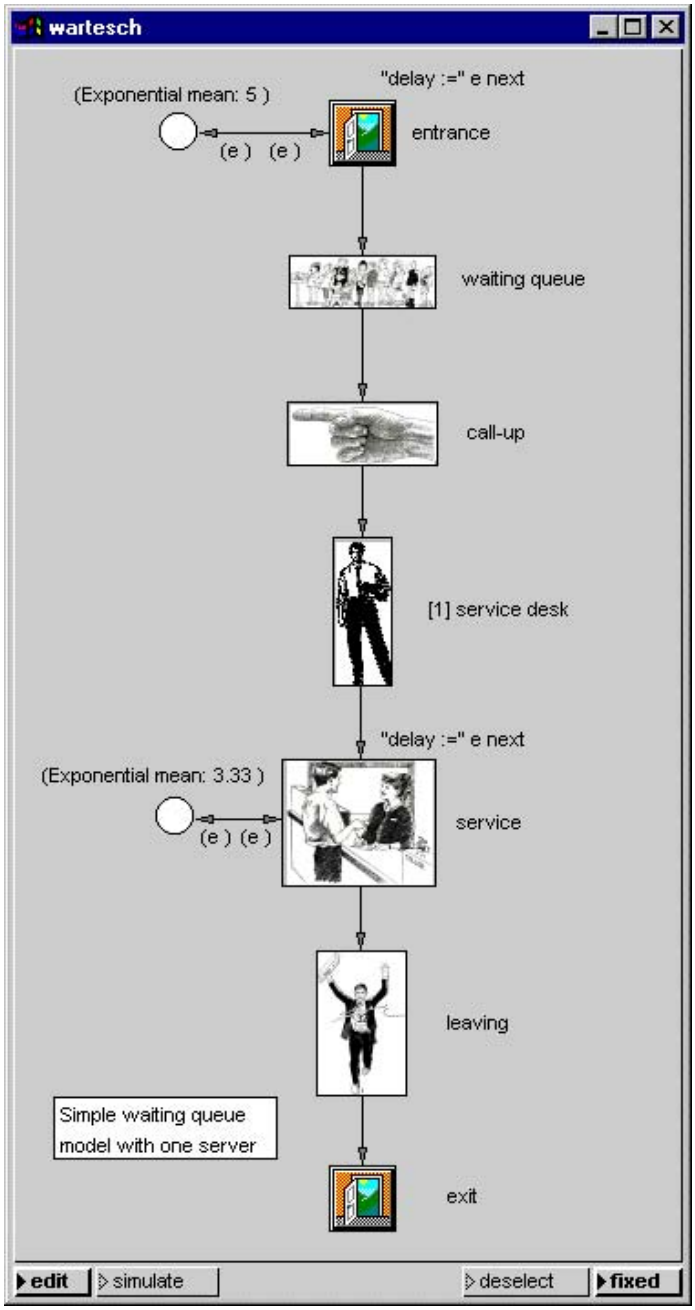

Fig. 16: Simple waiting queue model

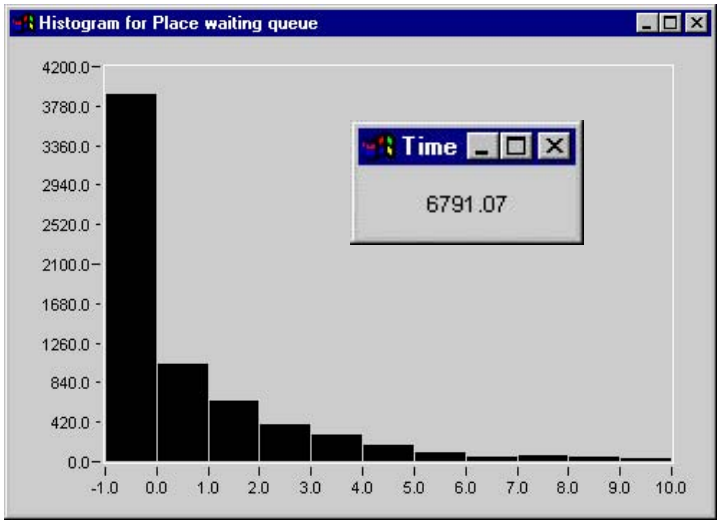

Fig.17: Distribution of waiting queue lengths

Since a Petri net models only the discrete steps within control flows, and physical processes are connected only functionally, the specification must additionally include program segments to communicate with the physical environment (e.g., I/O handlers and drivers for processing peripherals). Certain parts of the specification which were needed for simulating the environmental behavior (e.g., use of distribution functions to create stochastic events) should be replaced with commands which fix the occurrence of an event in the technical process (e.g., arrival of a part or occurrence of a certain interrupt).
There are essentially three methods of getting from the net representation to an executable program:

Generation of Programs in High Level Programming Languages
This is the most frequently used method. From the net representation, programs are produced in a high level language (usually C). With some development tools the programs must be reworked manually, at least to complete the external functions from processing input/output.

This method has the advantage that it can be used with most computers. But there are also some disadvantages we should mention. Where the program cannot be completely generated by the development tool, or, if generation is complete and one continues working directly with the generated program, experience with other CASE tools shows us that the specification and program diverge over time, and that later

on expensive reverse engineering work is needed to update the specification. Additionally, we must expect that further adaptation efforts will have to be made, since compilation systems for the various target platforms implement varying dialects and libraries.

Use of a Petri Net Machine
The preferred method can also be used with most platforms. It is based on a virtual Petri Net Machine, PNM. Programs for the PNM (which must be implemented separately for each target platform) are generated directly and completely from the net. In our case we have the PNM available in ANSI-C and can therefore use it easily on various platforms.

To make this method work, the development tool must have components which can be associated with external functions (e.g., handling programs for process devices) and which can be replaced by simulation routines during the simulation process. So that the program will run on the target platform, the virtual machine must be extended to include these external functions.

This method is just as efficiency as direct translation into source code; because even during direct generation, large parts of the generated code are data that are processed interpretively. This method is the preferred procedure for developing embedded systems and controllers.

Execution of a Net with the Petri Net Tool
The best and most elegant way to execute the net would be provided if the Petri net tool, with certain changes and enhancements (such as connection to physical time), could be used for program execution as well. As in the case of the PNM, the inscriptions which reflect the parameters of characteristics of the physical environment would have to be replaced by a second set of inscriptions by means of which, e.g., direct calls of external procedures would integrate the connections with the environment into the net.

This possiblity was rejected in the past as too inefficient. But now that even faster processors are available, it's no longer out of the question to use the development tool for the third development phase as well. This means we have fully reached the goal of a unified approach for system development.

6. Use of the Procedure in Various Application Areas

Making use of the Petri net development system PACE [2], the procedure we have described has been successfully used in recent years in many areas of industry, in governmental agencies, in research and in education. Below we shall list a small selection of the successful applications to illustrate the broad application spectrum of this procedure.

Industrial Applications
- Specification, simulation and optimization of automation projects for suppliers in the automobile and paper industries

- Specification, simulation and optimization of production lines

- Simulation of communication networks
- Simulation of automatons
- Industrial workflow analysis
- Modeling and simulation of factory operations
- Programming controls
- Analysis and optimization of production lines for relays and optical sensors
- Engineering and consulting per DIN/ISO 9000
- Specification and simulation of the CIM production concept
- Analysis and optimization of existing transport facilities
- Logisitical simulation for military airports
- Simulation of group and line fabrication
- Simulation of paint shops
- Specification of CIMOSA business structures
- Planning of flexible finishing equipment for the automobile industry
- SPS programming of milling machines
- Programming of automatic ticket machines
- Minimization of waste in gear production
- Optimization of processor structure for online image processing
- Simulation of an aluminum factory, including development of an automation program using the PACE simulator
- Modeling, simulation and optimization of a production line for solar collectors.

Administration

- Concept, review and refinement of luggage distribution in a large European airport (Zurich Kloten)
- Analysis and optimization of the materials and information flow through different divisions of a large company

- Activity processing and simulation of program execution

Commercial Applications

- Design and programming of prototypes for automatic tellermachines in banks
- Creation of network protocols
- Workflow modeling, simulation and results analysis for banks, insurance companies and public agencies.

Research, Development and Education

Numerous university institutes, technical training institutes, research facilities and development divisions of large companies use the procedure for teaching and for the most varied research programs, such as

- Research on man-machine interfaces
- Modeling of fabrication cells
- Analysis of existing production methods, and development of new ones.

7. Summary and Prospects

As has become clear above, today we can use Petri net development systems to develop systems consistently from specification through simulation, all the way to the finished automation program. The functions needed in the first two development phases are available in appropriate and efficient forms and hardly require modification. Experience shows that the available functions meet all the needs which arise during analysis and optimization of technical and commercial projects.

Compared with other procedures common today, the third phase is also very advanced. As a rule, other procedures only transform statically tested specifications into higher language source code. The generated programs must be manually reworked because often only the execution structure is generated, and because the dynamic test can be done only after program installation.

The PNM Petri Net Machine we have used will be required in the future as well, either if we need very fast automation programs, or if the development platform differs from the target platform and no Smalltalk system is available on the target system. The PNM machine offers the advantage that installation of the generated programs is simple and requires little effort. It would make sense to examine whether the PNM could be implemented as a part of a core processor.

To my knowledge, no one has yet implemented direct program execution using a Petri net tool.

References

[1] Carl Adam Petri: "Kommunikation mit Automaten," (Communications with Automatic Equipment), Dissertation, published in: Schriften des Rheinisch-Westfälischen Instituts für Instrumentelle Mathematik an der Universität Bonn, Bonn, 1962.

[2] PACE User Manual, Version 2.3, 1996
IBE, Postfach 1142, D-85623 Glonn, Germany

[3] DIN EN ISO 9001 - 9003 : 1994, DIN Deutsches Institut für Normung e.V., Beuth Verlag GmbH, Berlin.

[4] Bernd Baumgarten: "Petri-Netze: Grundlagen und Anwendungen" (Petri Nets: Foundations and Applications) BI-Wiss-Verlag, Mannheim Wien Zurich, ISBN 3-411-14291-X, 1990.

[5] J.L. Peterson: Petri Net theory and the Modelling of Systems, Prentice Hall, 1981

[6] Wolfgang Reisig: "Petri nets. An introduction", EATCS Monographs on Theoretical Computer Science, Vol. 4, Springer Verlag, 1995.

[7] VisualWorks, User's Guide, Part Number: DS 10005002, 1994.
ParkPlace Systems, Inc., 999 E. Arques Avenue, Sunnyvale CA 94086-4593.

[8] Tim Howard: „The Smalltalk Developer's Guide to VisualWorks", SIGS Books, New York, ISBN 1-884842-11-9, 1995