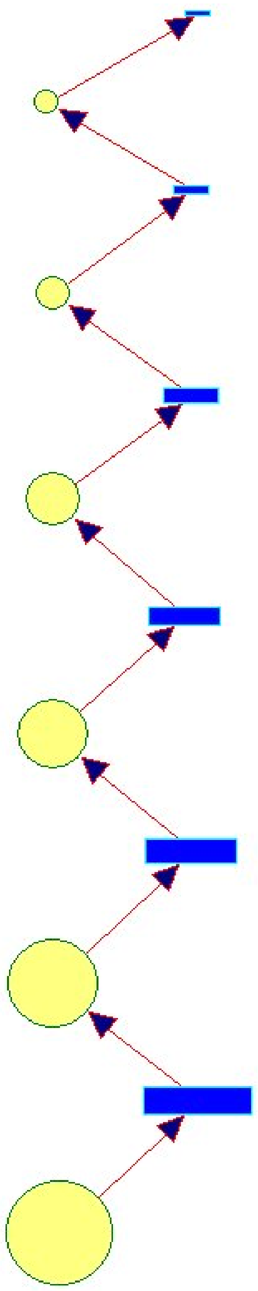


Getting Started with *PACE*

Tutorial

IBE Simulation Engineering GmbH

IBE Simulation Engineering GmbH
Postfach 1142
D-85623 Glonn
Germany
Tel.: +49-(0)8093-5000
Fax: +49-(0)8093-902687
E-mail: info@ibepace.com
Home: http://www.ibepace.com

This tutorial was developed for PACE 2008.

Table of Contents

1. Foreword

Practice makes perfect.
Proverb

1.1 General Introduction

If one purchases a new program with a complex functionality, then it usually takes several weeks until one is familiar with it and one can use it efficiently. Tutorials and manuals, and their appropriate use, become handy tools only when one has seen their operation in a number of examples and the use of their constructs has been tried in person.

The following introduction hopes to speed and ease the usual learning curve with practical examples. In these examples, PACE's many characteristics are described and used, so users have a basic knowledge of PACE after working through the tutorial and can create models on their own. The PACE manuals, which may be drawn on at any time, offer even more depth of information to allow quickly for problem-free modeling.

To ensure that the individual steps of the modeling process are understood, a short representation of net modeling is needed. Here we will attempt to show the procedures as closely as possible to real world situations, and will dispense with more theoretical discussions which are rarely or never helpful in practical applications.

To introduce net modeling and modeling and simulation with PACE we'll use a simple example, a car wash. We will take you step by step, mouse click by mouse click, through the modeling and simulation processes. Then we'll demonstrate how to make configurable, reusable building blocks or components in PACE. Finally, we'll discuss several additional useful characteristics of PACE.

After introducing the general use of PACE, we'll describe ways in which PACE assists in optimizing models. Here again, we'll start with a simple example specifically chosen to assure that the procedures to be discussed are easily seen.

PACE optimization procedures will then be discussed with a more extensive example. To keep the time and effort required for model creation to a minimum, we've somewhat simplified the task assignments. It's up to you whether you wish to actually model these rather complex optimization models or simply review them. All examples are available on the PACE-CD in the "samples\new starter" directory and, after installing PACE, may be started by double-clicking on the respective models with the extension **.imm**.

1.2 Why Do We Need Simulation Models?

Discussions on business process modeling usually deal less with the process improvements achievable through development tools, and more with the complexity of the models created. For example, at a production process seminar in Stuttgart in May 2001, the panel discussion focused on the topic, „How complex may business modeling be?“

It is generally assumed that, especially in the planning of business models, one can limit oneself to a rough modeling of relationships, and that there is no need for a detailed simulation of relationships. The following arguments are usually used:

- A simple modeling method is adequate for the assignment, as it allows for easy and flexible changes in the model.
- Simulation models requires many additional informations and are thus more complex than necessary.
- Simulation tools are too IT-intensive and are therefore not suitable modeling tools for business processes.

So today it is standard in modeling business processes to use only rough procedures in order to keep the models simple. That is not a problem, as long as models are merely used for the documentation of the processes. Unfortunately, it is common practice, very often with the use of spreadsheets, to use them for the planning of processes also.

The many provisions of spreadsheets for special applications, e.g., data in/output, cost planning and cost analysis, are not at all called into question here. However, using such a static tool, it is not possible to transparently describe, model, and completely simulate the many organizational processes and relationships which appear, for example, in production situations or in logical task assignments. Such comprehensive simulation is necessary to represent the different sequential procedures, to synchronize them, and to find and distribute the necessary resources.

Many examples, which may even be found in the daily news, show that advance planning is often not done with sufficient precision. Today many management decisions in industry and business administration are still made by gut feel due to a lack of objective decision criteria. The damage created by bad decisions can hardly be estimated, and only on rare occasions (e.g., in the public domain through audit agencies) is it registered and evaluated.

The standard example for inadequate planning is the often-described Denver airport disaster, in which huge sums were wasted in cost overruns and follow-up costs.[1]

[1] While reading this tutorial’s correction a further great civilian disaster of this kind takes place: the new terminal of London Heathrow. According to the reports on the occurrences one must assume that no sufficient modeling and simulation was carried out

Even more recently, six months after the major reorganization of a large German corporation, the press asserted that the reorganization had not had the desired results and that the company was planning in part to undo it. Using modern simulation methods in the early stages of the reorganization, one could at least have avoided the enormous follow-up cost. One could have studied the target organization before implementing changes, and would have known in advance if such organizational changes can lead to success or if perhaps better alternatives exist.

For successful planning of business processes, one needs exact simulation models which provide a good foundation for difficult decisions and planning. These models then have to be as complex as they must be to solve this task satisfactorily.

This does not mean that the models must be time- and cost-intensive, but rather that a good model requires an appropriate tool and appropriate expertise. Usually, the costs of modeling and simulation are quickly amortized through the smooth and resource-optimal running of processes which would otherwise be achieved only by iteration, if at all.

This iteration can become very expensive. So, when a large German company introduced a new IT product, it required three years to ensure that production and service would work seamlessly. During that time, there were many problems with the distribution of resources, and many emergency measures were required to eliminate bottlenecks, not to mention that the clients were not happy and began to consider other options. Many of these issues could have been avoided if the planned reorganization had been examined in advance by use of a simulation model.

Finally here, a small example: To create the simple production optimization used in Chapter 7 of this Tutorial from scratch, a worker familiar with PACE requires approximately two eight-hour working days. At an hourly rate of 125€ for an IT professional, that puts the cost at 2,000€. If one looks at the simplest case, where just one worker too many is planned into a job, the weekly cost (50€ times a weekly work time of 40 hours) is also 2,000€. Thus the creation of the simulation model would be amortized in just one week!

here either. How can one explain the reports on an unsatisfactory resource availability (e.g., for the transport and selection of luggage or for the check-in of airline passengers) differently otherwise?

1.3 The PACE Methodology

The PACE methodology is based on the premise that one can achieve realistic statements of the relationships in realtime systems if one maps the systems in programs which behaves exactly as reality, i.e., that they show an identical behaviour as well with respect to the processes as with respect to the occurring process data. If that is done, one can use these programs to run cost-effective experiments to determine how outside influences and environmental scenarios affect reality, and what the optimal parameters are for certain criteria (development times and costs, production costs, resources, etc.). This tutorial will describe how realtime systems may be translated into simulation programs in an engineering-friendly way.

Before we discuss how the suggested methodology is implemented, we will sketch out the difference from other methodologies so the differences may be better understood. Older methods usually express the relationships in the system to be modelled in mathematical equations. These are computerized using programming in a simulation language (e.g., GPSS, Simula) or via a calculation tool (e.g., spreadsheets). Other methods lean more in the direction suggested here, and model their processes without including the characteristics required for reality-based execution of the processes (e.g., flow diagram tools, CASE tools, component systems). While flow diagram tools and most CASE tools can only represent the static aspects of process systems, with the dynamic aspects only perhaps added in the form of comments, component systems at least partially consider dynamic aspects. The model of a process system is built up with pre-made components or modules, which can be matched with a set of parameters more or less good to real world conditions. Each module is assigned an algorithm which is executed when the module is called up. A true simulation as in PACE, in which objects run through the virtual system in a realistic and animatable fashion, does not occur. The problems in using module systems will be briefly discussed at the beginning of Chapter 4.

Important in PACE methodology is to statically as well as dynamically provide a detailed and exact picture of the parallel activities (processes) as well as the relationships of the objects being worked upon. The required degree of detail is based on the precision of the desired results.

Examination of the descriptive means of representing realtime systems led to three types of language elements and to the development of a semi-graphic modeling language, MSL, which is a component of PACE. The structural agreement between reality and model is achieved in PACE by translating the visual structure of an application one-to-one to the model, using a graphic editor. To ensure that the model is functionally identical, it must be executable, i.e., it must be set up as a simulation model. In the three types of language elements, this means:

- Language elements to describe the structure of an application, including process ways in the application.
- Language elements to represent the objects moving on the process ways.
- Language elements to process the data of the objects and other process data.

Language elements for representing process ways can be taken from Petri nets. In PACE, the processing steps are described with generalized Petri-net elements. The generalization is found in envisioning Petri-net elements with multiple attributes which set the exact processing of objects at net nodes (junctions). When applying attributes, for example, some net elements (so-called transitions) are provided with program code which pictures the processing of a real object.

In addition to the process steps, the system structure of the real system must also be mirrored in the model. For that reason, PACE includes two additional net elements, modules and channels, for the hierarchical structuring of nets.

After the structure and the process ways are represented, the dynamic objects which operate on these process ways must be introduced. In the model, they are represented by tokens or placeholders which run through the net on the process ways and which carry the characteristics of the real object needed in the model as attributes (arguments or parameters). The object data is changed by running through the net using the program code mentioned above, ensuring that the processing, i.e., the changing of objects, represents a true and realistic picture.

Finally, a script- or programming language must be chosen, to specify all net inscriptions (program codes, names, comments, etc.). Since PACE itself is largely written in the object-oriented programming language Smalltalk, it seemed appropriate to avoid interface problems by using the same language, or a subset thereof, as the inscription-language. This provides for working with model data in an easy-to-use language with an extensive method library.

These procedures fulfil two further important requirements of model development and model change. They must provide simple test opportunities and must be flexible. Both requirements make creating and using models considerably easier and faster. Through step-by-step representation of the processes, one can use a debugging mode to quickly determine (if necessary, with the help of an expert who knows the modelled processes well) whether the model behaves realistically by analyzing the parallel process steps as they run and the data they generate.

2. Basics

2.1 Net Description of Discrete Processes

Computer-supported simulation of business processes is in its infancy today, and only few programs are able to represent the complex processes to be analyzed in a complete and reproducible way. The PACE program, with its mature semi-graphic **M**odeling and **S**imulation **L**anguage **MSL** which has been continually refined over more than 15 years, has proven itself in modeling, simulating and optimizing processes in both industry and research/development. It is based on the net description of information streams invented in 1962 by Carl Adam Petri, and was expanded with many descriptive elements for detail-exact modeling of business processes. This includes the ability to convert net nodes into processing objects by entering code, includes graphical and statistical libraries, fuzzy-logic, methods for visual and automatic net optimizing, modules for repetitive task assignments, and many opportunities for simple data input and results visualization.

Figure 2.1: Overview of the Car Wash

Petri nets are derived from the concept that most processes can be broken down into discrete individual steps which can be represented in the form of nets (directional graphs).

To derive the elements of a classic Petri net, the following example will use a simple procedure, namely the washing process in a car wash, and decompose it into discrete individual steps. We are not interested here in the washing of the vehicles itself, but rather only in the behavior of the car wash under various customer loads. Thus, we will assume the car itself is an unchanging object which moves through the wash queue, changing the status of the wash line.

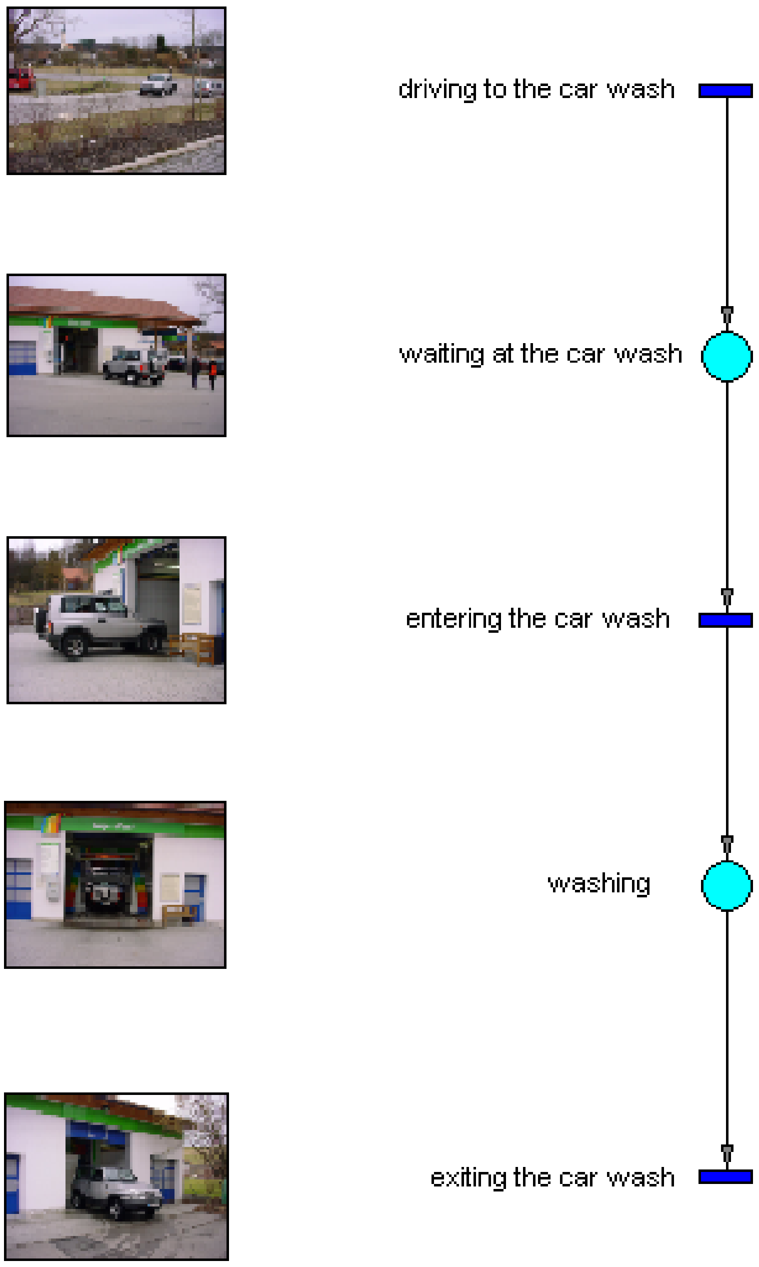


Figure 2.2: Breaking down the washing procedure into individual steps

Figure 2.1 shows an overview of the car wash. Cars arrive at the line from the right and, depending on whether a vehicle is currently being washed or other vehicles are already waiting, may have to wait. After washing, the vehicle leaves the car wash at the back of the building.

Figure 2.2 shows the wash procedure decomposed into five discrete individual steps.

While the process steps ’driving to the car wash’, ’entering the car wash’, and ’exiting the car wash’ change the status of the car wash, the same is not true for the process steps ’waiting at the car wash’ and ’washing’. One can divide the washing process into three actions and two waiting positions. The simple Petri-net shown on the right side of Figure 2.2 is created if you assign a small square as the symbol for an action (**transition**), a small circle to symbolize a waiting position (**place**), and show the direction of movement through connecting arrows (**connectors**). Our net is not complete yet, however. For example, we have not yet expressed that the cars to be washed arrive in statistical distribution, that only one vehicle can be washed at any one time, and how long the wash procedure takes. These additional entries are set in PACE by adding attributes to net elements (see Section 2.3).

A Petri-net is executed in a stepwise fashion, in which the object to be processed (represented by a small solid circle or token) is picked up from an entry holding zone (input place), is processed in a transition, and then is sent to an exit holding zone (output place). For this reason, places and transitions must alternate.

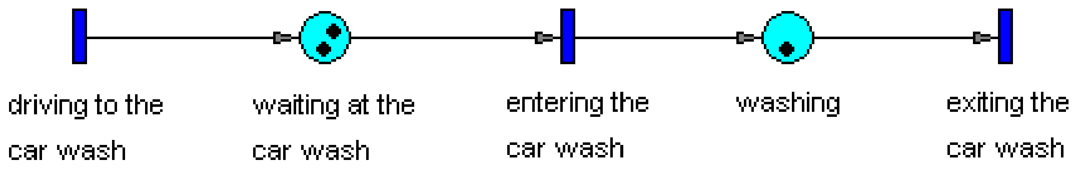


Figure 2.3: Car wash with two waiting vehicles and one vehicle being washed.

Figure 2.3 shows the Petri net from Figure 2.2 with two waiting vehicles in the entry zone and one vehicle currently being washed. The transition ’driving to the car wash’, which has no entry connector, constantly creates tokens (cars), while the transition ’exiting the car wash’, which has no exit connector, removes the cars from the net again. If we assume that the place ’waiting at the car wash’ can take up any number of tokens, but only one can be stored in the place ‚washing’, the next step would be to switch (or fire) the transition ’exiting the car wash’ as soon as the time for the washing process has elapsed. Then the transition ’entering the car wash’ can fire, the next vehicle is transported from the place ’waiting at the car wash’ to the place ’washing’, etc.

2.2 Simple Nets

As a basis for later descriptions, in this section we’ll present several characteristics of classic Petri nets.

Marking

As the marking of a Petri net the token occupancy of its places is used. It changes from the initial setting of tokens during the running of a net from step to step. The current status of a net is defined by the current token occupancy.

The initial marking of the net represented in Figure 2.5 is a single token on the place 'car wash is empty'.

Switching Rules

The preceding description of a car wash specifies that the place 'washing' may have only one token, while the place 'waiting at the car wash' may have any number of tokens. In classic Petri nets, this requirement is realized using a so-called switching rule. Another option would be to specify the capacity of a place, which may later be accessed.

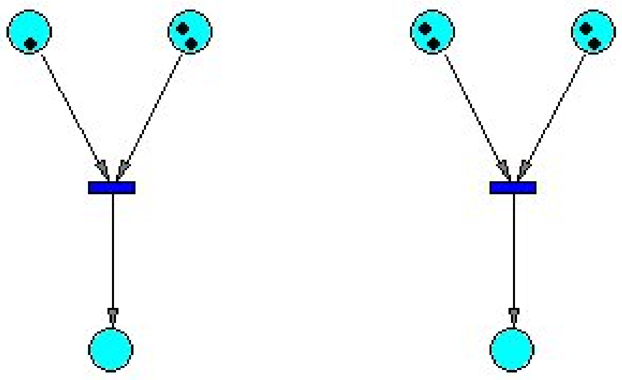


Figure 2.4: Examples of switching rules

The switching rule controls the situation when a transition has two or more input places, and says that the transition may only fire when all input places have at least one token. When it fires, one token is removed from each input place and used by the transition. A token is then generated at each output place. A transition whose input places each contain at least one token is designated as 'activated'.

Figure 2.4 shows two nets with multiple input places. The transition in the left-hand net can fire exactly once, as one of its input places carries only one token. The transition in the right-hand net can fire twice, because there are two tokens in each input place.

Using the switching rule, we can specify that the place 'washing' in Figure 2.3 can always contain only one vehicle. To accomplish this we need to expand the net from Figure 2.3 as shown in Figure 2.5. We've added an additional place, 'wash is empty' to the net elements from Figure 2.3. We've also assumed that the place 'washing' and the waiting queue 'waiting at the car wash' are empty.

Since the transition 'driving to the car wash' has no input connector, its firing is not condition-dependent. Consequently, it can fire first and send a token to the place 'waiting at the car wash'. The transition 'entering the car wash' can then fire, because there is now a token in both of its entry places. When it fires, a token is removed from each input place, and one is added to the place 'washing'. The transition 'entering the car wash' cannot then fire again, because there is not a token in each of the two entry places. In the place 'washing' there can be at most one token at any given time. The transition 'driving to the car wash' fires again and sets a new token in the place 'waiting at the car wash'. After washing, the transition 'exiting the car wash' switches and sets a new token in the place 'wash is empty'. This makes the transition 'entering the car wash' ready to fire again, etc.

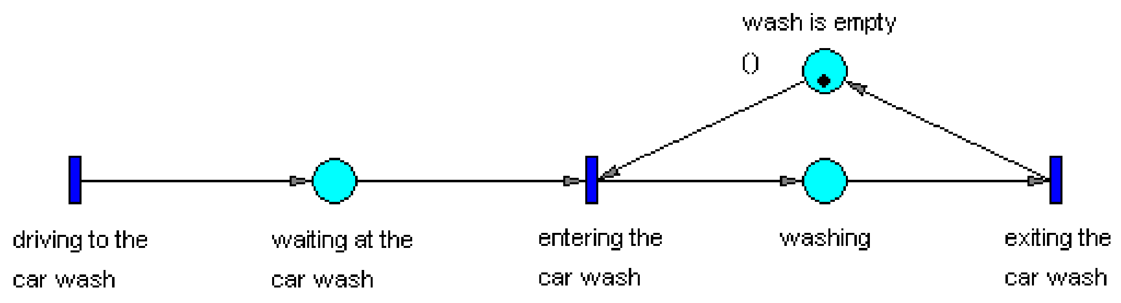


Figure 2.5: Car wash with control of the washing booths

Double Connectors
In most cases, two net elements are connected in only one direction, that is, with a simple connector. Such net elements are designated as 'pure' elements. If the net consists only of pure net elements, the net itself is designated as 'pure'. An example of a pure net is shown in Figure 2.5.

A simple case of two net elements doubly connected with a double connection would be if any number of tokens are to be produced. In the net shown in Figure 2.5, we accomplished this with the transition 'driving to the car wash', which has no entry connector, i.e., firing conditions, and consequently fires continuously. The same can be accomplished with the net shown in Figure 2.6, which uses a double connector.

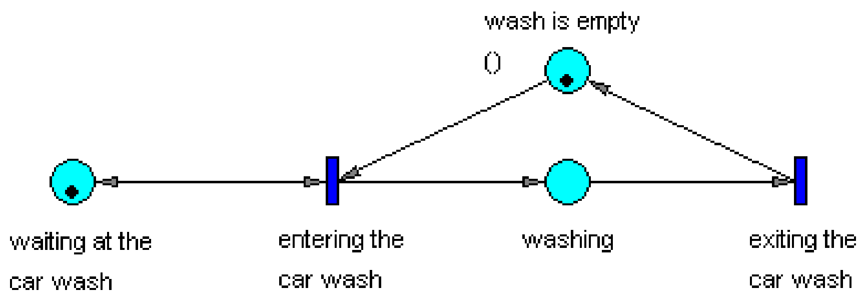


Figure 2.6: The wash street using a double connector

When the transition 'entering the car wash' fires, a token is also placed in the place 'waiting at the car wash' to represent the next vehicle to be washed.

Reversibility
A net is reversible if the inital marking of the net can be recreated from any desired later markings.

All the example nets we have looked at so far are reversible. Most of the attributed Petri nets we will consider later are irreversible.

Inhibitors
Besides the normal connectors, there are inverting connectors (inhibitors) at which the transition just then may fire if no token lies on the place which is connected to the

transition with the inhibitor. Inhibitors are represented by a little full circle on the arrowhead of a connector. While tokens flow over normal connectors, this is not true for an inhibitor. It only steers the firing of the transition.

Even with just one inhibitor, it is easy to ensure that the place 'washing' can store at maximum only one token. In Figure 2.7, the wash queue represented in Figure 2.5 is modelled using an Inhibitor.

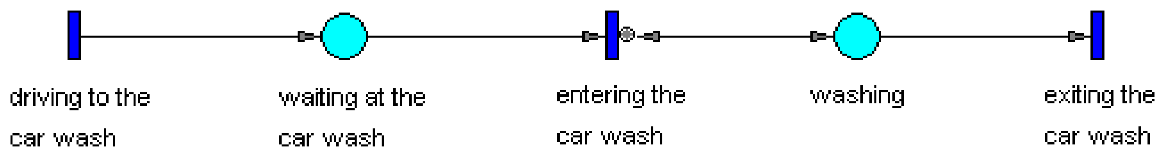


Figure 2.7: The wash queue using an Inhibitor

2.3 Attributes of Nets

In order to accept the example nets we have shown as reality-proximate models of a car wash, at least two additional characteristics must be present:

1. The statistically distributed arrival of the vehicles to be washed must be considered.
2. The duration of a washing must be specified in the model.

Such additional conditions are handled in PACE through net attributes.

The two above requirements, along with the requirement that no more than one vehicle may occupy the place 'washing' at any time, determines the capacity utilization of the simple wash queue we're modeling, and can be fulfilled by adaptation of single net elements. The first requirement can be fulfilled with the transition 'driving to the car wash', since that generates the tokens, i.e., the cars to be washed. The condition that only one token may be stored in the place 'washing' is obviously a characteristic of that place. The second requirement can be handled as a delay in exiting the car wash, and consequently impacts only the 'exiting the car wash' transition.

Attaching attributes to net elements in PACE is done using net inscriptions or, inscriptions for short. These are programming texts which are attached to the net elements and evaluated later in the execution of the net (simulation). Inscriptions, which have been used in all examples so far in this tutorial, are also the names or identifiers of net elements.

Figure 2.8 shows the car wash model with attributed net elements.

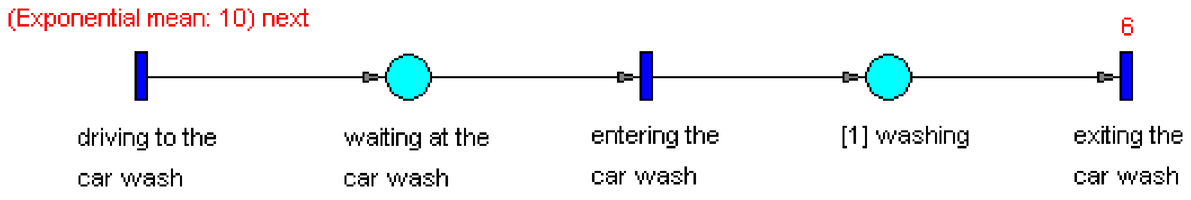


Figure 2.8: Car Wash with Attributes

Arrival of cars at the car wash was modelled with an exponential distribution with a mean value of 10 minutes, which provides the statistically distributed time spans between the arrival of two vehicles. The command is executed with each firing of the transition ’driving to the car wash’ and provides the time delay until the next firing of the transition. The ’washing’ place was assigned a capacity of 1, which appears in square brackets in front of the place’s identifier. This has the same effect as the steps described in the last section to limit the maximum number of tokens to 1. Finally, the exit from the car wash is delayed by 6 minutes, which means the washing procedure is intended to last 6 minutes.

All together, this represents a simple model of a car wash in which several dynamic aspects are also considered. The model can thus be used to examine the condition of the real car wash in operation.

An interesting point here is the place ’waiting at the car wash’, in which the queue of vehicles is stored before the washing booth. To get an impression of the conditions of the car wash in the anticipated arrival of vehicles, it is useful to attach a time histogram to the place ’waiting at the car wash.’

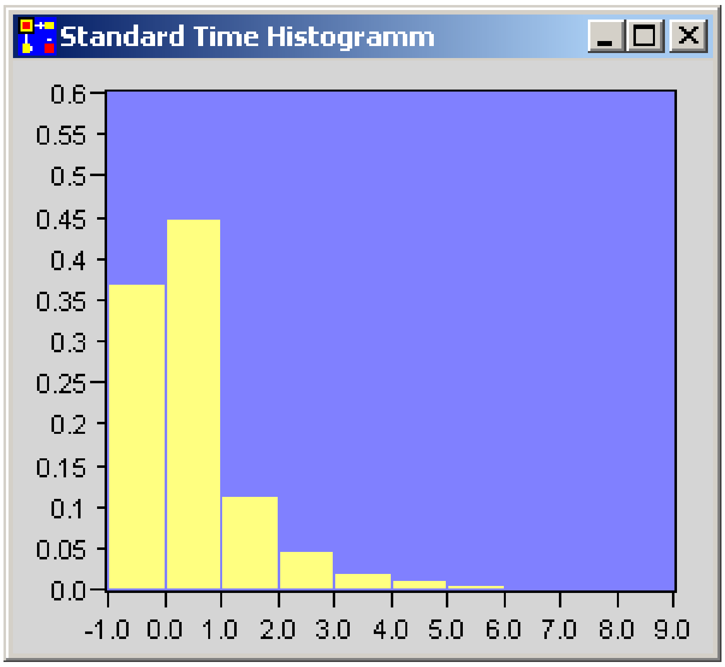


Figure 2.9: Distribution of different waiting queue lengths

If you execute the net, the result will be as pictured in Figure 2.9. The bars represent the number of vehicles shown in the abscissa to the right of the bar. The bar ordinates assign the percentage of operating time in which the car waiting queue contains that number of vehicles. You can see from the illustration that only rarely do more than two vehicles sit in the queue, and that the waiting queue is empty about 36% of the operating time. It is easy to integrate additional system requirement specifications into the model, such as different numbers of vehicle arrivals in different day parts and weather conditions, different washing programs with varying wash times, the behaviour of drivers who simply drive by if more than one vehicle is already waiting, or industry-specific requirements/conditions.

Even the simple nets we've reviewed so far show, that to see the correspondence between a net and the reality can be difficult. In many nets, it is not easy to see exactly what they describe. This difficulty can be greatly reduced by using appropriate names for net elements and substitution symbols (graphics) for individual net elements. Figure 2.10 shows a snapshot of a net window during simulation, in which the net elements have been replaced with graphics (icons). The background is a picture of the entire car wash.

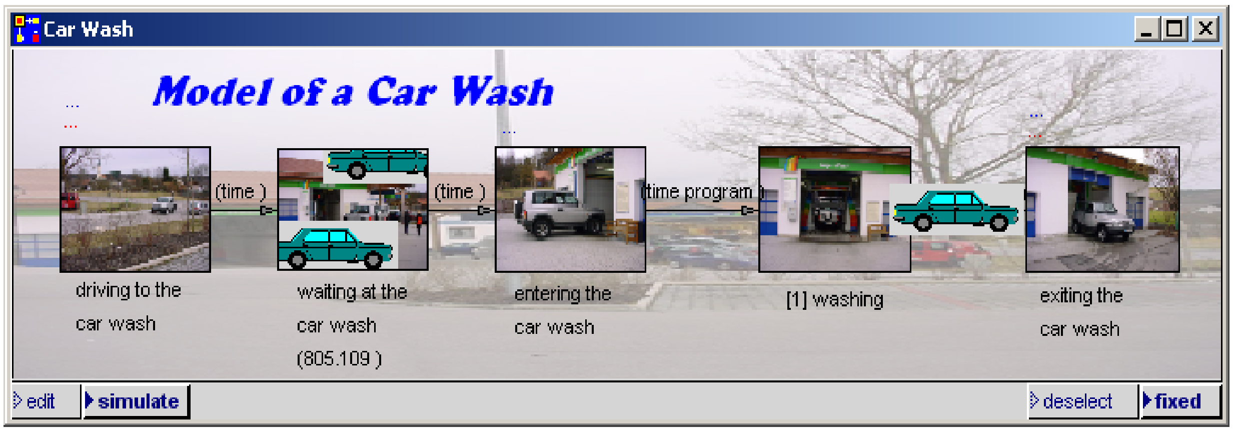


Figure 2.10: A picture is worth a thousand words.

With the three elementary static net elements transition, place and connector, an application can be pictured as single process steps. If, for example, data are processed, they flow from one input place, which models a storage space, to the transition, in which the actual processing takes place, and after processing are stored in the output place, which again represents a storage space. Similarly, processing of products by machines, production runs on assembly lines, or the processing of administrative procedures in bureaucracies can be illustrated using attributed nets. You can look at an attributed net as a runable virtual model of the pictured real system, and use it for analysis and further development of the real system.

3. Modeling a Car Wash

3.1 The Basic Model

PACE is used with a three-key mouse. For most PCs in use today the middle mouse key is a wheel which can also be used for clicking. To simplify the following descriptions, the mouse keys will be identified as follows:

le.MK	left mouse key	Selects or marks an object
mi.MK	center mouse key	Shows the system menu
ri.MK	right mouse key	Shows the context-specific menu

If you use the touch pad of a notebook with two buttons, normally the left button corresponds to the **le.MK**, the right button corresponds to the **mi.MK** and the right button together with the control key of the keyboard corresponds to the **ri.MK**.

After installing PACE, start the program using the Windows menu bar or call up Windows Explorer, go to the PACE installation directory and double-click with the **le.MK** on the PACE icon **pace2008.imm**.

After the start of PACE, the follow menu bar will appear on the monitor:

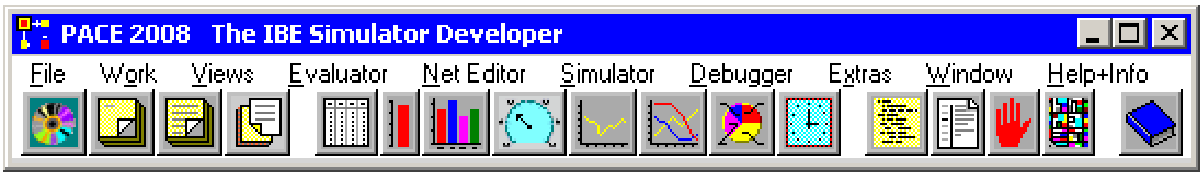


Figure 3.1: PACE Menu Bar

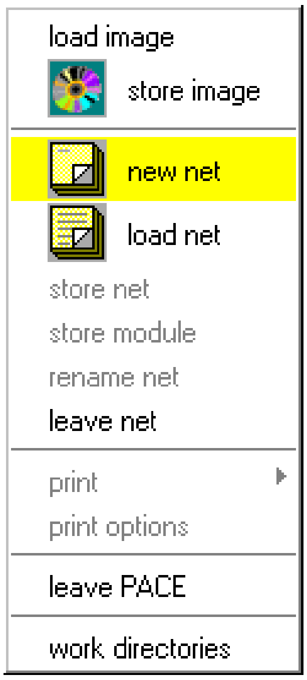


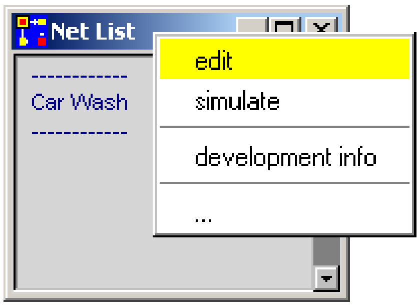
Figure 3.2: File Menu

This is the starting point for model creation with PACE. The menu bar is accessible during all phases of work. You can use it to adjust, among other things, all system wide settings.

To create a new model, click on **new net** in the File menu. This opens a query window in which you will enter the name of the new net. ’newNet’ is the default name. After inputting the model name **Car Wash**, press return. A frame will open for a window, the so-called net list, in which the modules of the net will later be listed. Move the window to your preferred location on the monitor, then click and hold the **le.MK** key. The cursor will move from the upper left corner to the lower right corner. This allows you to use the mouse to adjust the net list window to the desired size. Then release the key. The results are shown in Figure 3.3. In the case at hand, the list includes only one line with the name **Car Wash**.

The File menu includes general file-related functions, such as loading a net (load net), saving the current net (store net), etc.

Use the **le.MK** key to click on the line in the net list (select the line), use the **ri.MK** to open the net list menu, and select **edit**. The net window for the **Car Wash** will open in editing mode (Figure 3.4). The size of this window may be changed as usual in Windows. The net for the car wash can now be modelled in this window. Modeling is done by setting the elements needed for the net under development. These are entered interactively with the graphic PACE editor, which has menus available from which net elements may be selected.



3.3: Net-List and Net-List-Menu

In the net window (that is, with the cursor in the net window) open the **No-Selection-Menu** by pressing the **ri.MK** key. As its name indicates, it is called when no net element in the window has been selected (at the moment, none exists yet). After selecting the menu option **tran-**

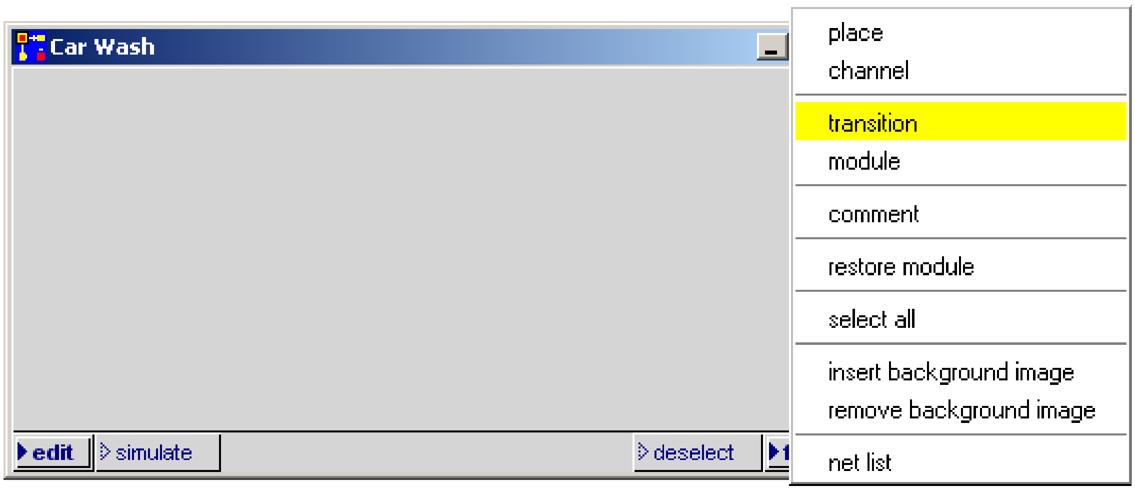


Figure 3.4: Net window and No-Selection-Menu

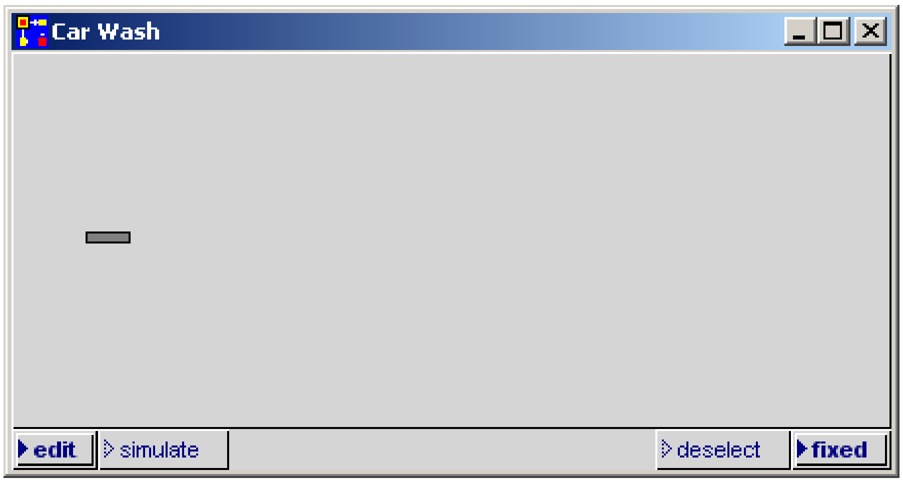


Figure 3.5: Net window after fixing the Transition

sition, a symbol for a transition will appear at the cursor position. Move the mouse to place it where you want it. Fix it in position by pressing **le.MK** (Figure 3.5).

Similarly, the remaining net elements are inserted in the window, using the menu option **place** for places. If you select the wrong net element by mistake, you can delete it prior to fixing by pressing **ri.MK**. After fixing, a net element can be deleted by selecting it with the **le.MK**, then choosing the net element window with the **ri.MK**. There select the menu option **delete**. We will discuss net element menus in more detail later. The result is represented in Figure 3.6.

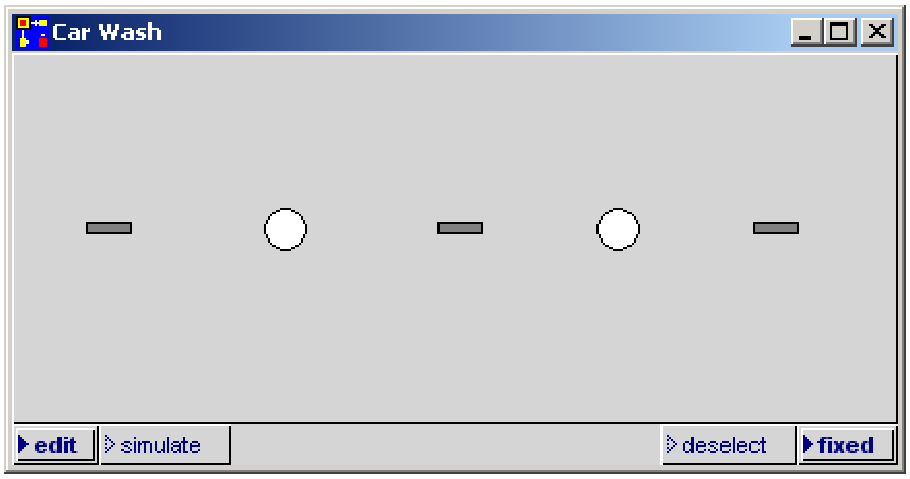


Figure 3.6: Net window without connectors

If you are unhappy with the position of a net element and want to move it to another location in the net window, mark it with the **le.MK**. Then click again on the net element with the **le.MK** and hold the mouse key down. Move the element to its new spot with the mouse key held down, and then release the key.

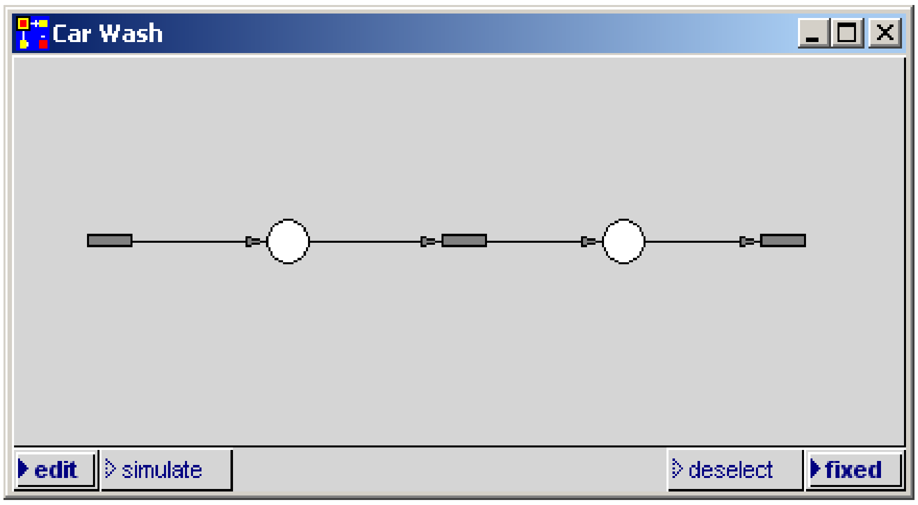


Figure 3.7: Net window with connectors

Next, set the connectors. Position the mouse cursor on one of the net elements, press the **le.MK** and hold it. With the key held down, move the mouse cursor onto the net element which is to be connected. As mentioned earlier, places and transitions must alternate in the net. A connector is set and identified only if this condition is met. The result is shown in Figure 3.7.

If you like, you can improve the look of the net by joining the connectors not at the narrow sides, but at the wide sides of the transitions. The standard transition icons must be turned 90 degrees to do this, as follows:

Select one of the transitions with the **le.MK**. Use the **ri.MK** to highlight the transition menu, the submenu option **Icon**, and finally the menu option **alternate**. When this is done for all transitions, the net window will look as shown in Figure 3.8.

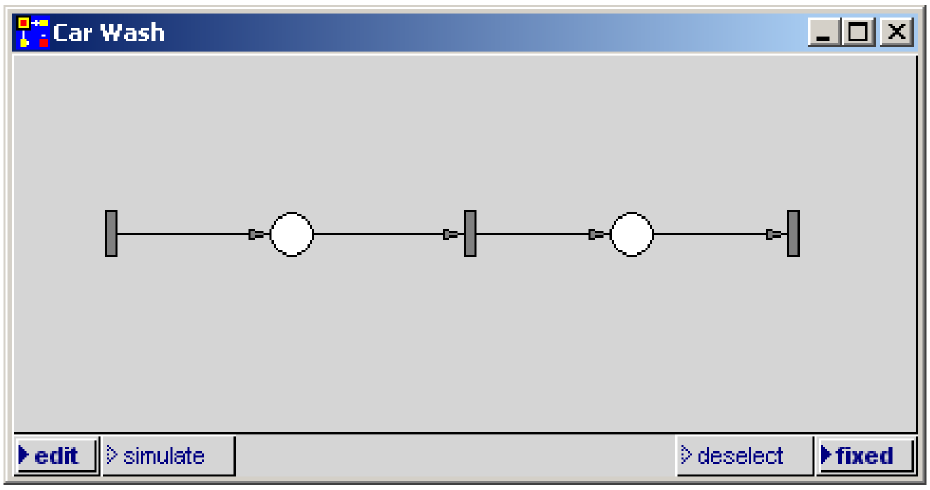


Figure 3.8: Net window with alternative transition representation

Next, attributes need to be added to the net.

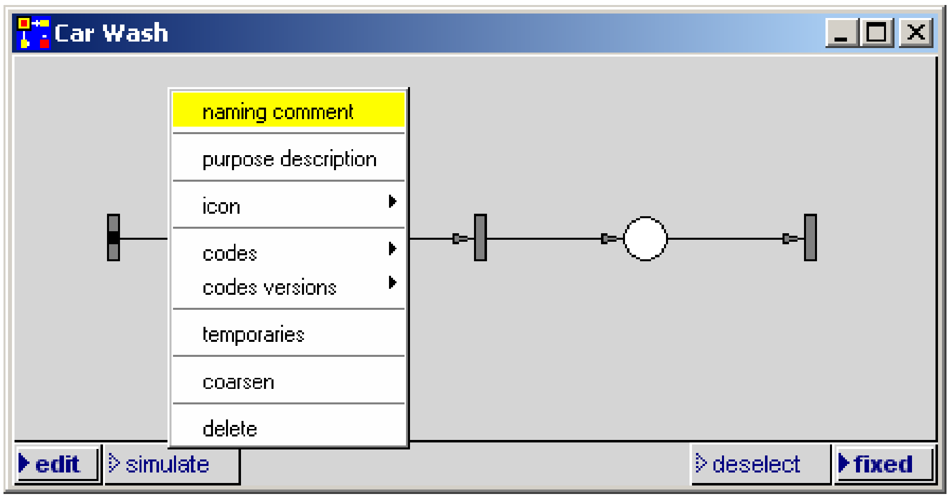


Figure 3.9: Attributing of net elements

Identifiers for individual net elements are inserted as follows: Mark the selected net element with the **le.MK** key (a black dot is drawn on the net element) and use the **ri.MK** to access the net element's menu. Select the menu option **naming comment** (Figure 3.9). It opens the input window for net element identifiers shown in Figure 3.10.

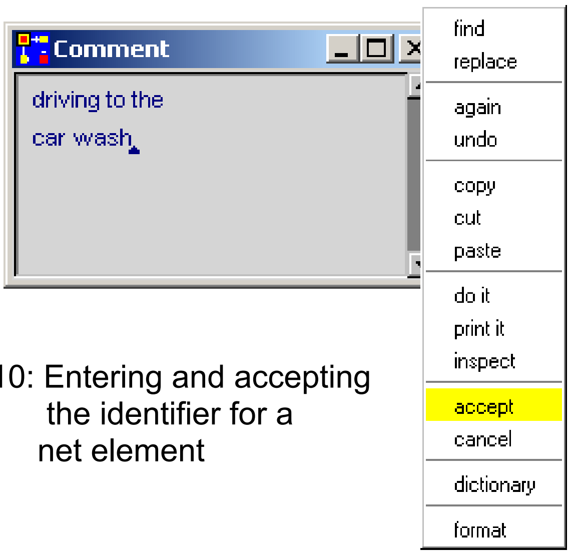


Figure 3.10: Entering and accepting the identifier for a net element

After inputting the identifier, which may run across several lines like a comment, use the **ri.MK** in the menu of the text window and select **accept**. This assigns the identifier to the selected net element, and it is represented in the net window (Figures 3.10 and 3.11).

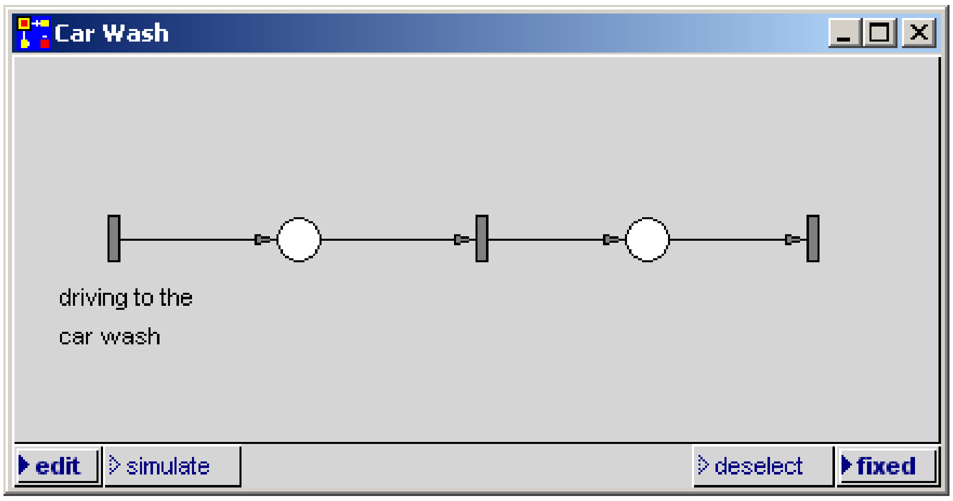


Figure 3.11: Net window with an identifier for a net element

To move an identifier (the “naming comment“) to a different location in the net window, mark it with **le.MK**. Then click on the identifier again with the **le.MK** and hold the mouse key down. Move the identifier to the new position with the mouse key held down and then release. All the identifiers from Figure 2.3 can be entered and positioned in this way. You may wish to enlarge the window so all texts can be represented in the proper locations (Figure 3.13).

If you notice a typo or spelling error later, you can correct it by marking the identifier, pressing the **ri.MK** key and selecting menu option **inspect**. The text input window will open again with the text to be changed. After correcting it, select the **accept** menu option. By the way, if you choose menu option **owner** instead of **inspect**, the mouse cursor is positioned on the net element associated with the identifier. This function is occasional very useful in larger nets with many inscriptions.

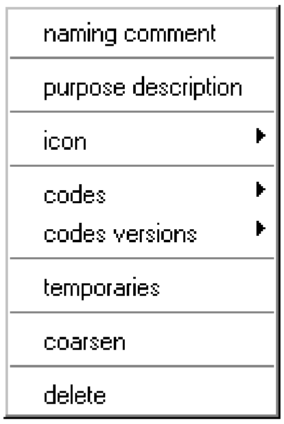


Figure 3.12: Menu for a transition

Next we'll enter the attributes shown in Figure 2.8. Mark the transition 'driving to the car wash' with the **le.MK** and select the transition menu with the **ri.MK** key (Figure 3.12).

The menu includes the menu option **codes** with three submenu points, with which a transition may be matched to the task at hand.

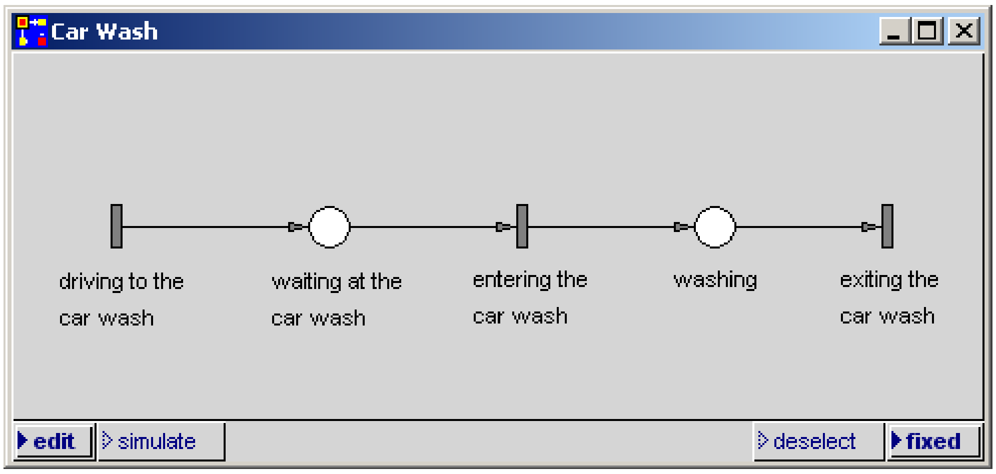


Figure 3.13: Net window with identifiers for all net elements

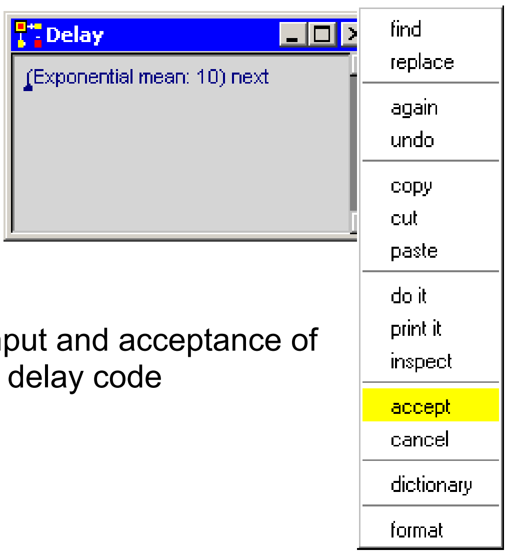
A transition can be refined with the following three code insertions:

condition code: (green) Here you can formulate a condition under which the transition may fire. The code insertion must deliver a **true** or **false** value.

delay code: (red) The code insertion calculates the number of time units by which the firing of the transition is to be delayed.

action code: (blue) To represent a real object in a PACE model, the data representing the object is attached to the virtual object (the token). The action code expresses the processing of the real object, as it calculates the data of the exiting tokens from the data of the incoming tokens.

In the case at hand, only the delay code needs to be assigned. After selecting the submenu option **delay code**, an input window will open to receive the programming text (Figure 3.14).



(Exponential mean: 10) next

Figure 3.14: Input and acceptance of the delay code

The text to be entered is a Smalltalk message. The bracketed expression **(Exponential mean: 10)** defines the exponential distribution with a mean value of 10 time units. The next value of the distribution is calculated with the unary operator **next**. By choosing the menu option **accept** the text is added to the net element, as shown earlier with the 'naming comments'. If the text is to appear in a different location in the net window, move it as discussed earlier with a 'naming comment' (see above). Use the same process in the transition 'exiting the car wash', where you only need to enter the number 6.

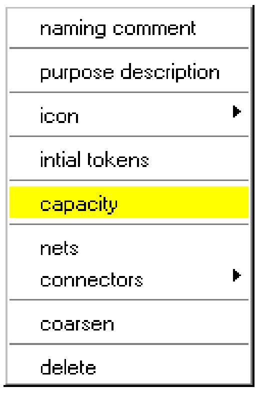


Figure 3.15: Menu of a place

Assignment of simulator time units to physical time units is handled by assigning all time specifications in the model in the same dimension; in this case, all are in minutes. This implicitly sets the physical dimensions of the simulator time unit.

To set the maximum number of tokens which may be stored in the 'washing' place to 1, mark the place with the **le.MK** and choose the menu of the place with the **ri.MK**. If you select the menu option **capacity**, an input window will appear for entering the acceptable maximum number of tokens.

When the window opens, a default value of 0 is assigned. This value means that any number of tokens may be stored. Replace the value 0 with the value 1 and press **return** key of the keyboard. The maximum number of tokens will appear in square brackets in front of the identifier 'washing' in the net window.

The net for the car wash is now done with the exception of the results output and possible improvements by inserting icons (see Figure 3.16).

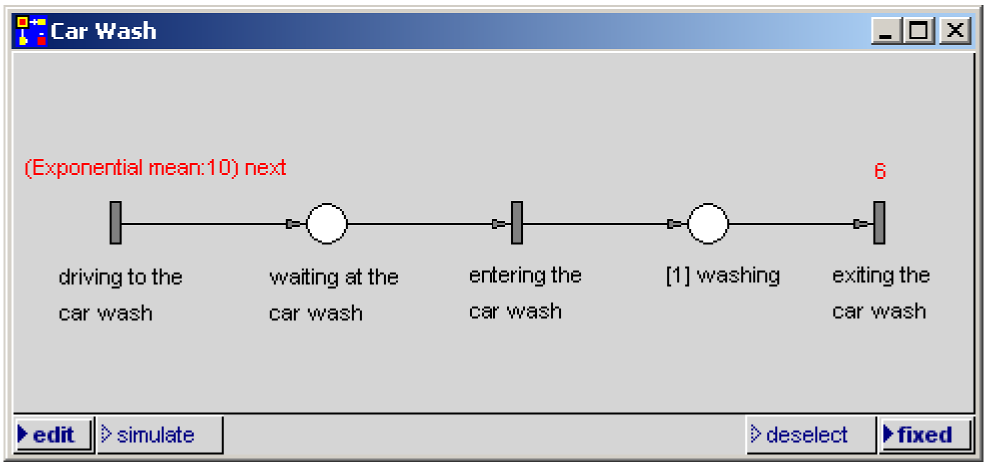


Figure 3.16: Net window of the car wash with inscriptions

To test the net, switch from the editor mode to the simulation mode. At the bottom left of the net window, press **simulate** with the **le.MK** key. Then call up the simulation mode's No-Selection-Menu (shown in Figure 3.17) with the **ri.MK** key.

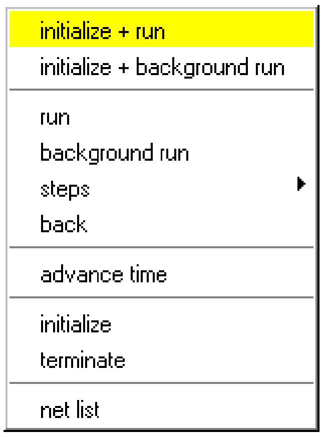


Figure 3.17: Net window menu in Simulation mode

To test for error-free net functioning, select the menu option **initialize + run**. You can see how tokens (cars) enter from the left, occasionally must wait in the 'waiting at the car wash' place, and exit the car wash after passing through the 'washing' place. To stop the simulation, move the mouse cursor into the net window (where the cursor is hidden) and press the **le.MK**.

An overview of the characteristics of the waiting queue can be created using a time histogram which shows how the time percentally spreads out on the different waiting queue lengths..

Open the menu of the place 'waiting at the car wash' in simulation mode and select menu option **diagram**. Select the menu option **time histograms** in the **diagram** submenu. This in turn has another submenu, where you will select **standard**. A frame for a graphic window will appear at the cursor position, which can be positioned and opened as described earlier (Figure 3.18).

Before you can use the diagram, a scale must be set. You can scale the ordinates by moving the mouse pointer to the left frame and pressing the **ri.MK** there. A menu with three options will appear; select menu option **maximum value**. Enter the value 0.6 in the window which opens, and then press **return**. Using the **inscriptions** menu option, you can set the number of scale values. Enter the number 12 here.

The abscissa is scaled analogously: Position the mouse pointer in the lower window frame and press the **ri.MK**. Here you fix the scaling, the maximum value and the number of bars. Input is done as just described in the scaling of the ordinates. We are selecting the inputs from Figure 2.9:

Minimum value = -1
Maximum value = 9
Number of bars = 10
Number of inscriptions = 10

Figure 3.19 shows the result. Note that the number of bars is shown by the scale number at the right edge of the bars.

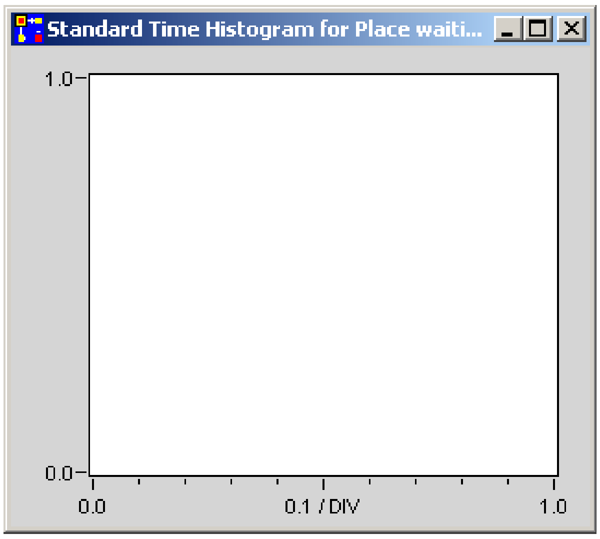


FIgure 3.18: Histogram window before scaling

If you want, you can now also introduce colors as in Figure 2.9. Bring the mouse pointer to the centre of the window and press the **ri.MK**. Select the menu point **colors**. A window will open in which you can select the foreground (color of the bars) and background color (color of the window).

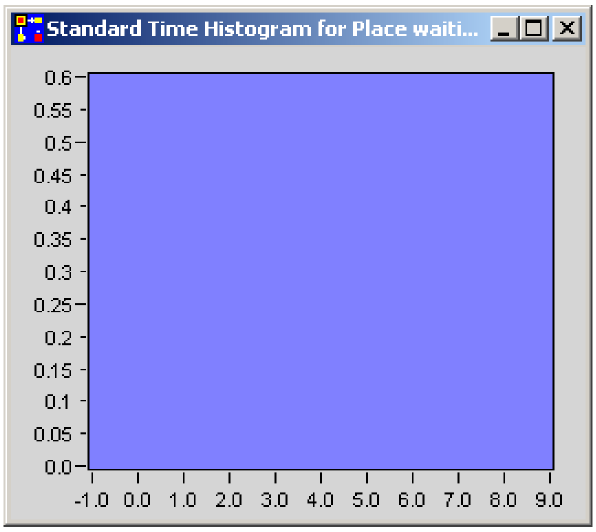


FIgure 3.19: Histogram window after scaling

The car wash histogram is generated by calling up the net window menu in simulation mode (Figure 3.17). This time, select menu option **initialize + background run** to quickly create the histogram (see Figure 2.9). The simulation may be halted by moving the mouse pointer into the net window and pressing the **le.MK** key.

Except for inserting illustrating icons for net elements, this completes the basic model of a car wash. We'll cover the procedure for inserting icons later.

The model can be saved using the **store image** menu option in the PACE navigator's File menu. This menu option opens a Windows window to save a file to the installation directory. It is useful to give the model a new name, e.g., CarWash. If you end the model using the **leave PACE** menu option, you can reload it later with the Windows Explorer by double-clicking on the name CarWash.imm. It appears on the monitor as it was when saved.

In the next section, we'll introduce several important characteristics of PACE useful for expanding the model.

3.2 Multiple Washing Programs

In describing action codes, we mentioned that the real objects are specified by tokens which carry their characteristic data. In addition to the objects data, other data important for processing can be attached to a token and transported through the net. In car manufacturing, for example, such information could be a list of extras to be installed in the car currently being built.

In the case at hand, let us assume that the car wash offers two different washing programs with respective wash times of 6 and 9 minutes. In addition, before the simulation run, we want to be able to set the percentage of vehicles which use the shorter washing program, using a bar gauge.

Access to token-associated data which runs through the net is provided with the help of connector variables. If we assume that the washing program is selected when entering the washing station, the information must be transported by the token to the transition 'exiting the car wash', so that the washing program and consequently the washing time are correctly set.

Our starting point is the net shown in Figure 3.16. If you have been running simulations and the net window is in simulation mode, press the **edit** button in the lower left corner with the **le.MK**. Set the mouse pointer on the connector between the net elements 'entering the car wash' and 'washing' and mark the connector by pressing **le.MK**. The marking is shown by a black dot on the connector.

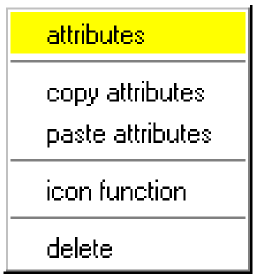


Figure 3.20: Menu for an exit connector

Use the **ri.MK** key to select the menu of the output connector of a transition as shown in Figure 3.20. If you select the **attributes** menu option, a two-part window will open (Figure 3.21).

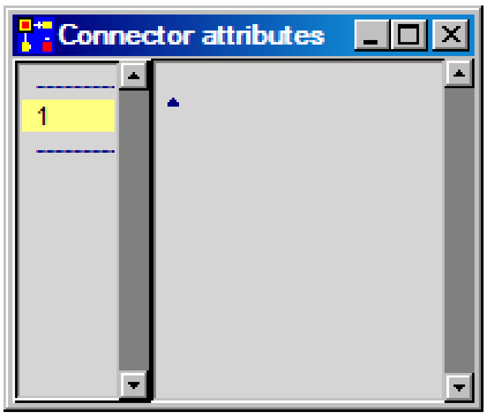


Figure 3.21: Connector attributes

The left part of the window gives the number of attributes or parameters which the tokens must show to be allowed to run across the connector. In Figure 3.21 we've specified one attribute that will serve for the current example.

The name of the connector variable is given in the right part of the window. Position the mouse pointer in the right part, press **le.MK** and enter the text **program**. Please note that connector variables must begin with a lower case letter.

Then, as demonstrated earlier, press **ri.MK** in the right part to call up the window menu and select menu option **accept**. The connector variable is now shown in the net window in parenthesis alongside the connector. As described earlier, it can be moved to a different location.

So the selected washing program can reach the transition 'exiting the car wash', the connector from the place 'washing' to the transition 'exiting the car wash' must also be attributed. If this is not done, the token which carries the washing program as an attribute cannot continue, because so far the connector will only accept tokens without attributes.

One could proceed here as above, by opening a window for the connector's attributes, etc. This would, however, be very tedious when many connectors have to be equipped with the same attributes. It is much easier to use the copy function for attributes.

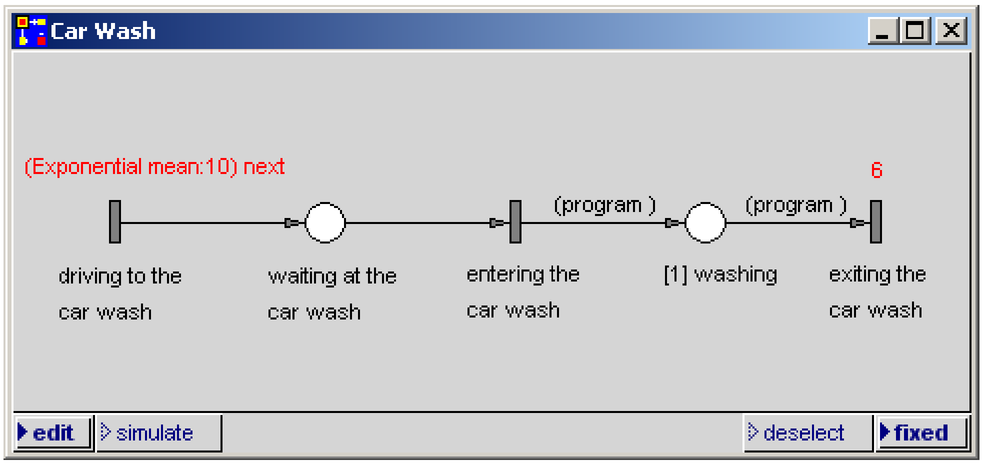


Figure 3.22: Net window with connector variables

To copy, mark the already-attributed connector and call up its menu with the **ri.MK**, then select menu option **copy attributes**. Then, mark the connector to be attributed, call up its menu with the **ri.MK** and select menu option **paste attributes**. The current status of the net window is shown in Figure 3.22.

To input the percentage of vehicles which select washing program 1 with 6 minutes wash time, a vertical bar gauge will be used. To create such a bar gauge, call up the View menu in the PACE navigator, then select submenu **linear gauges** and therein the submenu option **bar gauge**. A new frame for a graphic window will open, which may be adjusted as described earlier.

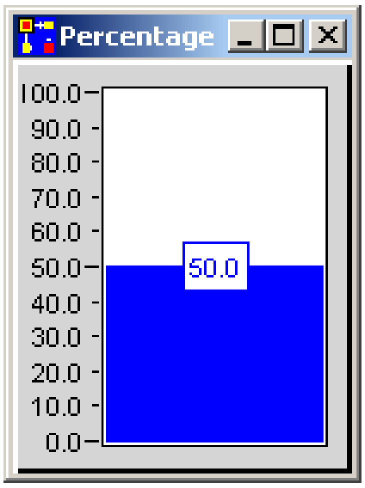


Figure 3.23:

Vertical Bar Gauge

As described earlier, a maximum scale value of 100 is set for the window. The scale can be further refined by setting the number of inscriptions to 10.

To set the name of the window, position the mouse pointer in the window and press **mi.MK**. This brings up the system menu for the window. After you select menu option **relabel as ...**, an input window will open in which you can enter the name **Percentage**. Then use the **le.MK** key to press the ok-button.

The bar gauge should now look as shown in Figure

3.23. To set a new value, move the bar with the **le.MK**.

To access the bar gauge in the model, it must first be connected to the model. This is best done at the beginning of a simulation run in the so-called **initialization code**.

Select the **Net Editor** menu in the PACE navigator, then select the submenu **extra codes** and, within the submenu, choose **initialization code**. This opens a text window for inputting the programming code which is to be executed at the beginning of the simulation run. Input the following text in the text window:

```
PercentageWashprog1 := BarGaugeValue named: 'Percentage'.
```

As described earlier, this text is accepted by pressing the right mouse key in the Initialization window (i.e., with the mouse pointer in the initialization code window) and selecting the menu option **accept**.

The input line will not be automatically accepted, because the identifier **Percentage-Washprog1** is as yet unknown. A query window will appear, in which you should press the **global** button with the **le.MK**. This declares the PercentageWashprog1 as a global variable which is recognized throughout the entire model and whose contents, the assigned bar gauge, can be accessed throughout the whole model. Global variables begin with a capital letter.

The washing program is selected when entering the car wash. So, as described earlier, call up the menu of the transition ’entering the car wash’ and select menu option **action code**. A new text window will open in which you can enter the action code to be executed when firing the transition:

```
barvalue := PercentageWashprog1 value.
program := (Bernoulli parameter: barvalue / 100) next
```

The first line assigns the set value of the bar gauge to the transition’s local variable **barvalue**. In the second line, the ”next“ value of a Bernoulli distribution with the prob-

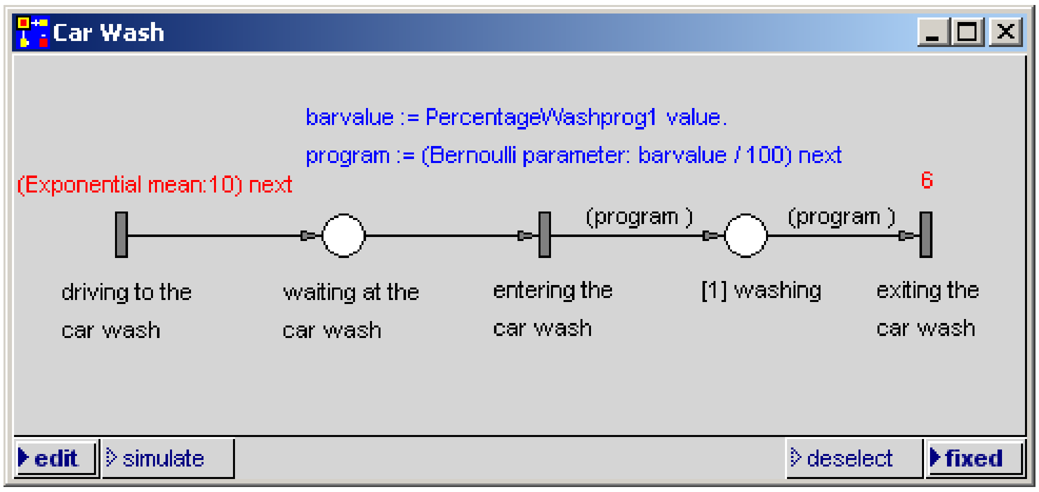


Figure 3.24: Net window with action code

ability 'barvalue / 100' is assigned to the connector variable **program**. It returns a value of 1 if washing program 1 is to be used, otherwise the value 0.

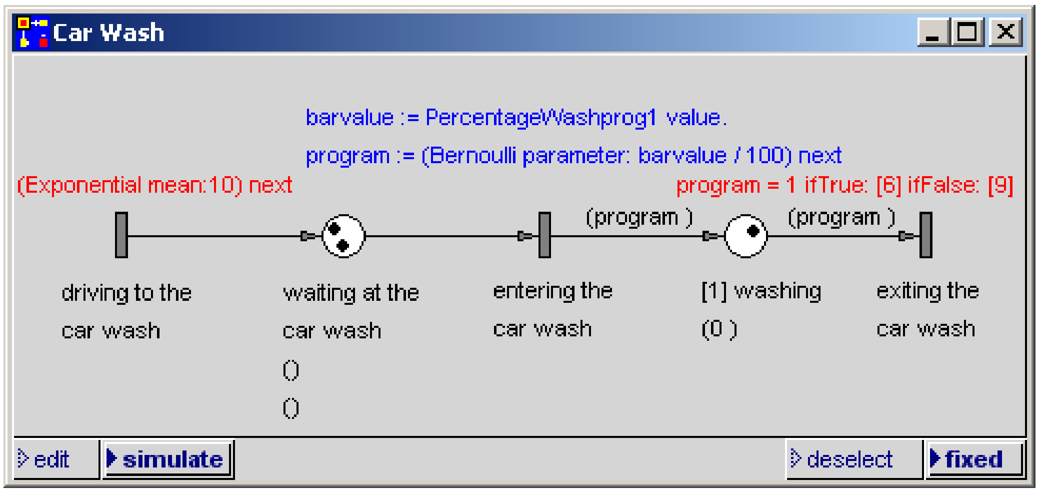
The program code is once again attached to the transition with the accept-function. Here again a window will open saying that the variable 'barvalue' is unknown. This time, press the **temp** button and the variable 'barvalue' will be set as a local variable of the transition. The action code for the transition can again, as described earlier, be moved to the appropriate place in the net window. The current status of the net window is shown in Figure 3.24.

When the net is running, the current value of the Bernoulli distribution is assigned to the token (the vehicle). You can access it via the connector variable 'program'. To set the correct delay time (wash time), change the delay code of the 'exiting the car wash' transition as follows:

program = 1 ifTrue: [6] ifFalse: [9]

This is a conditional message. It returns the result 6, if program = 1 (in other words, if washing program 1 was selected); else it returns a value of 9.

When changing a transition's code, it is not necessary to use the transition menu, but rather one can change the code directly. Mark the current delay-code in the transition 'exiting the car wash' with the **le.MK**. A small black dot is displayed in the middle of the code. Call up the code menu with the **ri.MK** key and select **inspect**. A window with the current delay-code will open. In this window, change the current code 6 to the conditional expression we have just discussed, and save it to the transition with **accept**.



```
barvalue := PercentageWashprog1 value.
program := (Bernoulli parameter: barvalue / 100) next
(Exponential mean:10) next
program = 1 ifTrue: [6] ifFalse: [9]
```

Figure 3.25: Net window with Extra Codes

The net window is shown in Figure 3.25. Reviewing the window you can imagine that with much longer inscriptions the net can become confused. To avoid that, you can hide the transition-code and replace it with three dots (...). To do this, call up the menu for the code again and, instead of **accept**, select the Three-Dot-menu on the last line of the menu. The inscription in the net window is replaced by three dots in the respective text color. The three dots are generated at the upper left corner of the previous inscription. To see which transition the code belongs to, call up the Three-Dot-menu. Mark the Three-Dots with the **le.MK** key, call up the appropriate menu with the **ri.MK** key, and select **owner**. The mouse pointer is placed on the transition to which the Three-Dots belong.

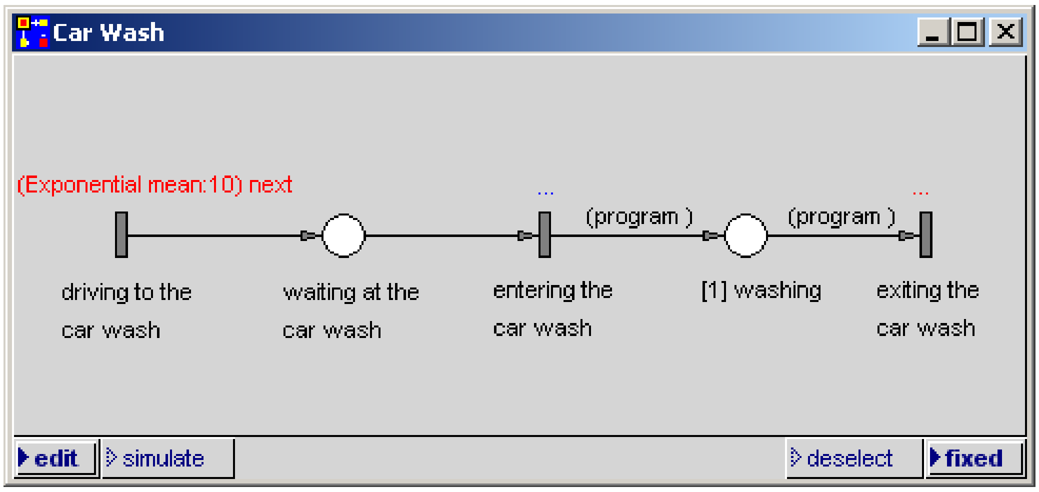


Figure 3.26: Net window with Three-Point-Codes

It's a better idea to move the three points in the net window into the vicinity of the transition to which they belong. This is done exactly as described earlier for other inscriptions. Figure 3.26 shows the net window with some inscriptions replaced by three dots.

If you want to return to the inscription text instead of the three dots, mark the dots and use the **ri.MK** key to select menu option **inscript**. A text window with the inscription will open. In this window, use the **ri.MK** to again select Tree-Points-Option ... . This menu option works like a toggle switch. You can, of course, also use the transition menu to point to the inscription hidden behind the three points.

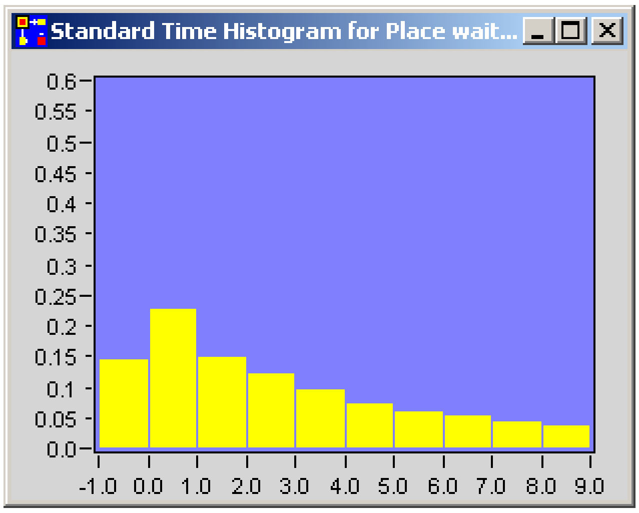


Figure 3.27: Time increments of different waiting queue lengths with a wash time of 9 minutes

With this, the model is finished and one can return to the simulation mode to experiment with the model. If you set the 'Percentage' bar gauge controller to 100%, you will again get the result shown in Figure 2.9. If you set the controller to 0%, the result is as shown in Figure 3.27. It shows a much better efficiency of the facility, caused by much longer waiting queues.

3.3 Runtimes

In this section, we will discuss working with simulation times. The current simulation time is stored in the global system variable **CurrentTime** and is set to zero when initializing a model.

In discussing net changes, we can be somewhat briefer here, as many of the editing steps have already been described multiple times. The changes are shown in Figure 3.28.

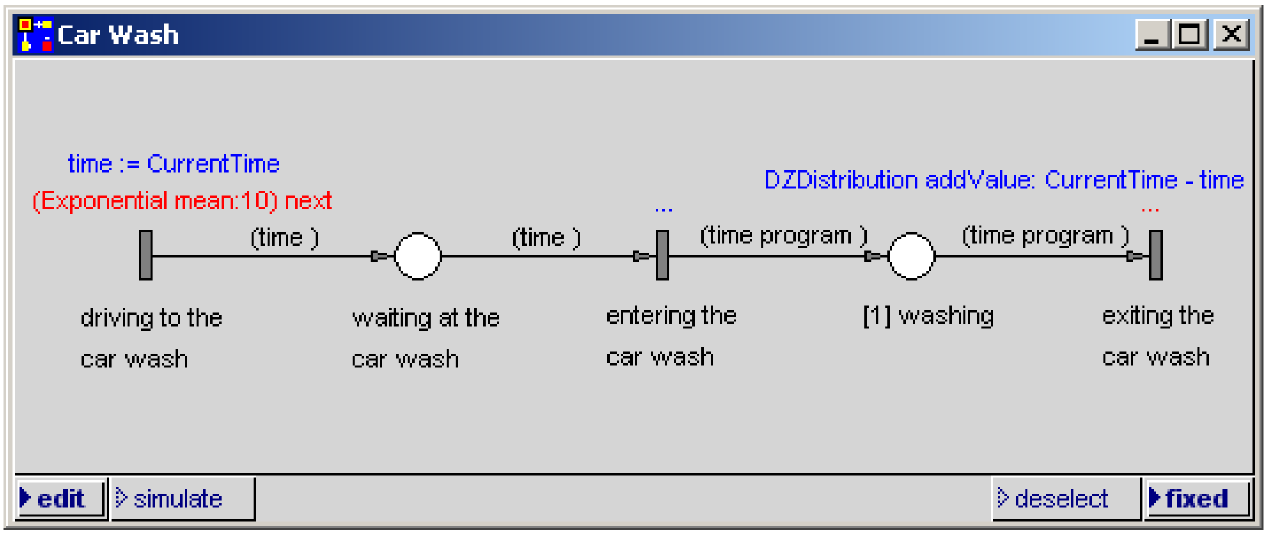


Figure 3.28: Changes in the net window

An additional connector variable 'time' is introduced for all connectors. With one of the two right-hand connectors, proceed as follows: in the connector attribution window (Figure 3.21), call up the selection menu in the left part of the window with the **le.MK** and select menu option **insert**. An additional connector variable is added in front of the already existing variable. If you want to set the variable after the other pre-existing variable, deselect the selected, yellow-highlighted variable by clicking on it with the **le.MK**, use the **ri.MK** key to call up the No-Selection-Menu and then select menu option **add**. In the right part of the window, enter the identifier **time** and choose **accept**. The second of the two connectors on the right can be attributed, as before, by copying.

In the transition 'driving to the car wash', the connector variable **time** is assigned the current time:

time := CurrentTime.

After going through the car wash, the runtime is:

CurrentTime – time.

It is interesting to look at the distribution of the runtimes. To do this, select the menu option **histograms** in the **Views** menu of the PACE navigator, and select **counts** in the submenu. In this example, the count-histogram shows the number of tokens (vehicles) across the runtime. The window is given the name ’Distribution of Runtimes’ (**mi.MK**) and is scaled as follows (Figure 3.29):

Ordinate:	Maximum value = 3200
	Number of inscriptions = 10
Abscissa:	Maximum value = 60
	Number of inscriptions = 10
	Bars = 60

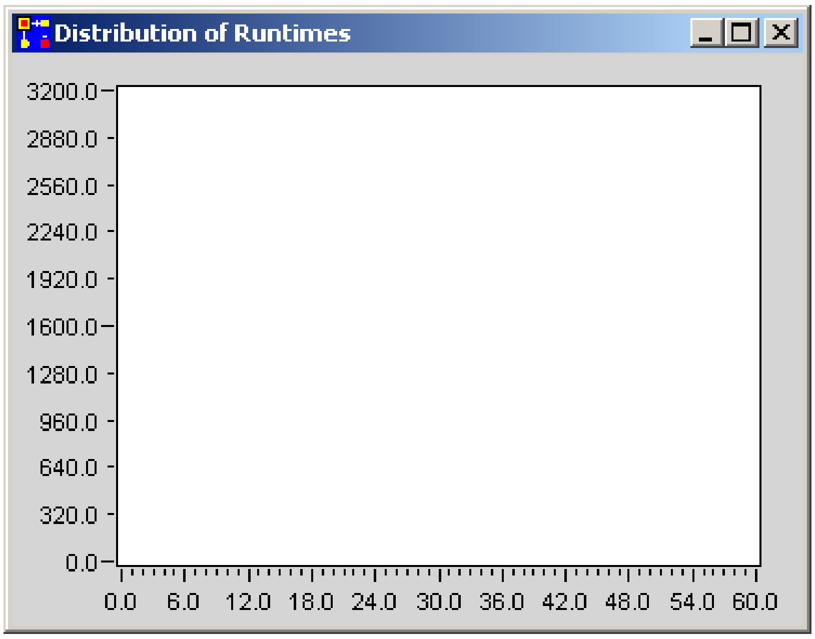


Figure 3.29: General Count Histogram

The count-histogram is attached to the model as before in the initialization code. This is expanded with the lines:

```
DZDistribution:= CountHistogram named: 'Distribution of Runtimes'.
DZDistribution clear.
```

The second line clears the contents of the histograms during initialization.

The action code in the transition ’exiting the car wash’ is expanded with the output code for the standard histogram. The expansion is as follows:

```
DZDistribuiton addValue: CurrentTime – time.
```

If you execute the model with a percentage of 50%, you will get the runtime distribution shown in Figure 3.30.

In this result, you will get very large unrealistic runtimes. They stem from the large waiting queues which appear in the model, but which do not really occur in actual practice, as drivers no longer enter a waiting queue of more than approximately 3 vehicles. This condition will be considered in the expanded model (Chapter 4).

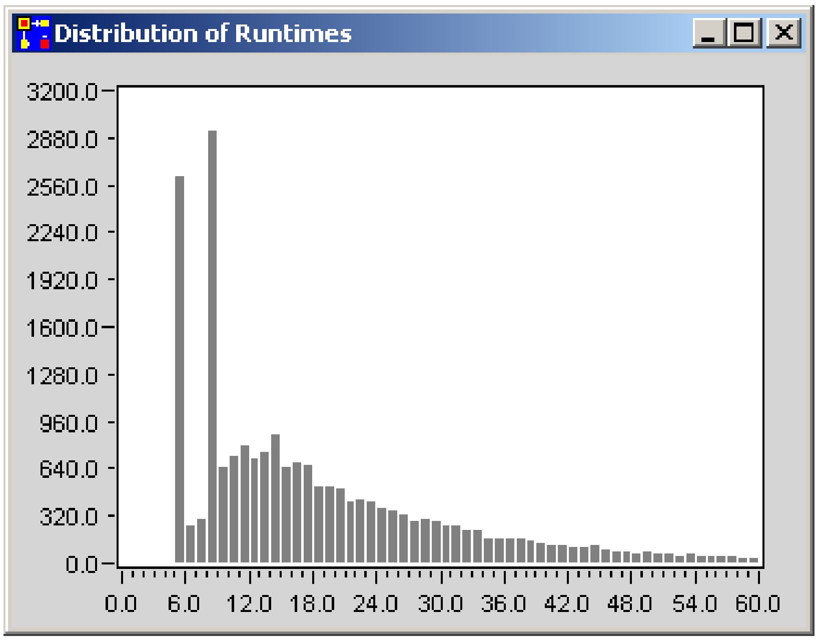


Figure 3.30: Distribution of Runtimes

3.4 Iconizing

In general, one cannot simply see which processes are represented in a net. The user who is to use the model later and who is usually only peripherally interested in the model's implementation, can work better with the model when the default net elements are replaced by application-specific pictures (Figure 2.10). We'll now describe how that is done.

The graphics needed here are stored in the icon file **CarWash.icn** in the **nets** directory of the PACE installation directory. To load them, mark the line CarWash in the net list. Then select the **Extras** menu on the PACE navigator and select the **icons** submenu. Finally, select menu option **icon file**.

A window for loading an icon file will open. It shows the entries in the **nets** directory. Select the file **CarWash.icn**, which is loaded into the model either by doubleclicking on the entry in the Explorer list or by pressing the **Open** button.

To look at the icons you've loaded, select menu option **icons** in the **Extras** menu, and option **individual icons** in the submenu. A window will open with a list of the names of all stored icons (graphics). Click on a name with the **le.MK**, and its graphic will appear in the upper right corner of the window as long as the mouse key is pressed down (Figure 3.31).

The selection menu for this window is also interesting. It may be called up as usual with **ri.MK**. We recommend that you experiment with the different menu options, in particular replacing an icon (**from screen**, **from clipboard**, etc.), scaling an icon (**scale**), and fading an icon (**fade**). The graphics WholeCarWash1 and WholeCar-Wash2 are one-step and two-step fades of the WholeCarWash graphic. You can change a graphic with the Icon Editor. The No-Selection-Menu of the window contains only the menu option **add**, with which a new name may be added to the list. This may then be assigned a graphic with one of the from-menu options.

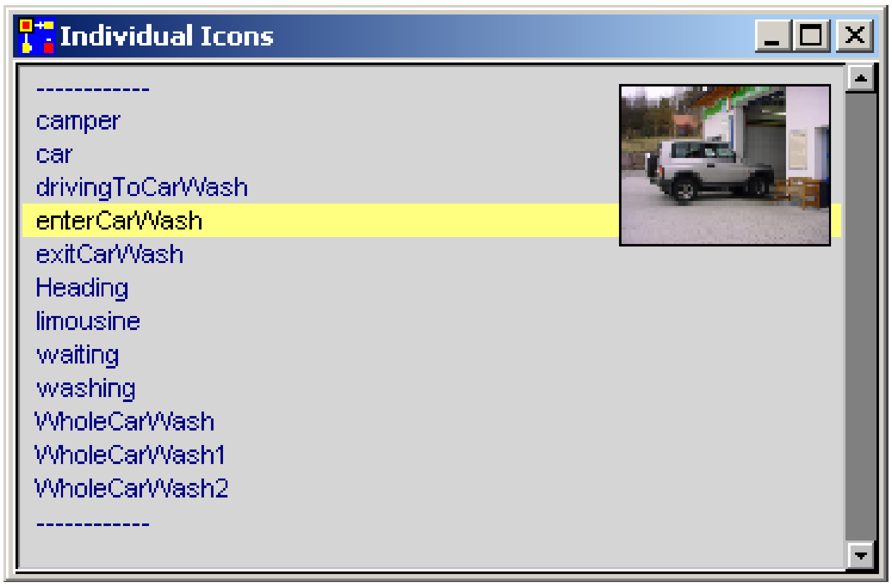


Figure 3.31: Icon list

In the net shown in Figure 3.28, the transition 'driving to the car wash' is marked, and the menu option **individual** is selected in the **icon** submenu. In the window which opens, select the 'drivingToCarWash' line and press the **ok**-button. The inscriptions may be partially covered by the graphic and should be moved accordingly. Similarly, assign graphics to the other net elements for 'waiting', 'enteringCarWash', 'washing', and 'exitCarWash'. All code inscriptions can be replaced with Three-Dot inscriptions.

To represent the title of the window, you can insert a net element, e.g., a place or a transition, to the spot in the window where you'd like the title to appear later. That net

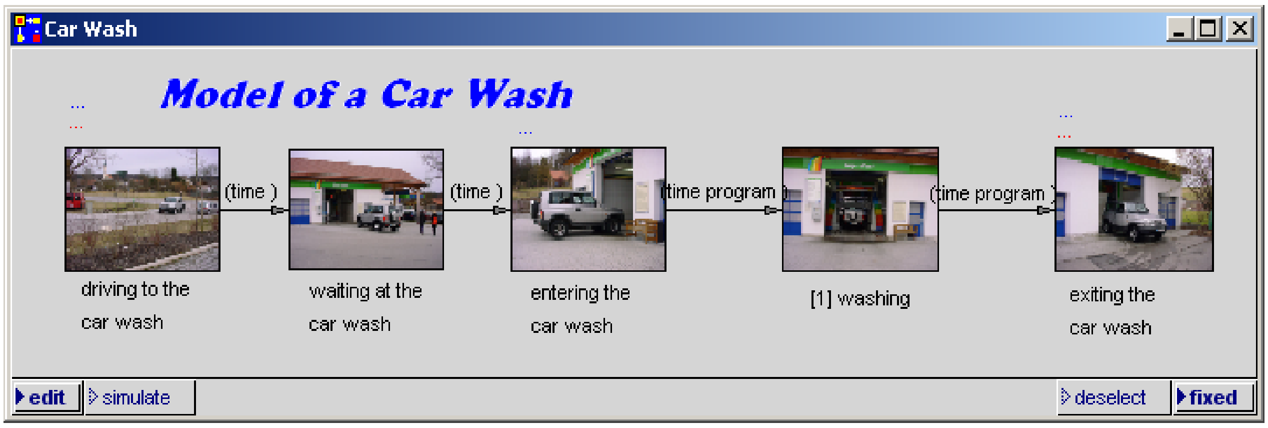


Figure 3.32: Iconizing elements in the net window

element is replaced by the graphic 'Heading'.

The current status of the net window is shown in Figure 3.32.

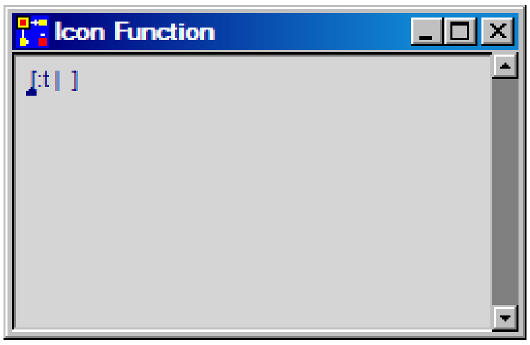
If you want to replace the standard symbol for a token (a small solid black circle) by graphics of the object which moves through the net, the procedure is as follows: mark the output-connector (of a transition) over which the icons should cross and call up its menu with the **ri.MK**. Then select menu option **icon function** and select **edit** in the submenu which appears. The window shown in Figure 3.33 will appear.

This window includes a Smalltalk-Block which is evaluated each time the token crosses the connector. The block delivers as its output the name of the icon to be used, in the form of a so-called symbol. A symbol appears as a number sign # in front of the name (Example: #limousine).

If the result of the block is only one name, it will always run the assigned icon over the connector. The block then looks like this:

[:t| #limousine].

As before, the code becomes functional by clicking **accept** in the **ri.MK** menu of the window. Then close the icon window.



```
[:t| ]
```

Figure 3.33: Window to specify a token icon

Again, use the copy function to attribute token icons to the second output connector, which runs from the transition 'entering the car wash' to the place 'washing'. Mark the connector you have just attributed, then select menu option **icon function** and execute the **copy** command in the submenu. Now mark the second exit connection, call up **icon function** again and select **paste**.

If you like, you can also set a background graphic. In the No-Selection-Menu of the window in Figure 3.32, select the **insert background image** menu option, then select the **WholeCarWash2** line in the window which opens. To avoid disruptive flickering during editing, background graphics should be added at the completion of a module or model.

With this, the net is set and you can exit PACE. In the File menu of the PACE navigator, select **leave PACE**. A query window will appear which asks if you wish to save the model. To say, answer **yes**. The rest of the operation has been described earlier (at the end of Section 3.1). PACE will close automatically after saving the model.

4. Components (Modules)

In our language, we need concepts for complex relationships so we can express ourselves clearly and efficiently. Similarly, in nets one has an option of using closed subnets to represent specific partial tasks. These are called components or modules. Only with such groupings can we represent reality-based hierarchies and get a better overview of the nets: the nets become understandable.

Many simulator development systems only use pre-defined components and create simulations out of these. Since the real world is not one-size-fits-all, this procedure often requires quite a number of hard-to-understand adaption parameters with which the predefined modules are fitted to the current situation, and which can turn the debugging process into a time-consuming activity. In spite of the extra effort, the resulting models only roughly reflect the actual situation and often deliver results with limited validity. Fitting the model to reality by changing the modules is difficult and can usually be done only by specialists.

PACE takes a different approach, providing means for exact replication of reality in a model that can be animated and verified step-by-step. To represent the hierarchical structure of process systems, reusable components (also called modules or subnets) can be created using the net elements described previously. Thus you can use PACE's features to create adaptable components and component libraries which can be altered and expanded by the user if they don't fit the specific situation or don't accurately represent reality.

4.1 Creating a Component

The goal of this section is to create the adaptable module **CarWash**, which can be used in other models. It is to be adaptable to the car wash at hand via the following four parameters:

Washtime1
Washtime2
Percentage
Waitingqueuelength

As its result, it should deliver (in the form of a token attribute) the runtime for the just-washed vehicle. By runtime we mean the time span between arrival of the vehicle at the car wash and its departure (see Section 3.3). The four parameters in question were discussed in previous sections.

The current plan differs from the models in Chapter 3 in that the adaptation will involve no net changes. All fitting is done by pre-establishing arguments (parameters). This way, later users of the module need no detailed knowledge of how the module was built, but must only know which task the module solves, what the adaptation parameters mean, and how they can be adjusted.

Basically, there are two options for creating a module:

- Start with an empty module and build a new net.
- If a usable starting net already exists, modify it as desired and generate the new module from its net elements using menu option **coarsen**.

While the second option above is useful for users with PACE-experience, we will select here the first option, as a description of changes to be executed would distract from what is important in this discussion.

Start with a new model, and give it the name **ModuleDevelopment**. After opening the net window, use the No-Selection-Menu to create the three net elements shown in Figure 4.1 and connect them via connectors. The net element represented by the square is a module and is created using menu option **module**.

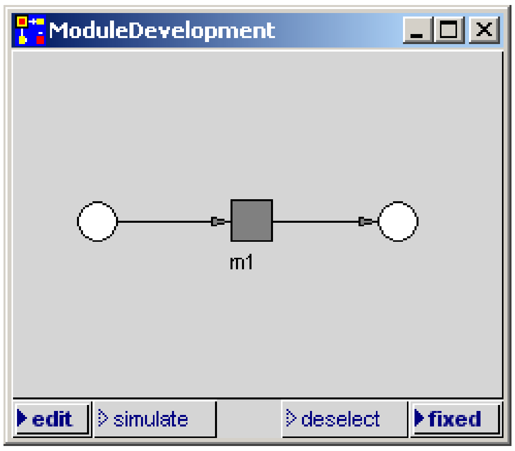


Figure 4.1: Exit net

The net elements are identified as shown in Figure 4.2.

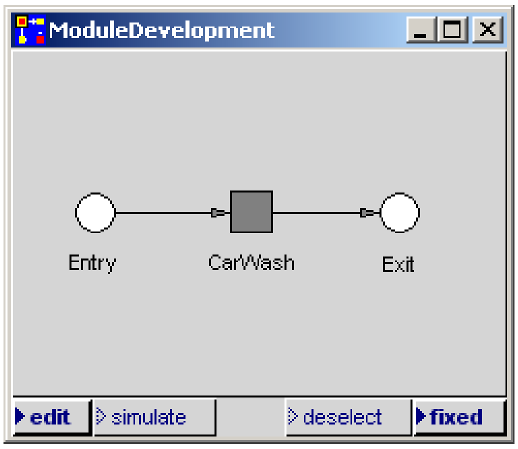


Figure 4.2: Exit net with legend

In the case of a module, the default module identifier m1 is to be replaced with the identifier 'CarWash' by marking m1 with the **le.MK**, then opening the input window for the module name in the **ri.MK** window with menu option **inspect**. Enter the name CarWash and use menu option **accept** in the **ri.MK** menu or press the **Return** key.

To make the module 'CarWash' independent of its environment, no external relationships may appear. Thus, it is not possible to represent parameters like Washtime1, Washtime2, etc., as global variables as we did previously. It would also not be helpful to declare the global variables after the insertion in the model in which the module is to be included! If the module is inserted more than once, all inserted modules would access the same global variables. It would not be possible to have CarWashes with different wash times without changing the nets of the modules.

To model closed modules, we have to introduce net or module variables which are known to the whole module and, if they exist, its submodules. To declare the parameters given at the beginning of this chapter as net variables, mark the 'CarWash' module and select menu option **net variables** in the **ri.MK** menu. The window shown in Figure 4.3 opens to declare the net variables. Insert the name of the net variables in the form of symbols in the left part of the window. In the right part of the window, you can use the buttons **initial value** or **current value** to set values. The value designated as the initial value is assigned to the net variables at initialization of a model.

Select menu option **add** in the **ri.MK** menu in the left part. An input window will open in which you can enter the name of a net variable in the form of a Symbol (that is, with a # sign in front of it). Then enter **#Washtime1**, and finish by pressing **Return**. Then position the mouse pointer in the right part of the window, enter the wash time **6**, and select **accept** in the menu of the right part. Then deselect the identfier **#Washtime1** in the left part (click on the identifier with the **le.MK** and the yellow highlighting will disappear). Then enter the other net variables shown in Figure 4.4 and assign them their respective initial values of 9, 50 and 0.

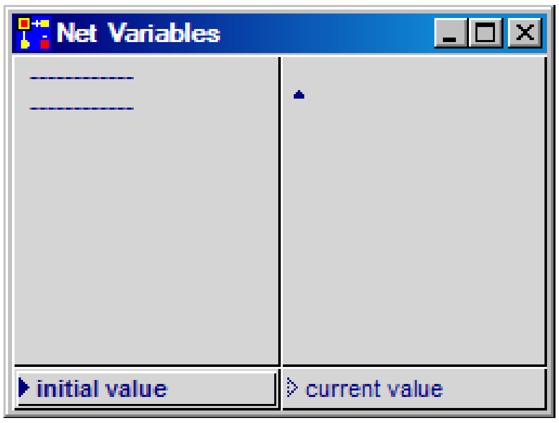


Figure 4.3: Window for agreement of net variables

Net variables are accessed via messages. The current value of the net variable **Waitingqueuelength** is produced by the following message:

(self at: #Waitingqueuelength) value

The value 40 can be assigned to the net variable #Percentage by entering:

(self at: #Percentage) value: 40

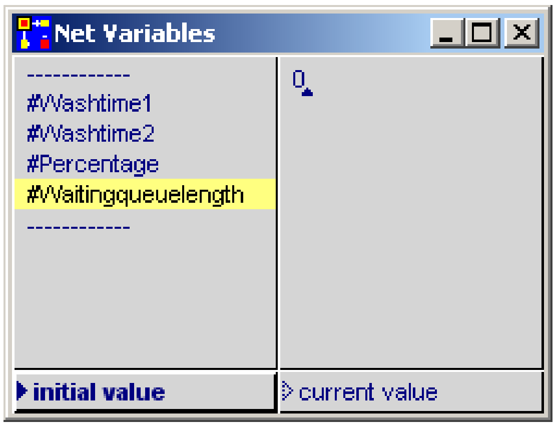


Figure 4.4: Window with agreeing net variables

The net window of the CarWash module is selected by marking the module and choosing the menu option **subnet** in its **ri.MK** menu (Figure 4.5). The module window which opens shows both places, 'entry' and 'exit,' which constitute the interfaces to the module’s environment. To indicate that these are declared outside the module (that is, within the module they only serve as placeholders), they are shown paler (faded) in the net of the module.

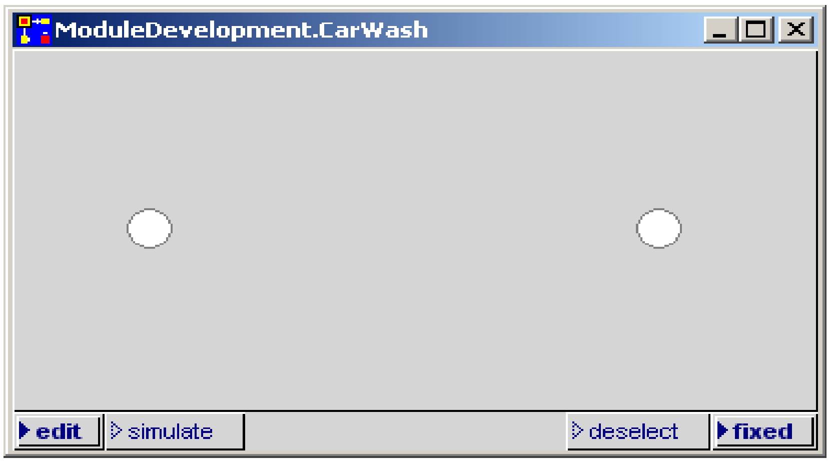


Figure 4.5: Net window after opening

In the module, you'll next draw the net shown in Figure 4.6 without code inscriptions. Those will be discussed later.

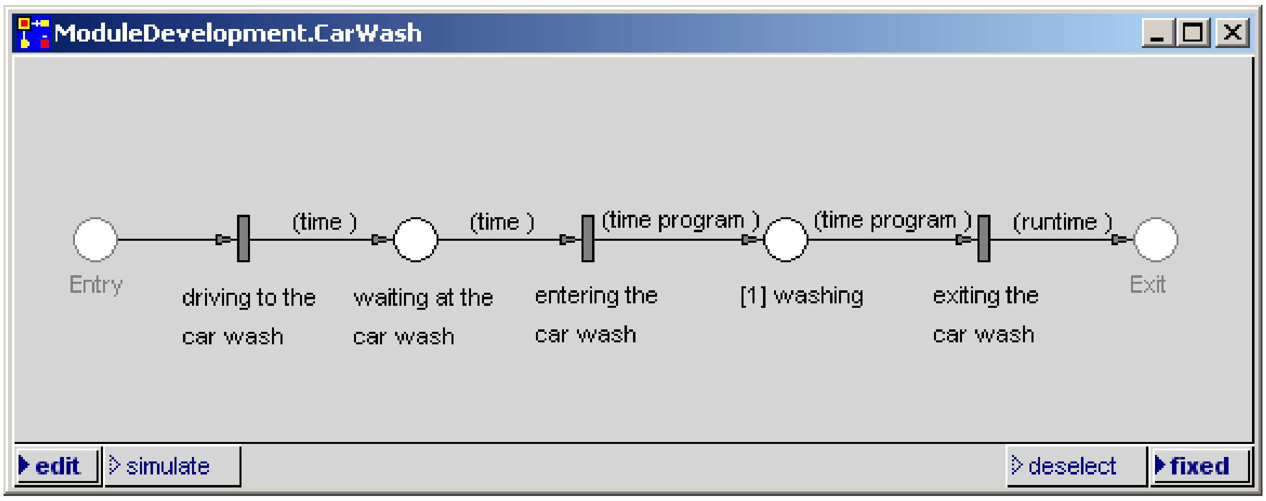


Figure 4.6: Net of the 'CarWash' module without code inscriptions

Since arrival at the car wash proceeds from outside, the Transition 'driving to the car wash' requires no delay code, unlike in Chapter 3. The action code reads:

```
time := CurrentTime.
temp := (self at: #Waitingqueuelength).
temp value: (temp value + 1).
```

The first line stores the current simulation time in the connector variable 'time'. During simulations, it is transported through the net with a token and, in the transition 'exiting the car wash', compared with the current time at that point. The difference is the length of time, the runtime, that the vehicle was in the car wash.

In the next line, a temporary variable 'temp' is introduced, to which the object net variable 'Waitingqueuelength' (not its value!) is assigned. In the third line, the message described above to set the value of a net variable is used, to increment the net variable 'Waitingqueuelength'.

Since the transition-local variable 'temp' is unknown, the menu option **accept** brings up a selection menu for defining 'temp' by pressing the **temp** button (Figure 4.7).

To see the current local variables in a transition, select the menu option **temporaries** in the transition's **ri.MK** menu. Here you can add more local variables with the No-Selection-Menu's **add** function, as well as renaming or deleting selected temporary variables with the **ri.MK** menu (careful!).

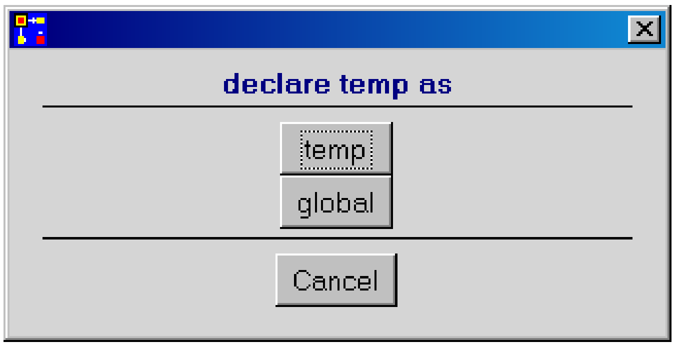


Figure 4.7: Query window to determine the variable temp

The action code of the transition 'entering the car wash' must reduce the Waitingqueuelength by 1 and then select the washing program. It reads:

```
temp := (self at: #Waitingqueuelength).
temp value: (temp value - 1).
barvalue := (self at: #Percentage) value.
program := (Bernoulli parameter: barvalue / 100) next.
```

Interpreting this should be no problem given earlier descriptions.

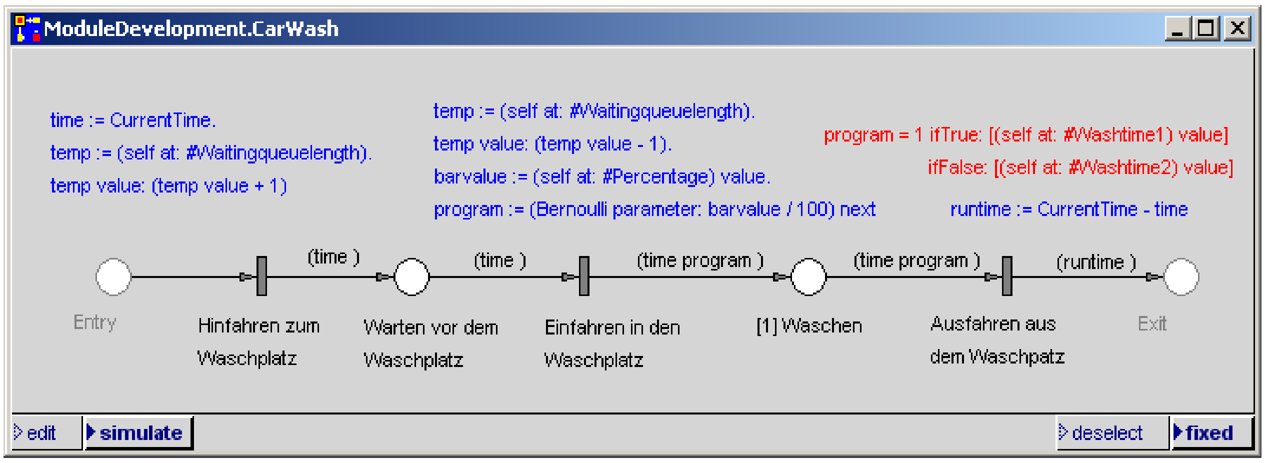
Now we only need the delay code and the action code for the transition 'exiting the car wash'.

The delay code can be drawn from the earlier code (see Figure 3.25). It reads:

```
program = 1 ifTrue: [(self at: #Washtime1) value]
            ifFalse: [(self at: #Washtime2) value].
```

The action code stores the runtime in the connector variable runtime:

```
runtime := CurrentTime – time.
```



```
time := CurrentTime.
temp := (self at: #Waitingqueuelength).
temp value: (temp value + 1)
```

```
temp := (self at: #Waitingqueuelength).
temp value: (temp value - 1).
barvalue := (self at: #Percentage) value.
program := (Bernoulli parameter: barvalue / 100) next
```

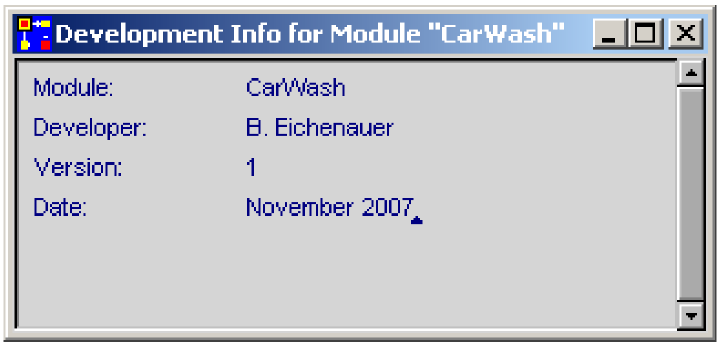
```
program = 1 ifTrue: [(self at: #Washtime1) value]
            ifFalse: [(self at: #Washtime2) value]
```

```
runtime := CurrentTime - time
```

Figure 4.8: Net of the ’CarWash’ module with code inscriptions

Now all that remains to be done is **documentation:** the type of work most unpopular with developers (but so important for quality assurance and for the information of later users of the module!). PACE offers several options:

1. Information about the development of the module
   Mark the module CarWash in the net list, then use the **ri.MK** to call up the window’s selection menu and select menu option **development info**. It opens a window for entering information about the development. Save the information with **accept**. In the case of module problems or later expansions, this information can make it possible to find the changes which have been made and identify the developers of the module.

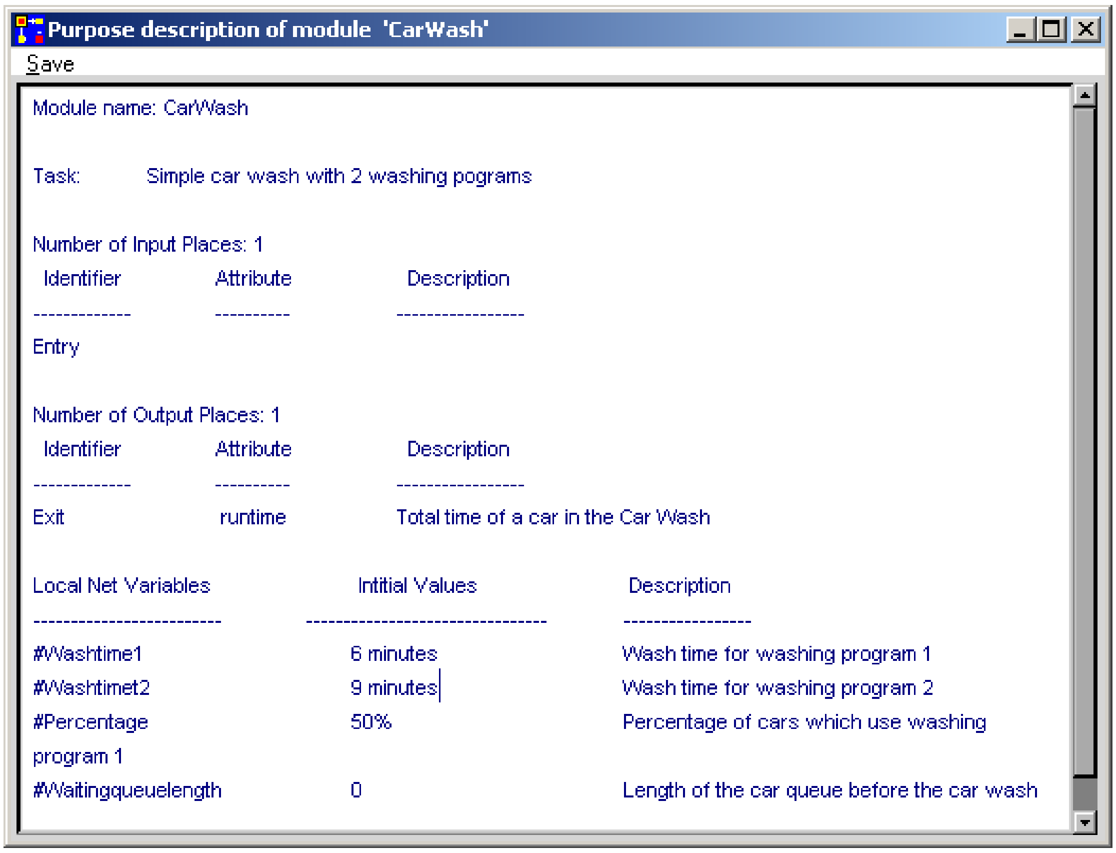


Module:	CarWash
Developer:	B. Eichenauer
Version:	1
Date:	November 2007

Figure 4.9: Development Info for the 'CarWash' module

2. Describing the module
   Individual net elements are documented with entries which are attached to the net elements. In the case at hand, mark the net module in Figure 4.2, open the module’s menu with the **ri.MK** key, and select **purpose description**. A text window will open into which you can enter the module information.

As an example, Figure 4.10 shows the most important entries in the module 'CarWash' for use by later users. In the module's net, additional implementation-specific entries can be attached to the net elements in use.



Module name: CarWash

Task: Simple car wash with 2 washing pograms

Number of Input Places: 1

Identifier	Attribute	Description
Entry		

Number of Output Places: 1

Identifier	Attribute	Description
Exit	runtime	Total time of a car in the Car Wash

Local Net Variables	Intitial Values	Description
#Washtime1	6 minutes	Wash time for washing program 1
#Washtimet2	9 minutes	Wash time for washing program 2
#Percentage program 1	50%	Percentage of cars which use washing program 1
#Waitingqueuelength	0	Length of the car queue before the car wash

Figure 4.10: Purpose Description for the 'CarWash' module

For models, PACE also offers an option for creating a user manual which is bundled with the model and can be called up at will. You can access it with the menu option **model user manual** in the **Extras** menu on the PACE navigator.

This completes the module 'CarWash' (Figure 4.8), which must now be stored in a directory. Generally, the **modules** directory in the PACE directory (the directory from which PACE is started, i.e., in which the started Image is stored) is intended for this. Users can also use their own directories, for example, to prepare a package of modules for a specific application area.

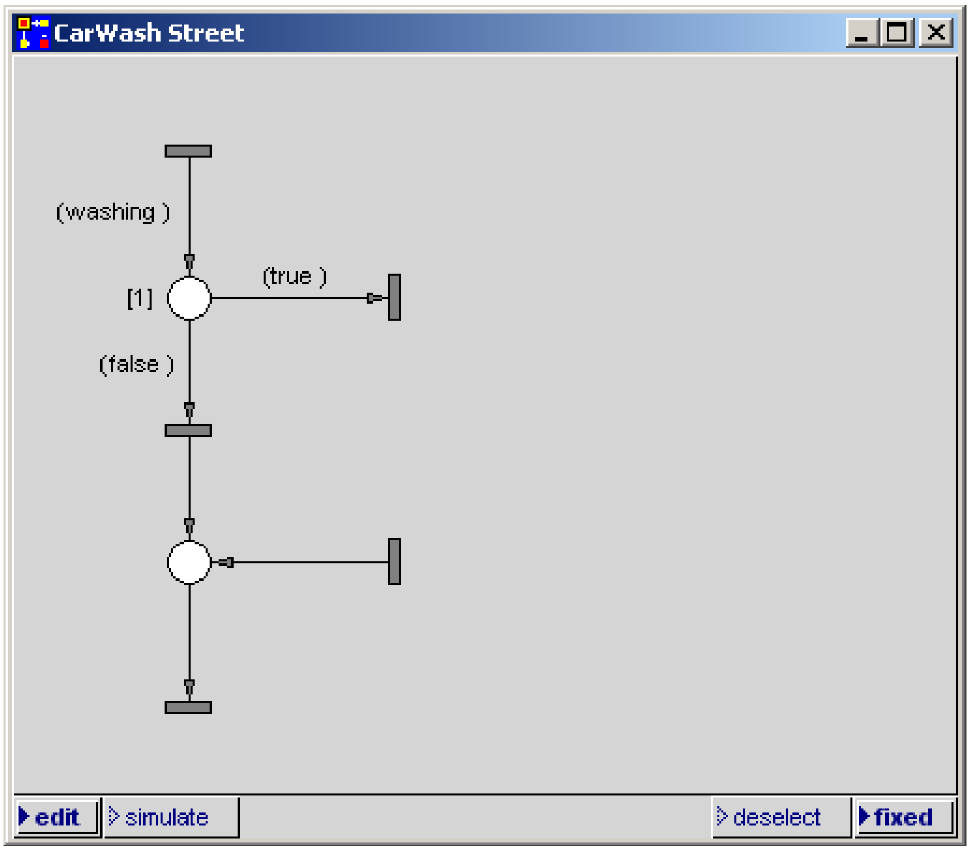
All of the model's modules are listed in the 'Net List' window. If the model 'CarWash' is not already there, click in the window with the **le.MK**. This refreshes the window and shows the latest status. Mark the CarWash modules with the **le.MK**, then call up the menu option **store module** in the PACE navigator's **File** menu. A standard Windows window will appear for saving a module to a pre-set directory **modules**. If you press the Save button, the module is saved with the name **CarWash.sub** in the **modules** directory of the PACE directory.

After this, you can exit PACE as previously described.

4.2 Using Components

To show how a module can be inserted into a model, we're now going to create a model of a street attached to the car wash we have modeled. To keep things simple, we're going to assume a one-way street, that is, we'll only need to specify the flow of traffic in one direction.

We'll start again with a new model, which we'll give the name 'CarWash Street'. The street and its driveway into the CarWash is shown in Figure 4.11 and can easily be replicated.



CFigure 4.11: One-way street with connection points for a car wash

The module we created in the last section is inserted as follows: execute menu option **restore module** in the net window's No-Selection-Menu. It opens a window for the selection of the module. The default is the **modules** directory in the PACE directory. Select the file **CarWash.sub** and press the **Open** button. At the mouse pointer position, a frame will appear which can be moved to the desired position of the module in the net window. After the move press the **le.MK** button. Figure 4.12 shows the result.

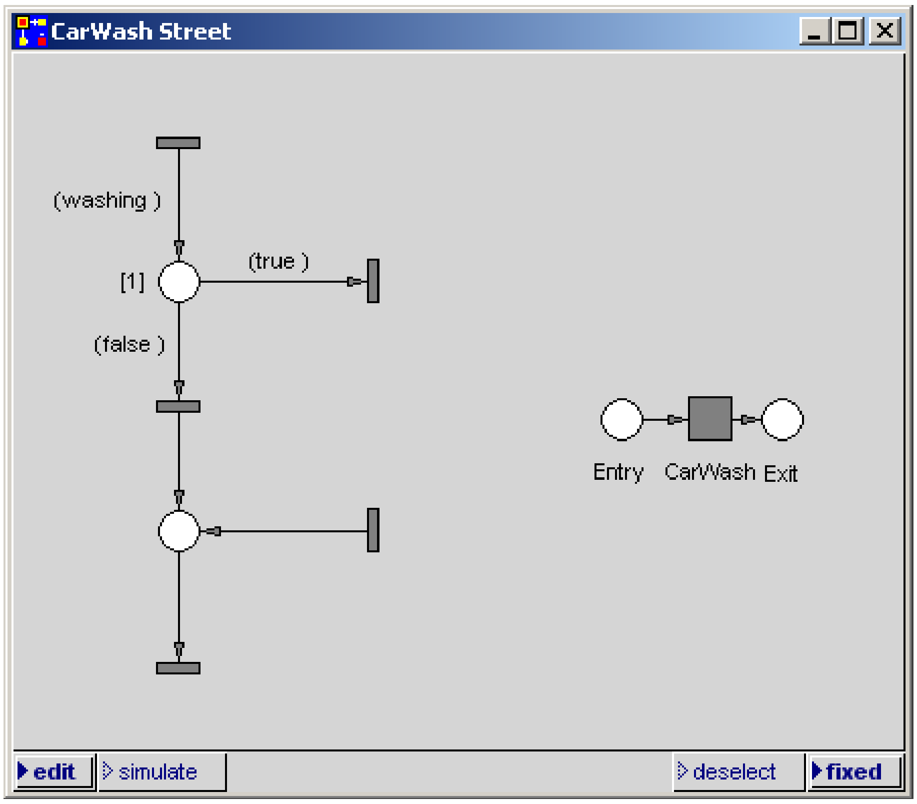


Figure 4.12: Net window after entering the module 'car wash'

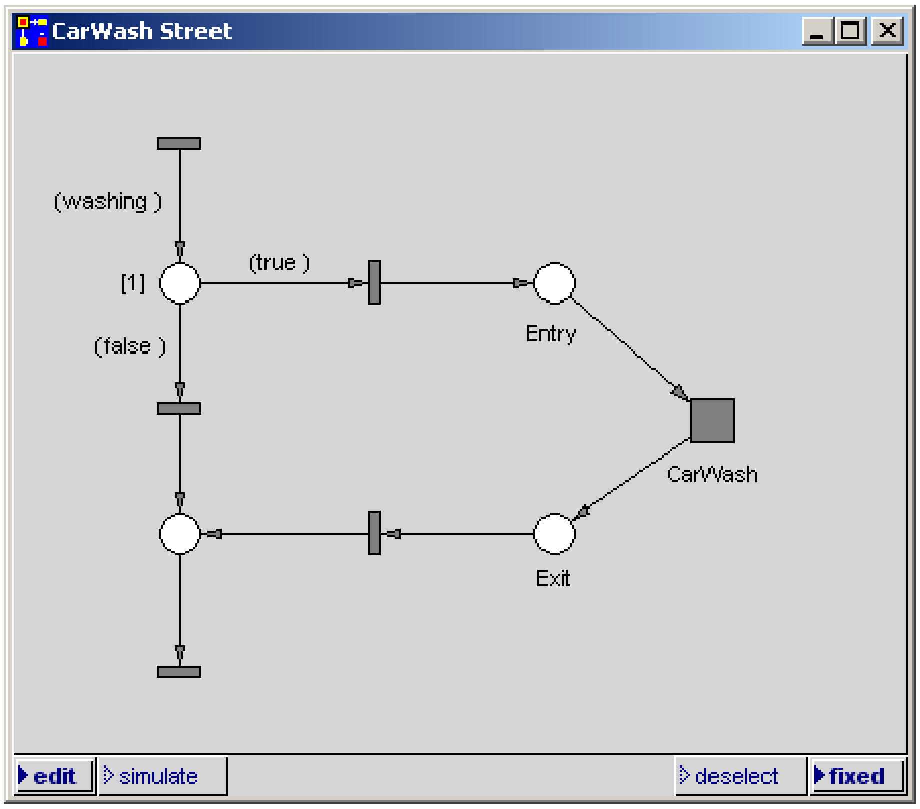


Figure 4.13: Street with tacked-on car wash

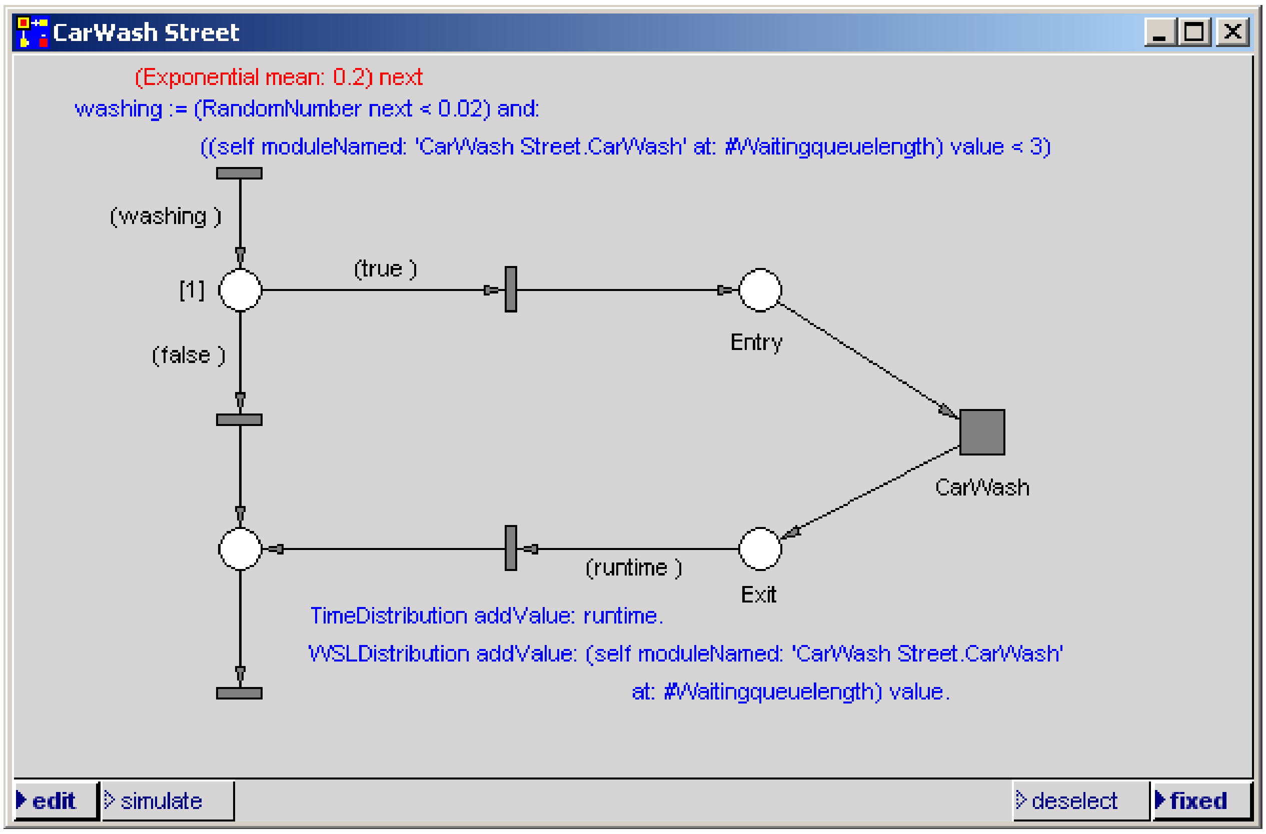
Now shift the individual net elements to the desired positions and connect the interfaces (the places 'entry' and 'exit') with the net for the street. The result is the net shown in Figure 4.13.

To finish the adaption, set the net variables for the 'CarWash' module as follows:

Washtime1	= 7
Washtime2	= 10
Percentage	= 60.

and draw the connector from the place 'exit' to the adjoining transition. The last is needed so that tokens with the attribute **runtime** can flow across this connector without hanging up at the place 'exit'. The required steps for this were already described earlier .

Figure 4.14 shows the net with all inscriptions.



```
(Exponential mean: 0.2) next
washing := (RandomNumber next < 0.02) and:
    ((self moduleNamed: 'CarWash Street.CarWash' at: #Waitingqueuelength) value < 3)
```

```
TimeDistribution addValue: runtime.
WSLDistribution addValue: (self moduleNamed: 'CarWash Street.CarWash'
    at: #Waitingqueuelength) value.
```

Figure 4.14: Street with car wash inscribed

The delay code of the uppermost transition says that in the mean, a vehicles arrives every 12 seconds at the modelled piece of street. The action code specifies which cars will be washed. A connector variable 'washing' is introduced here, which is assigned a 'true' or 'false' value in the action code. These values are stored as arguments in the outgoing tokens.

Depending on the value of this argument, a token will exit from the attached place either to the right (true) into the car wash, or downward (false) past the car wash. Here we apply the rule that a token can only pass a connector when the number of arguments for the token and the number of connector inscriptions are the same. If connector constants are specified (in this case, the values 'true' and 'false'), the arguments assigned to the token and the connector inscription must match. Thus, if the token has the value 'true' (or, as the case may be, 'false,') it can only pass to the connector leading to the right, (or, as the case may be, below).

The action code consists of an assignment whose right side consists of two conditional expressions joined by **and:** (logical and). The first expression:

RandomNumber next < 0.02

uses the global system variable RandomNumber which can generate uniform distributed random numbers between 0 and 1. The **next** method delivers the next random number. If this number is smaller than 0.02, so in the mean in 2% of all cases, the expression is returned as true. Else, it is returned as false.

The second expression accesses the net variable #Waitingqueuelength in the ’CarWash’ module. The net variable (not its value!) comes from the expression:

self moduleNamed: ' CarWash Street.CarWash' at: #Waitingqueuelength.

The module is thus identified by the string ' CarWash Street.CarWash', in which the modules, beginning with the root ‘CarWash Street' are concatenated sequentially, separated by dots, as they are hierarchically ordered in the 'Net List' window. Using the **value** method, the value of the net variable is read. If it is smaller than 3, the expression is returned as true, otherwise false. Drivers only go to the car wash when fewer than 3 vehicles are in the waiting queue.

The and-connection for the expressions says: If the vehicles falls into the 2% which will be washed, and the length of the waiting queue in front of the car wash is smaller than 3, then the car will drive into the car wash.

The inscription of the transition attached to the 'exit' place,

TimeDistribution addValue: runtime.

assumes that a count histrogram was created and assigned in the initialization code:

```
TimeDistribution :=  CountHistogram named: 'Distribution of Runtimes'.
TimeDistribution clear
```

Because of the changed parameters, the scaling of the histogram has also been changed from its earlier version (Figure 4.15).

If you switch to simulation mode and execute the net as **background run**, the runtime distribution shown in Figure 4.15 will apply.

The distribution of time in the different waiting queue lengths can be shown as before. In the **View** menu of the PACE navigator, a standard histogram is generated with the name 'Waitingqueuelength'.

The initialization code is expanded by the following two lines:

```
WSLDistribution := StandardHistogram named: ' Distribution of Waiting Queues'.
WSLDistribution clear.
```

The action code of the transition connected to the place 'exit' is expanded by the following inscription:

```
WSLDistribution addValue: (self moduleNamed: ' CarWash Street.CarWash'
                                              at: #Waitingqueuelength) value.
```

If you execute a simulation run, you will get the histogram shown in Figure 4.16. It shows that the waiting queue is empty about 60% of the time, that about a quarter of the time a car is waiting to be washed, etc.

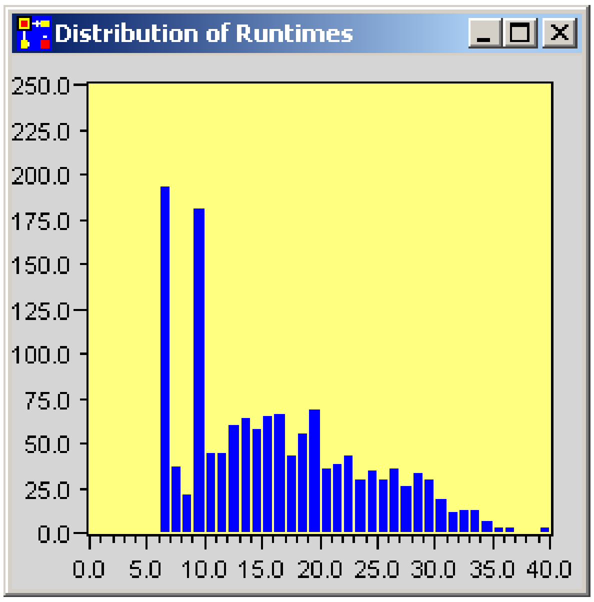


Figure 4.15: Distribution of runtimes

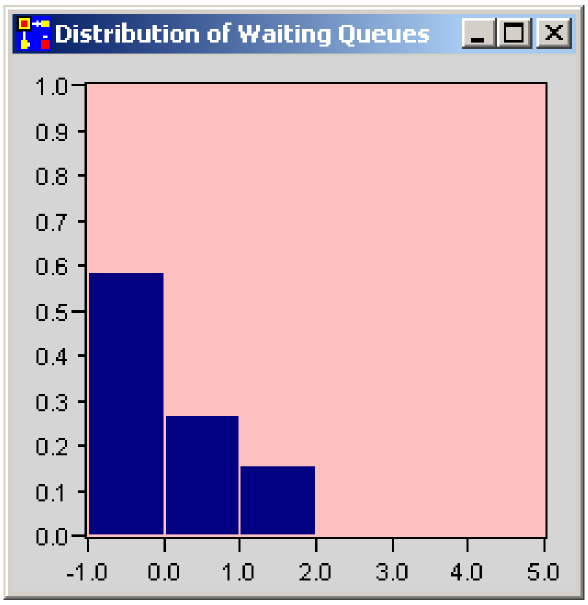


Figure 4.16: Distribution of waiting queues

Figure 4.17 on the next page shows an interface of the simulation model for carrying out experiments. You achieve this by arranging the PACE windows. The window in the upper right, with which the animation speed can be set, is opened with the menu option **animation speed** in the PACE navigator’s Simulator menu.

This completes the model, and it can be saved as described earlier .

It’s noteworthy that for creating this model, the interface description of the CarWash’ module was sufficient. This is the underlying requirement for preparing module librar-ies and for creating simulation models from pre-existing components (modules).

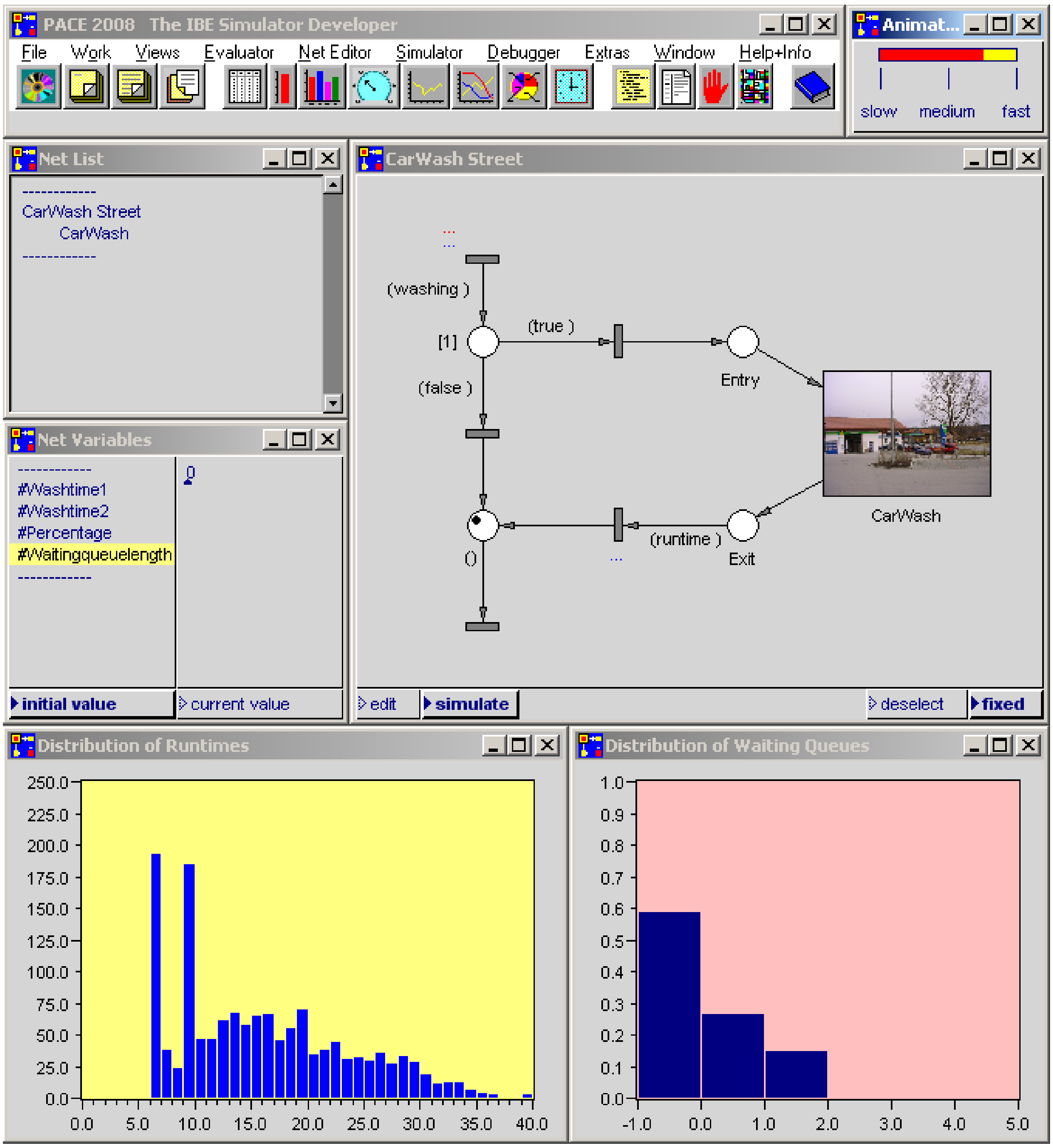


Figure 4.17: Work interface for experimenting

5. More Useful Features

In this chapter, we'll briefly discuss several useful features of PACE simply to introduce them. They are described in detail in the PACE 'Modeling and Simulation' manual.

5.1 Defining and Changing the Standard Representations of Net Components

The standard icons can be changed for any net. Changes are made using the **Extras** menu, **'icons'**, '**default icons**'. With '**change**' you can replace the pre-defined icons with any other pre-defined icons, or with icons stored in the list of individual icons. '**scale**' changes the size of the pre-defined icons.

You can also scale the standard representation of individual net elements. Select menu option **scale** in the **ri.MK** menu of the net element and enter the scaling factor in the window which opens. Then hit **return** or select **accept** in the input window's **ri.MK** menu.

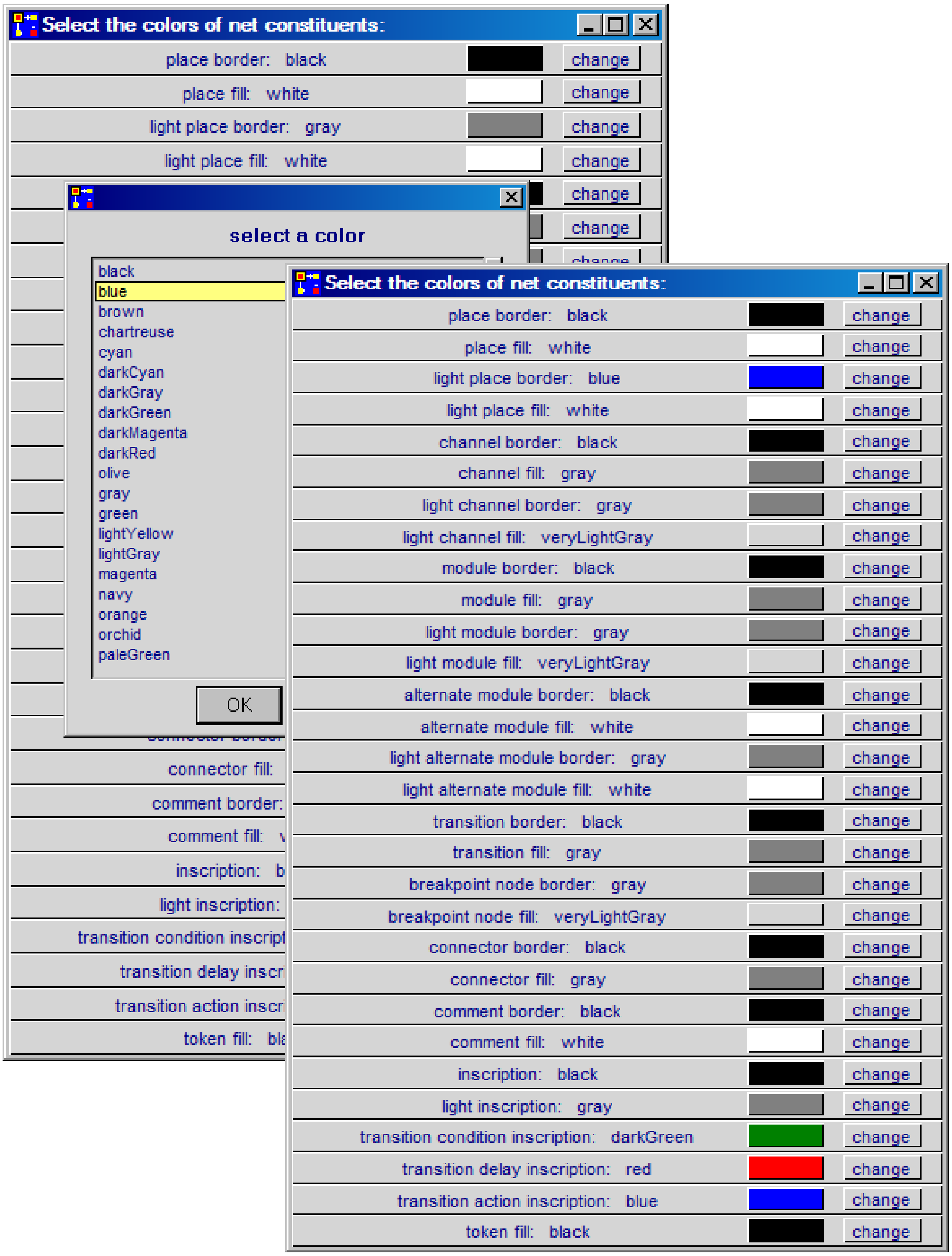
Additional changes of net components can be made with the **options** menu in the **Net-Editor** menu of the PACE navigator.

Here, for example, you can change the size of a connector's arrow points by using menu options **arrow length** and **arrow angle** in the **options** menu. **Element size** changes the absolute size of all elements in the net (with the exception of 'individual icons'). Finally, you can use **grid** to more precisely position the net elements.

5.2 Colored Nets

PACE offers the option of displaying the window and net components (icons, texts, etc.) in different colors. The available colors can be viewed by selecting '**show default platform colors**' under the **colors** menu option of the PACE navigator's **View** menu.

You can call up the function for changing the colors via the **view** menu**, colors, net constituents colors**. A window will open listing the most important components of PACE nets (Figure 5.1). The current color of the component is shown to the right of the identifiers. Use '**change**' to change the colors. A selection menu for the individual colors will appear. Mark your preferred color and press the ok-button. Using the scrollbar and the right edge of the window you can scroll to additional colors. The new color settings are stored in the model, i.e., in the so-called Image, so the model (Image) must be saved with **store image** after changing.



Select the colors of net constituents:

place border: black	⬛	change
place fill: white	⬜	change
light place border: gray	▨	change
light place fill: white	⬜	change
	⬛	change
	▨	change
		change
connector fill:		
comment border:		
comment fill: v		
inscription: b		
light inscription:		
transition condition inscript		
transition delay inscr		
transition action inscr		
token fill: bl		

select a color

black
blue
brown
chartreuse
cyan
darkCyan
darkGray
darkGreen
darkMagenta
darkRed
olive
gray
green
lightYellow
lightGray
magenta
navy
orange
orchid
paleGreen

OK

Select the colors of net constituents:

place border: black	⬛	change
place fill: white	⬜	change
light place border: blue	🟦	change
light place fill: white	⬜	change
channel border: black	⬛	change
channel fill: gray	▨	change
light channel border: gray	▨	change
light channel fill: veryLightGray	⬜	change
module border: black	⬛	change
module fill: gray	▨	change
light module border: gray	▨	change
light module fill: veryLightGray	⬜	change
alternate module border: black	⬛	change
alternate module fill: white	⬜	change
light alternate module border: gray	▨	change
light alternate module fill: white	⬜	change
transition border: black	⬛	change
transition fill: gray	▨	change
breakpoint node border: gray	▨	change
breakpoint node fill: veryLightGray	⬜	change
connector border: black	⬛	change
connector fill: gray	▨	change
comment border: black	⬛	change
comment fill: white	⬜	change
inscription: black	⬛	change
light inscription: gray	▨	change
transition condition inscription: darkGreen	🟩	change
transition delay inscription: red	🟥	change
transition action inscription: blue	🟦	change
token fill: black	⬛	change

Figure 5.1: Changing the color of a net component

5.3 Initialization code, break code, continuation code and termination code

These functions are called up with the **extra codes** menu option in the **Net-Editor** menu. Here you can enter Smalltalk-code to be used during initialization (**initialization code**), interruption (**break code**), continuation (**continuation code**), or termination (**termination code**) of a simulation run.

5.4 Special Bars for Simulation; Freezing Models

To simplify the use of ready models which can, for example, be used in daily practice by people without a knowledge of PACE, there are eight special bars (Executives) one can add to a model (e.g., see Figure 5.2). These bars contain control functions for execution of simulation runs so that models can be used without the need for menu skills.

To open an Executive in the **Simulator** menu, select the **install executive** menu option. You can make the Executive horizontal or vertical by choosing menu options **horizontal layout** or **vertical layout**. Then select one of the four possible Executives.

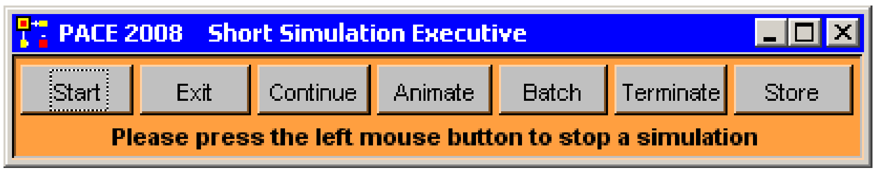

PACE 2008 Short Simulation Executive
Start
Exit
Continue
Animate
Batch
Terminate
Store
Please press the left mouse button to stop a simulation

Figure 5.2: Short horizontal navigator

To prevent inexperienced users from inadvertently changing models, models can be 'frozen' with an Executive in place. When a model is frozen, the PACE navigator and the net list are removed so no further change or expansion of the model is possible. In addition, most of the **ri.MK** menus will no longer be accessible, and all windows present at the moment of freezing cannot be closed. On the other hand, data can still be added, e.g., via the data input windows, by selection with the **le.MK** or by text inputs. This creates an easy usable application model which can be controlled only via an Executive and input windows.

5.5 Scenes

When creating larger models, the screen is almost always too small to show all the required windows (net windows, Executives, graphic in/output windows, etc.) at the same time. Usually, however, you don't need all of the windows concurrently. PACE allows you to group (into so-called Scenes) the windows which should be shown together, to make them appear or disappear together with the push of a button (**le.MK**). This allows you to prepare a well-organized user interface that is easy to overview.

Scenes are easy to assemble. The definition window is accessed via the **define scenery** menu option in the PACE navigator's **Extras** menu. It consists of two parts. Scene names can be entered in the No-Selection-Menu of the left window part (menu option **add**). When the Scene name is marked in the left part, you can use the No-Selection-Menu option **add** in the right part to enter the name of a window to be contained in the Scene.

We recommend, particularly for longer window names, that you do not simply type in the name, but proceed as follows: activate the window to be included in the Scene and call up its system menu with the **mi.MK** key. In the System menu, select the **relabel as...** option. An input window for the new window name will appear, in which the current name is marked. Now call up the input window's **ri.MK** menu, select **copy**, and finish by pressing **ok**. Then use the No-Selection-Menu of the right-hand window of the **define scenery** window and select **add**. In the query window for the name call up the **ri.MK** menu and select **paste**. Finally, close the input window with the **Return** key.

Once you have organized all Scenes, close the **define scenery** window and open the selection window for Scenes with the menu option **select scenery** in the **Extra** menu. There you will find a list of all Scenes. If you select (or de-select) a name in this window, the assigned Scene is either displayed or iconized.

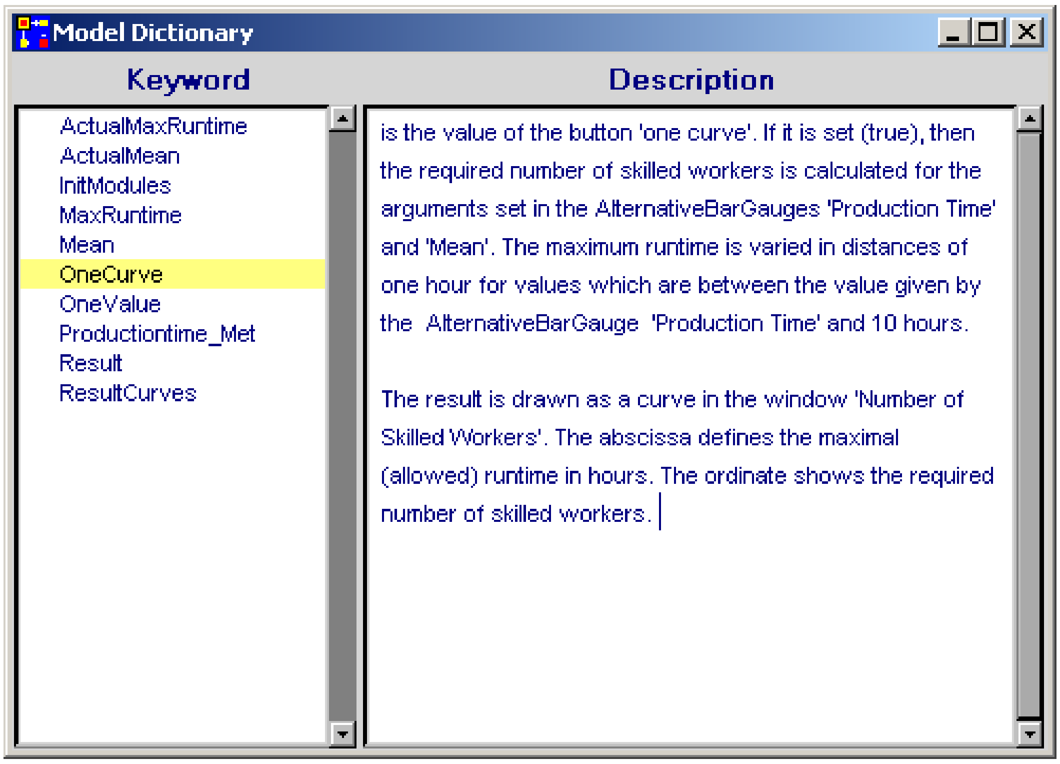
When designing interfaces, the usual procedure is to show a basic permanent set of windows which are then sometimes covered by Scenes. It should be noted that, during simulation, Scenes can also be programmed to be shown or hidden. This is particularly useful if different graphic evaluation windows are to be shown or for parameters input during simulation.

5.6 Model Dictionary

In later development or analysis of models it can be tedious to try to extract the exact meaning of program entities from the context. The model dictionary described here provides users an easy opportunity to capture this information during development of a model and to make it available with the model when the model is completed. [1]

You can open the model dictionary with the PACE navigator's Net-Editor menu, menu option **model dictionary**, or you can click on the fourth icon from the right which appears on the icon list under the main bar in the PACE navigator. It also opens when you mark a text in an Inscription, and then select menu option **dictionary** in the Inscription menu (**ri.MK**). In this case, PACE enters the text as an Entry in the Dictionary and marks the Entry. The user must then enter the descriptive text in the right part of the window (see Figure 5.3). The entry is saved to the Dictionary when the Entry marking is set back (by clicking the entry with **le.MK**).

[1] Additonal options for documenting models in PACE were described at the end of Section 4.1.



ActualMaxRuntime
ActualMean
InitModules
MaxRuntime
Mean
OneCurve
OneValue
Productiontime_Met
Result
ResultCurves

is the value of the button 'one curve'. If it is set (true), then the required number of skilled workers is calculated for the arguments set in the AlternativeBarGauges 'Production Time' and 'Mean'. The maximum runtime is varied in distances of one hour for values which are between the value given by the AlternativeBarGauge 'Production Time' and 10 hours.

The result is drawn as a curve in the window 'Number of Skilled Workers'. The abscissa defines the maximal (allowed) runtime in hours. The ordinate shows the required number of skilled workers.

Figure 5.3: Model dictionary with entries

6. Optimization

6.1 Optimization Procedures in PACE

In optimizing models, we distinguish between net and parameter optimization.

Net optimization compares the results generated from different nets with one another. From this we can deduce ideas for planning or for revising processes. In general, no automatic procedure can be set up for net optimizing, because for each model variant one needs to consider its actual feasibility in the real world, which can only be determined by a process engineer. We cannot know if an ’optimum’ solution we find really represents the optimum. Thus, we often designate ’net optimizing’ as ’net improvement’.

In contrast, for net parameter optimization we can prepare automatic procedures. These are also needed for the comparison of different alternatives in the net optimizing process described above. In parameter optimizing (we’ll use the short form, optimizing, here), a net with predefined process ways is manually or automatically executed with different parameters (e.g., number of resources) to find the parameter combination which is optimal with respect to the desired results (e.g., costs, runtime, etc.).

For optimizing nets, PACE offers three possible procedures from which to choose:

1. Repeated execution of the complete model
   Repeated execution of a model with changing parameters

2. Automatic repetition of partial models
   Repetition of partial models with program-controlled parameter setting and listing of results and/or graphic representation of the results. The optimum can be read visually from graphic representations (also called graphic or visual optimization).

3. Mathematical model optimization
   Applying mathematical optimization procedures (for example, Hill Climbing, Simplex, genetic procedures, threshold-acceptance) for automatically determining the optimum of a model.

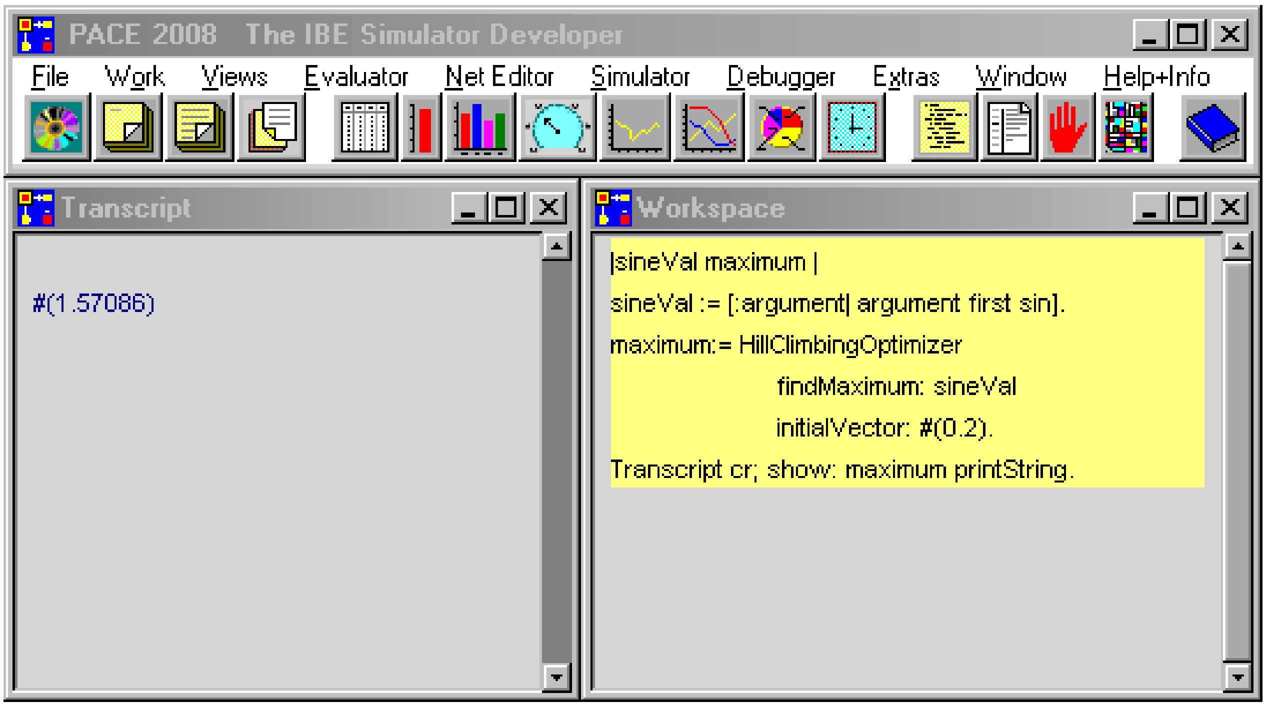
Which of these three procedures to use can only be decided by looking at the task assignments at hand. In general, you would look to mathematical optimization procedures for multi-dimensional optimizations to keep the number of model executions as small as possible.

In the following sections we will discuss all three net optimization procedures available in PACE using an example which has again been made as simple as possible so the description of the procedures, and not the example itself, takes centre stage. As an example, we’ll determine the maximum of the sine in the interval [0, π].

6.2 Optimizing Mathematical Functions

Before we discuss how to find the optimum of nets in the next sections, this section will cover the direct use of PACE’s optimization procedures in program code. In the present case, the optimum can also be determined without net modeling, e.g., with a Smalltalk program segment. Since it is sometimes a good idea to try out Smalltalk code in a so-called Workspace before using it in net inscriptions, we’ll cover those procedures in this section for practice.

In the PACE navigator’s Work menu select the menu options **transcript** and **work-space** one after another to open a Transcript- and a Workspace window (or ’Work-space’ for short). You can set the window as shown in Figure 6.1.



```
#(1.57086)
```

```
|sineVal maximum |
sineVal := [:argument| argument first sin].
maximum:= HillClimbingOptimizer
                    findMaximum: sineVal
                    initialVector: #(0.2).
Transcript cr; show: maximum printString.
```

Figure 6.1: Mathematical optimization in PACE

Enter the code shown in Figure 6.1 into the Workspace. In the first line, between the two vertical bars, the variables **sineVal** and **maximum** are declared. The second line contains the definition of a so-called Smalltalk-block:

```
[:argument| argument first sin ].
```

It starts with an opening square bracket "[" and ends with a closing square bracket "]". The argument of the block is given after the colon (in other programming languages this is identified as a "formal parameter"). If there are more arguments, they are entered in series, separated by a blank space and a colon. The end of the argument list is indicated by a vertical bar.

Next is the block’s code, in this case just one line. Since the optimization procedures prepare the data in the form of an array, the argument is an array. Therefore, the first element of the argument must be calculated at the beginning of the block code. This is done with the expression **argument first**; one could also (less elegantly) have written **(argument at: 1)**. The sine **sin** can then be applied to the element. The block is then assigned the local variable **sineVal**.

The next three lines call up one of the mathematical optimization procedures foreseen in PACE, the so-called HillClimbingOptimizer. The **sineVal** block described above is given as the calculation specification for the optimization. Then in the fifth line, a beginning value 0.2, with which the calculation of the optimum is to begin, is given in the form of an initialized Array #**(0.2)**. If multiple arguments are to be given, they are listed in parentheses separated by blank spaces. Less elegantly, one could also have written **Array with: 0.2**.

The HillClimbingOptimizer calculates the maximum value of the **sineVal** block and assigns the associated argument to the variable **maximum** in the form of an array. In the last line of this piece of code, this value is written to the Transcript window.

To execute this piece of code, first select it by sweeping over it with pressed **le.MK**. A more elegant way to select/deselect a piece of code is to click behind the last line of code in the window's empty space. The marked code is highlighted in yellow. Then call up the window's menu with **ri.MK** and select menu option **do it**.

6.3 Repeated Execution of the Whole Model

When executing a PACE model, the so-called initialization code is executed first. The procedure under discussion here is based on the fact that in the initialization code you can distinguish whether a simulation run is the first run of a model after manual initialization or a program-controlled repetition. This discrimination allows the initialization of program values in the first run, and their controlled change in subsequent runs.

Whether this is the first run or a further execution of the model can found out with the message:

```
self isRestarted
```

In the first execution right after initialization by the user, this query returns a value of **false**; in following executions it returns the value **true**.

The execution of a model or simulation run is automatically ended when no transition can fire again. It can also be forced in a net inscription with the message:

```
self terminate
```

If, instead of the terminate message, you use the message:

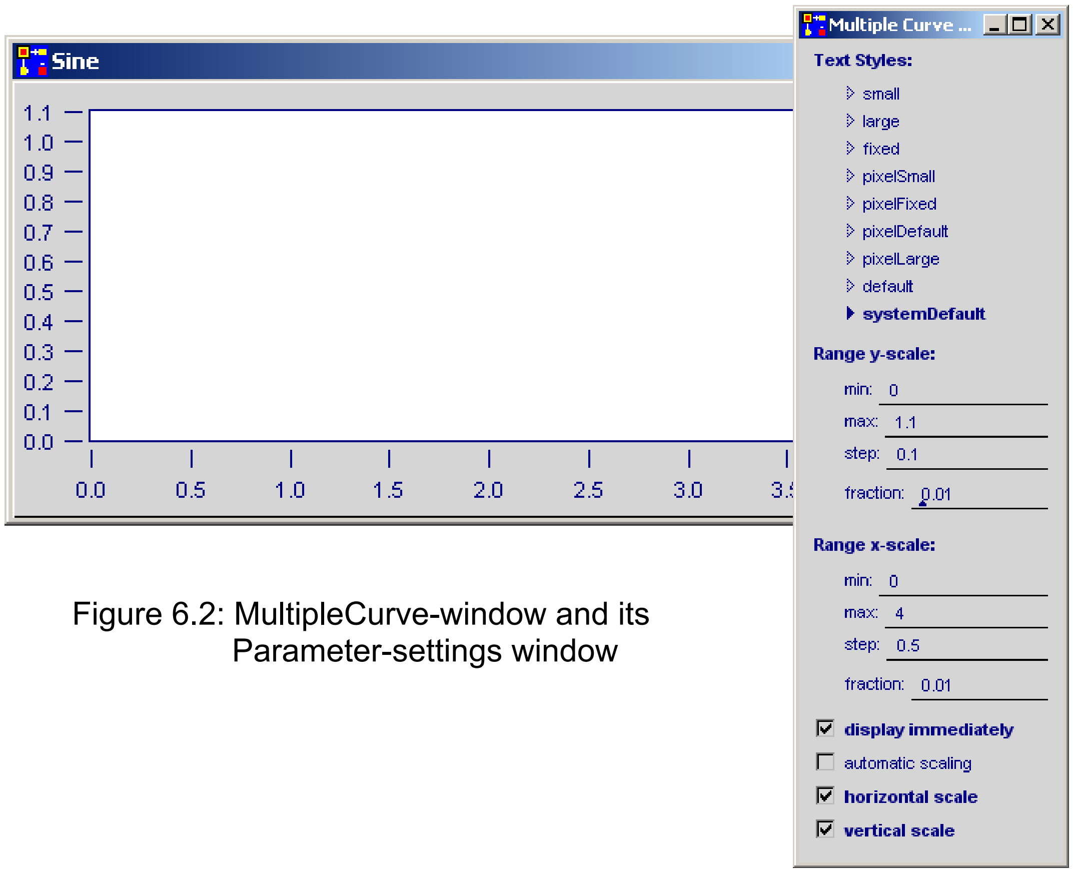
```
self restart
```

a new execution of the model will begin.

We'll start again with a new model, which we'll call 'Whole Model'. To show the results, open and scale a Message window and a MultipleCurve window.

The Message window is opened with menu option **message window** in the View menu of the PACE navigator. First an input window for the name of the Message window is opened, in which you'll enter the name **Maximumvalue**. Then press the **return** key of the keyboard. The Message window is now opened and, as described earlier, can be positioned and sized as desired.

The MultipleCurve window can be opened using the View menu, or with the icon button for multiple curves in the PACE navigator (eighth icon from the right). As described before, enter the window name **Sine** with the **mi.MK** after it opens, and call up the window menu with the **ri.MK**. There, select the menu option **parameter**. A parameter window will open in which you can, among other things, set the scaling of the window. Set the parameters as shown in Figure 6.2. After hitting **return**, the scaling of the window is updated.



Text Styles:

- small
- large
- fixed
- pixelSmall
- pixelFixed
- pixelDefault
- pixelLarge
- default
- **systemDefault**

Range y-scale:

min: 0
max: 1.1
step: 0.1
fraction: 0.01

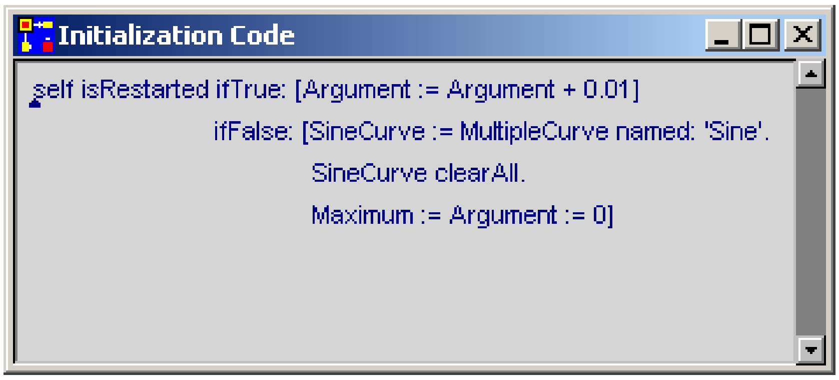
Range x-scale:

min: 0
max: 4
step: 0.5
fraction: 0.01

[✓] **display immediately**
[ ] automatic scaling
[✓] **horizontal scale**
[✓] **vertical scale**

Figure 6.2: MultipleCurve-window and its
Parameter-settings window

Next, enter the initialization code. Click on the corresponding icon on the PACE navigator (fifth icon from the right) and enter the code as shown in Figure 6.3.

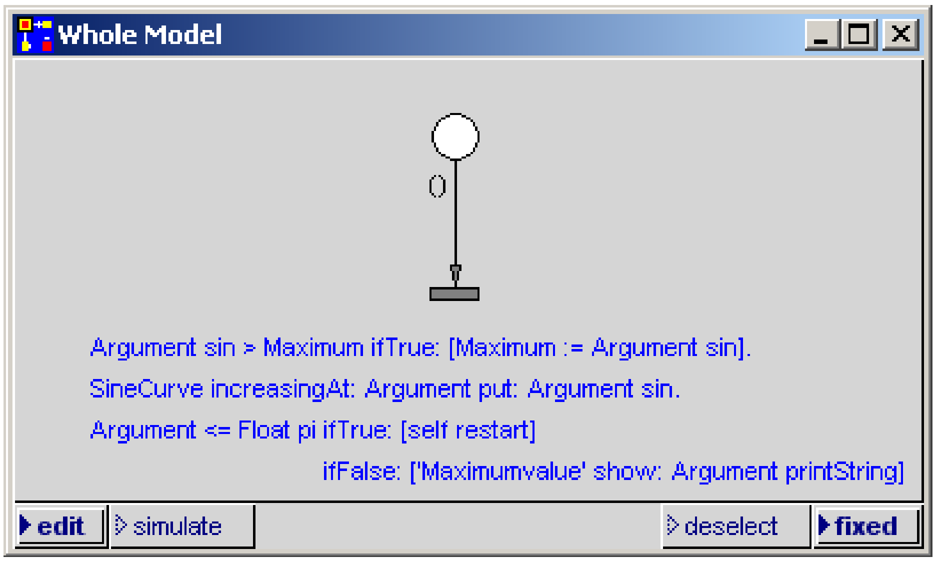


```
self isRestarted ifTrue: [Argument := Argument + 0.01]
                 ifFalse: [SineCurve := MultipleCurve named: 'Sine'.
                          SineCurve clearAll.
                          Maximum := Argument := 0]
```

Figure 6.3: Initialization code for Example 1

The initialization code consists of a conditional message. Depending on whether the isRestarted message delivers true or false, the corresponding block in square brackets is executed. In the true-branch of the message, the argument for the next run of the model is increased. The false-branch carries out the initialization of the global variables **Argument, Maximum** and **SineCurve**. With the first two lines of the false-branch, the MultipleCurve window you've just created is attached to the model, and any possible content from previous simulations is erased.

After you enter the program code, call up the text window menu with the **ri.MK** and select **accept**. Three query windows will appear sequentially to specify that each of the variables named above are global variables.



```
Argument sin > Maximum ifTrue: [Maximum := Argument sin].
SineCurve increasingAt: Argument put: Argument sin.
Argument <= Float pi ifTrue: [self restart]
                 ifFalse: ['Maximumvalue' show: Argument printString]
```

Figure 6.4: Net of Example 1

The net of this example is quite simple and is shown in Figure 6.4. To start the net execution a so-called initial token, which is generated at initialization, must be inserted in the place. This is done as follows:

Use the **ri.MK** to call up the menu of the place and select **initial tokens**. The window shown in Figure 6.5 is opened to define the initial token.

Here you can set as many initial tokens with attributes and initial icons as desired. In the present case, only one token (without attributes) is needed. So, in the left part of the window (tokens), select menu option **add** in the **ri.MK** menu. This window will look like the one shown in Figure 6.5 and may be closed.

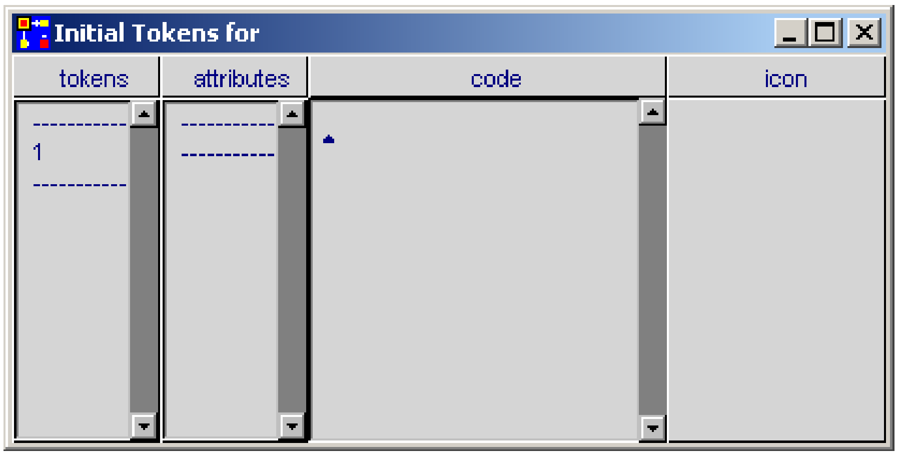
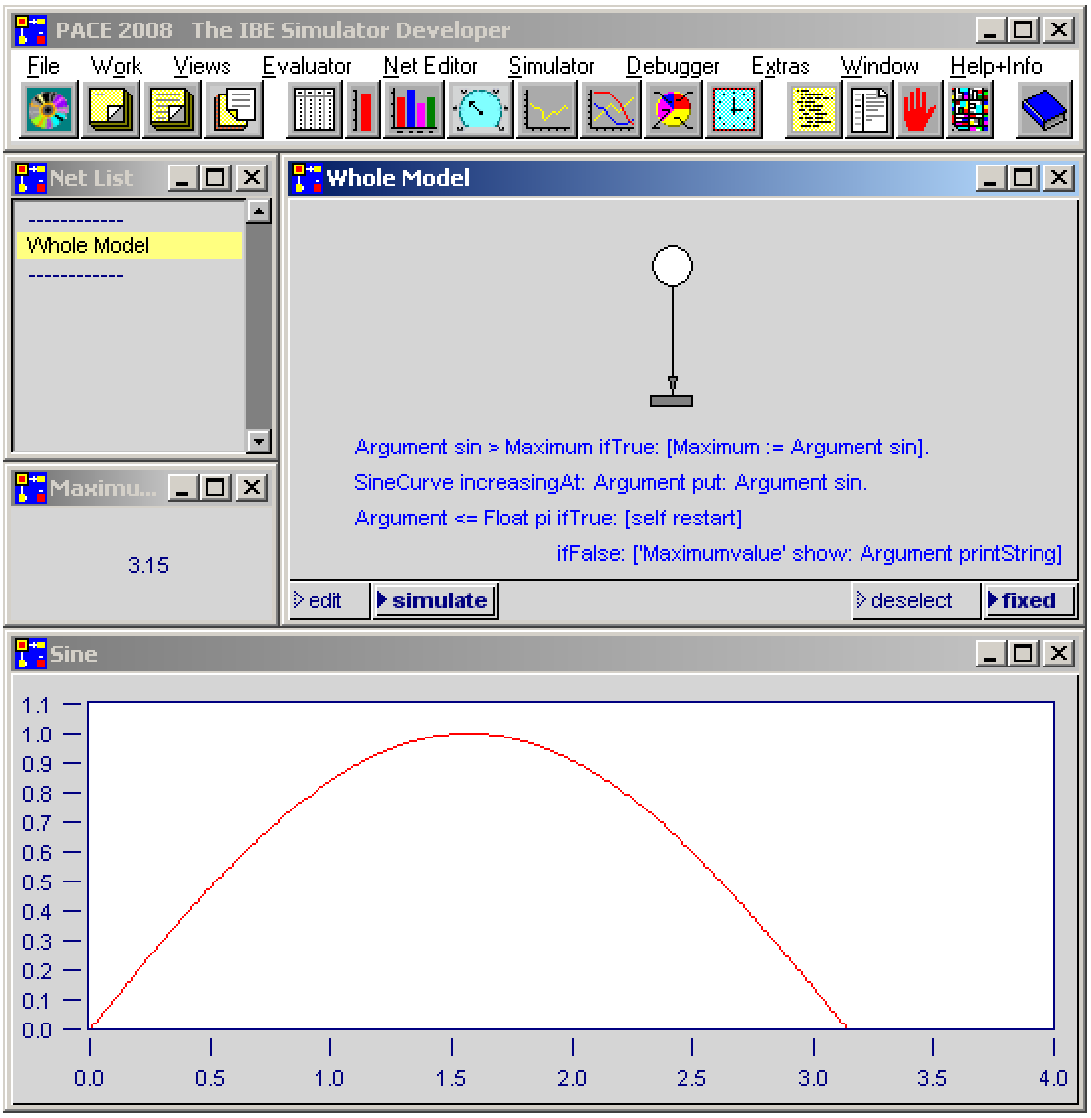


Figure 6.5: Window for agreement of initial tokens

The model is now complete and can be executed as described earlier.

It is always useful to arrange the windows to be shown in a model in a working-surface. If you save and leave the model (**leave PACE** in the File menu), it will open again when reloaded exactly as it was at that point. In the present case, one can, for example, compose the windows as shown in Figure 6.6.



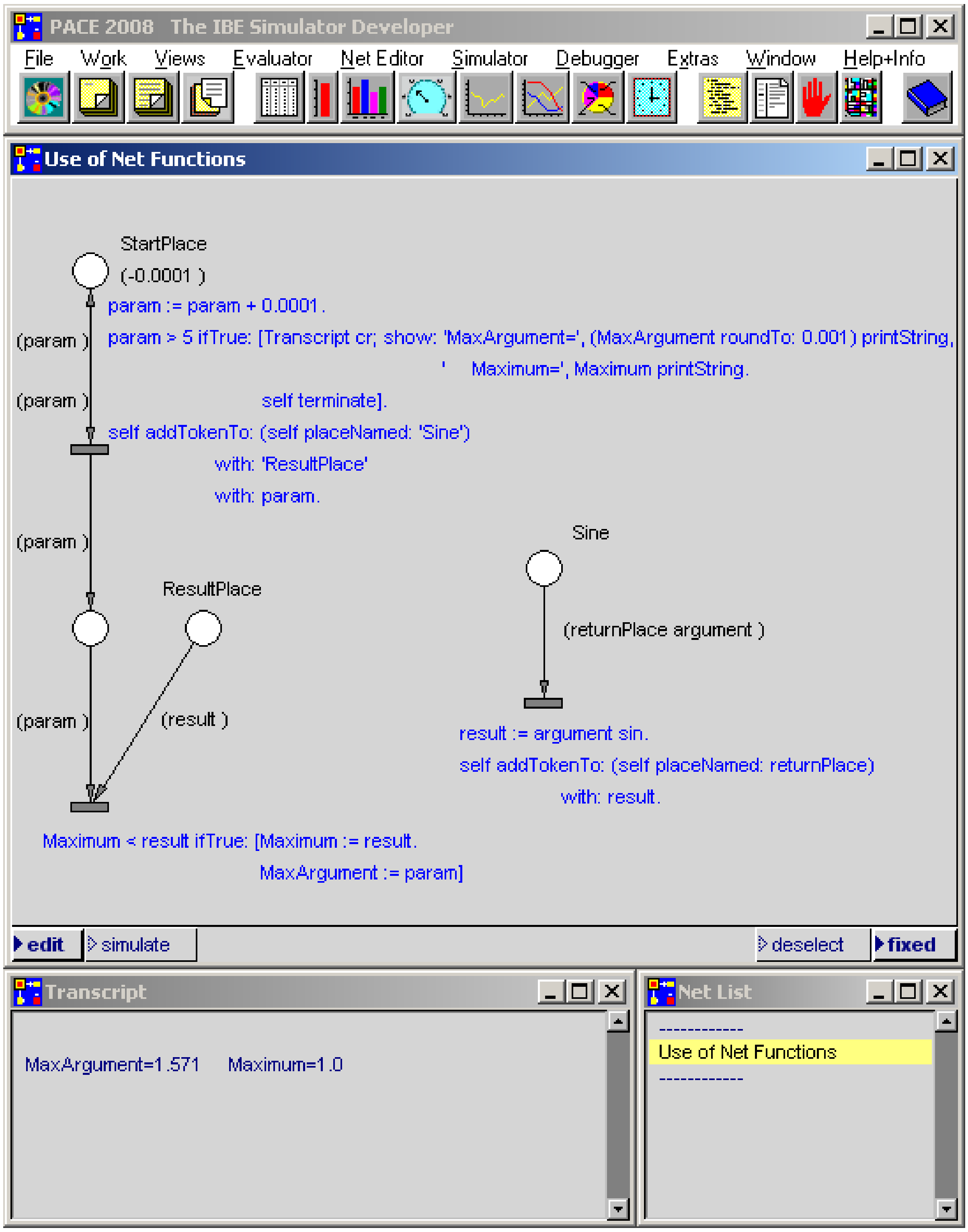
```
Argument sin > Maximum ifTrue: [Maximum := Argument sin].
SineCurve increasingAt: Argument put: Argument sin.
Argument <= Float pi ifTrue: [self restart]
                     ifFalse: ['Maximumvalue' show: Argument printString]
```

Figure 6.6: Working-Surface for Example "Whole Model"

6.4 Using PACE Net Functions

Whenever there is a functional relationship between arithmetic values, a set of arguments and a result, you can ask for the arguments for which the result is an extreme value. The optimization of nets is based on so-called PACE net functions. If parts of a model are represented as PACE net functions, these can be called up from other parts of the models and thus be supplied with input arguments.

Figure 6.7 shows a workspace in which the maximum of the net function Sine is calculated and output to a Transcript window. Since most modeling steps have already been described in previous sections, we'll go into detail only on steps which have not appeared before.



```
param := param + 0.0001 .
param > 5 ifTrue: [Transcript cr; show: 'MaxArgument=', (MaxArgument roundTo: 0.001) printString,
                          '      Maximum=', Maximum printString.
                   self terminate].
self addTokenTo: (self placeNamed: 'Sine')
           with: 'ResultPlace'
           with: param.
```

```
result := argument sin.
self addTokenTo: (self placeNamed: returnPlace)
           with: result.
```

```
Maximum < result ifTrue: [Maximum := result.
                          MaxArgument := param]
```

```
MaxArgument=1.571     Maximum=1.0
```

Figure 6.7: Working-Surface of Example "Use of Net Functions"

The net function in the simple example at hands consists of only one place and one transition (see the right subnet in the net window). It is called up when the partial net on the left side sets a token with parameters in the input place **Sine**. Since no fixed return place is specified, we still must provide the return-place **ResultPlace**, in which a token with the result, a function value, is to be placed.

A net function is called with the message:

self addTokenTo: aPlace with: argument1 with: argument2.

If the with:-entries are missing, a token without arguments is generated. Up to four with:-arguments may be entered. The first argument should always be the place into which the token is to be set. A place is attached with the message: self placeNamed:. As an argument, the name of the place must be entered in single quotation marks, that is, in the form of a Smalltalk-string. The return from the net function is also carried out with the addTokenTo: message.

In the left subnet, two more new modeling steps should be explained: the installation of an initial token with the initial argument –0.0001, and the Transcript-message, which will write the result in the Transcript window.

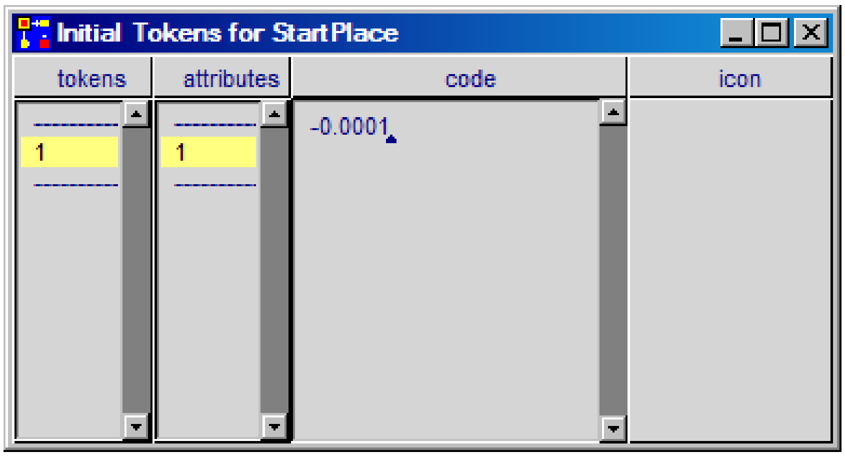


Figure 6.8: Agreement of starting tokens with values

To install the initial token, we'll mark the place **StartPlace** and select menu option **initial tokens** in its **ri.MK** menu. In the window which opens, install a token in the left partial window with the **add** function, as in the previous section. This token will carry one argument. Select the token identifier 1 in the **tokens** portion of the window, and call up the **ri.MK** menu in the **attributes** portion of the window. This includes only the menu option **add**, which you will select. Now enter the value –0.0001 in the **code** portion of the window and use **ri.MK, accept** to accept it. The window will now look as in Figure 6.8.

The Transcript message which writes the results in the Transcript window consists of the keyword Transcript followed by what is normally called a cascade. A cascade is a series of Smalltalk-messages separated by semicolons. They all relate to the same object, in this case to the Text-Output window given by Transcript. Instead of the Transcript message shown in the window, one could also have written the following two messages:

```
Transcript cr.
Transcript show: 'MaxArgument=', (MaxArgument roundTo: 0.001) printString,
                    '     Maximum=', Maximum printString.
```

The first of these two messages outputs a carriage return.

A string-object is always expected after the show:-message in the second line. Using the comma operator, different strings can be lined up (concatenated). Numbers can be converted to strings with **printString**. Finally, the **roundTo:**-message rounds the number to its left (the so-called receiver) to a multiple of the number at right. MaxArgument is calculated to four decimal places after the period, and is rounded to three decimal places after the period before output. The result of the Transcript message is shown in the Transcript window in Figure 6.7.

6.5 Mathematical Optimization of Models

The procedures in the previous sections have the disadvantage that in complex models, especially those where multiple arguments occur, a large number of model executions are required and the optimisation process can become very compute-intensive. It would be good, therefore, to reduce the number of model executions by using an optimizer.

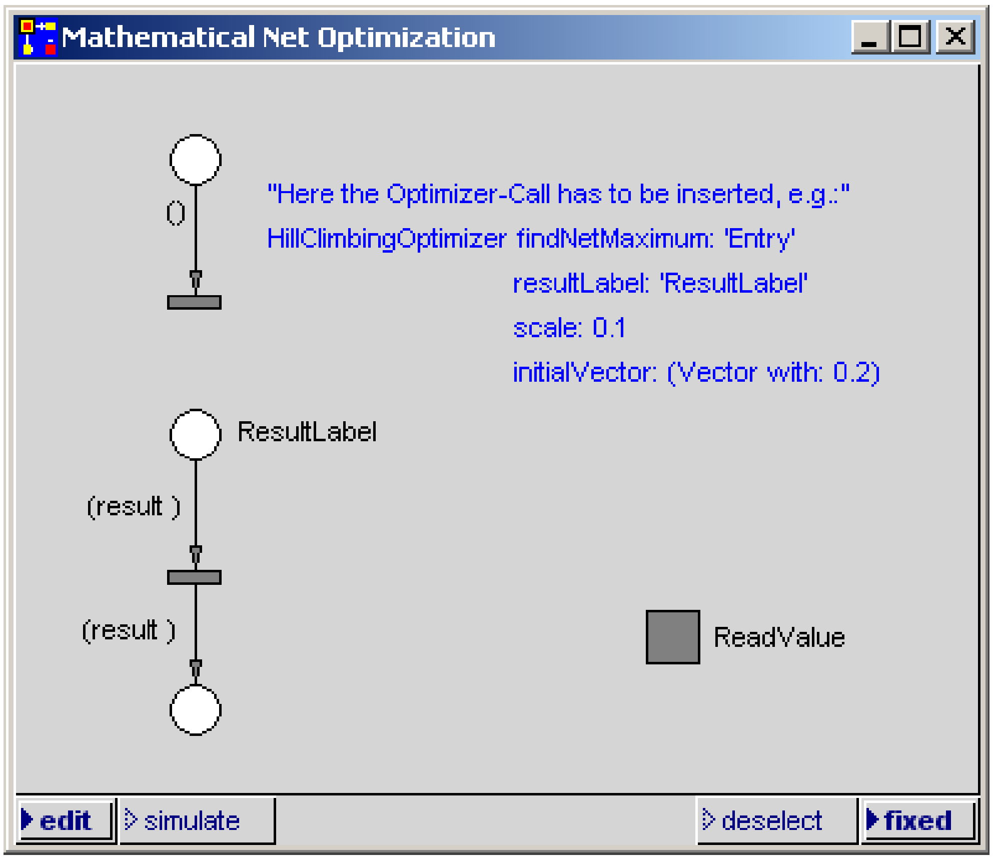
In PACE the following optimizers with countless options for optimizing mathematical and net functions have been implemented, which in part can also be used in combination:

Hill-Climbing-Optimizer
GeneticOptimizer
ThresholdAcceptingOptimizer
Simplex-Optimizer.

The first three of these can be scaled, i.e., can be implemented with reduced precision. Here we'll describe the use of a HillClimbingOptimizer for the selected simple example. In Chapter 7 we'll discuss the three scalable optimizers for optimizing a production.

We'll start again with a new model, which we'll call **Mathematical Net Optimization**. To implement an optimizer, PACE provides the library-module **NetOptimizer**, which may be placed in the empty net window (**ri.MK, restore module**). After insertion, mark the **NetOptimizer** module and select the **refine** menu option in its **ri.MK** menu. Press the **yes** button in the window which opens.

This releases the NetOptimizer module, and its net appears in the net window. Normally, the net elements and inscriptions must be positioned in the net window anew to make a net overview easy. The result is shown in Figure 6.9.


Mathematical Net Optimization

"Here the Optimizer-Call has to be inserted, e.g.:"
HillClimbingOptimizer findNetMaximum: 'Entry'
resultLabel: 'ResultLabel'
scale: 0.1
initialVector: (Vector with: 0.2)

Figure 6.9: Net window after executing refine

In this case, no scale factor is needed for optimizing. It is usually removed to reduce computer time when model parameters are only roughly known and/or the execution of the model requires a great deal of time. Thus, the line

scale: 0.1

is either eliminated or commented out (by putting a quotation mark " in front of the first character in the line and after the last character in the line). To do this, mark the action inscription with the **le.MK** and select menu option **inspect** in the inscriptions **ri.MK** menu. A text window will open in editmode to carry out the change. Then call up the **ri.MK** menu of the text window and select **accept**.

The PACE net function to be called up is contained in the **ReadValue** module. Mark the module and select the menu option **subnet** in its **ri.MK** menu. The module's net window shown in Figure 6.10 will open. The net must still be expanded by adding the nets for calculating the results.

For the simple example we're using here, it is only necessary to minimally expand the net (Figure 6.11). The required modeling steps have already been described earlier, but the two action inscriptions remain to be explained.

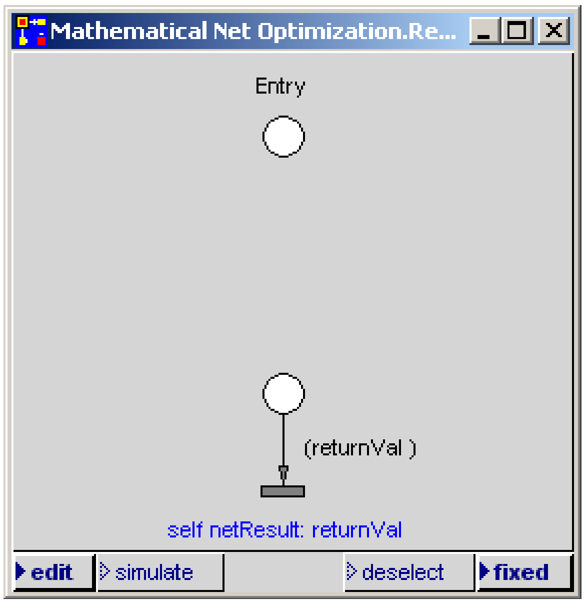

Mathematical Net Optimization.Re...
Entry
(returnVal)
self netResult: returnVal
edit
simulate
deselect
fixed

Figure 6.10: Original Net of the ReadValue module

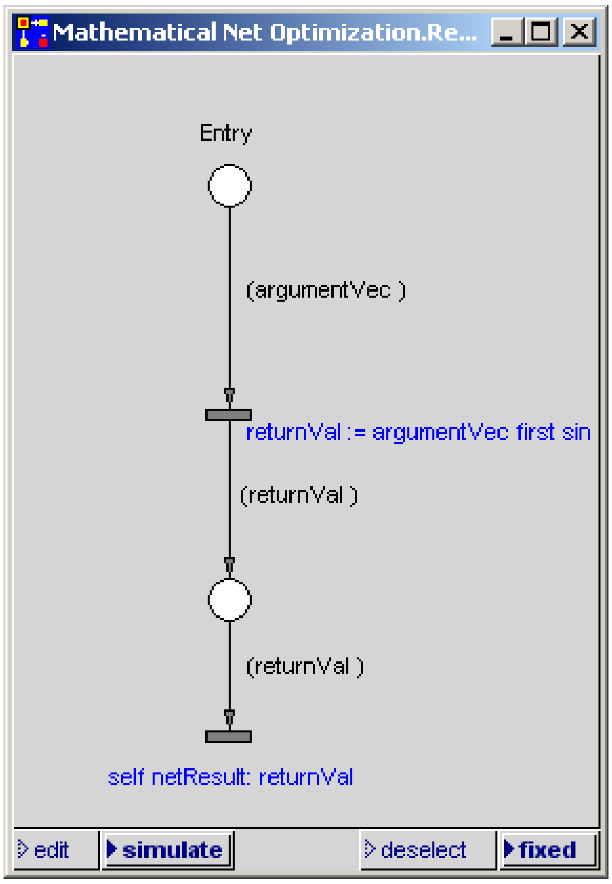

Mathematical Net Optimization.Re...
Entry
(argumentVec)
returnVal := argumentVec first sin
(returnVal)
(returnVal)
self netResult: returnVal
edit
simulate
deselect
fixed

Figure 6.11: Completed net for the ReadValue module

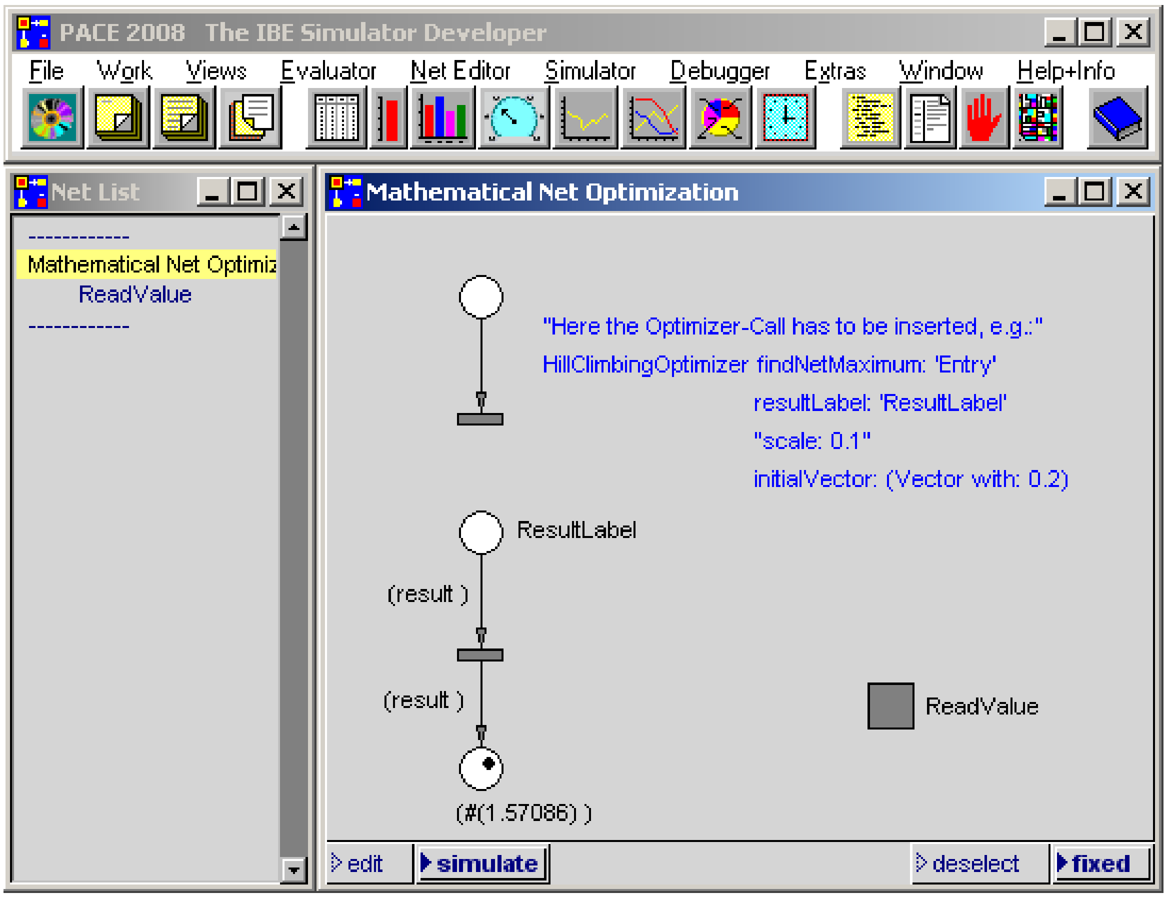
After the comments in Section 6.2 it is not necessary to explain the inscription for the calculation of the sine. With the net optimizers, too, the arguments are transfered in the form of a vector (vector is a subclass of array and can thus be processed by the same messages).

To return the result of a net function the following message is used:

self netResult: returnVal.

returnVal is returned to the optimizer, which continues to call the net function with changing arguments until the result achieves the desired precision. The result is then attached as an attribute to a token which is set to the return-place **ResultLabel** specified when the Optimizer was called up.

Figure 6.12 again shows the final workspace with several simplifications in the **Mathematical Net Optimization** net window and a result token.


PACE 2008 The IBE Simulator Developer
File Work Views Evaluator Net Editor Simulator Debugger Extras Window Help+Info
Net List

Mathematical Net Optimiz
ReadValue

Mathematical Net Optimization
ResultLabel
(result)
(result)
(#(1.57086))
ReadValue
edit
simulate
deselect
fixed

"Here the Optimizer-Call has to be inserted, e.g.:"
HillClimbingOptimizer findNetMaximum: 'Entry'
resultLabel: 'ResultLabel'
"scale: 0.1"
initialVector: (Vector with: 0.2)

Figure 6.12: Working-Surface for mathematical model optimization

7. Example: Optimizing a Production

7.1 Conceptual Formulation

A model is to be produced to calculate the number of skilled workers required for the production or processing of a given product. The following requirements must be considered:

- **Orders**
  The orders for each product arrive exponentially distributed. The mean value of the distribution should lie within the (left open) interval (0, 10] of hours and is set by the user before the simulation.

- **Production characteristics**
  Processing an order requires a fixed amount of time, which may be set prior to the simulation in the (left open) interval (0, 10] of hours. One order is performed by exactly one skilled worker.

- **Runtime**
  The time allotted for order processing is limited and is specified before a simulation in the (left open) interval (0, 10] hours. The specified execution time should be overstepped only by a very small percentage of the orders.

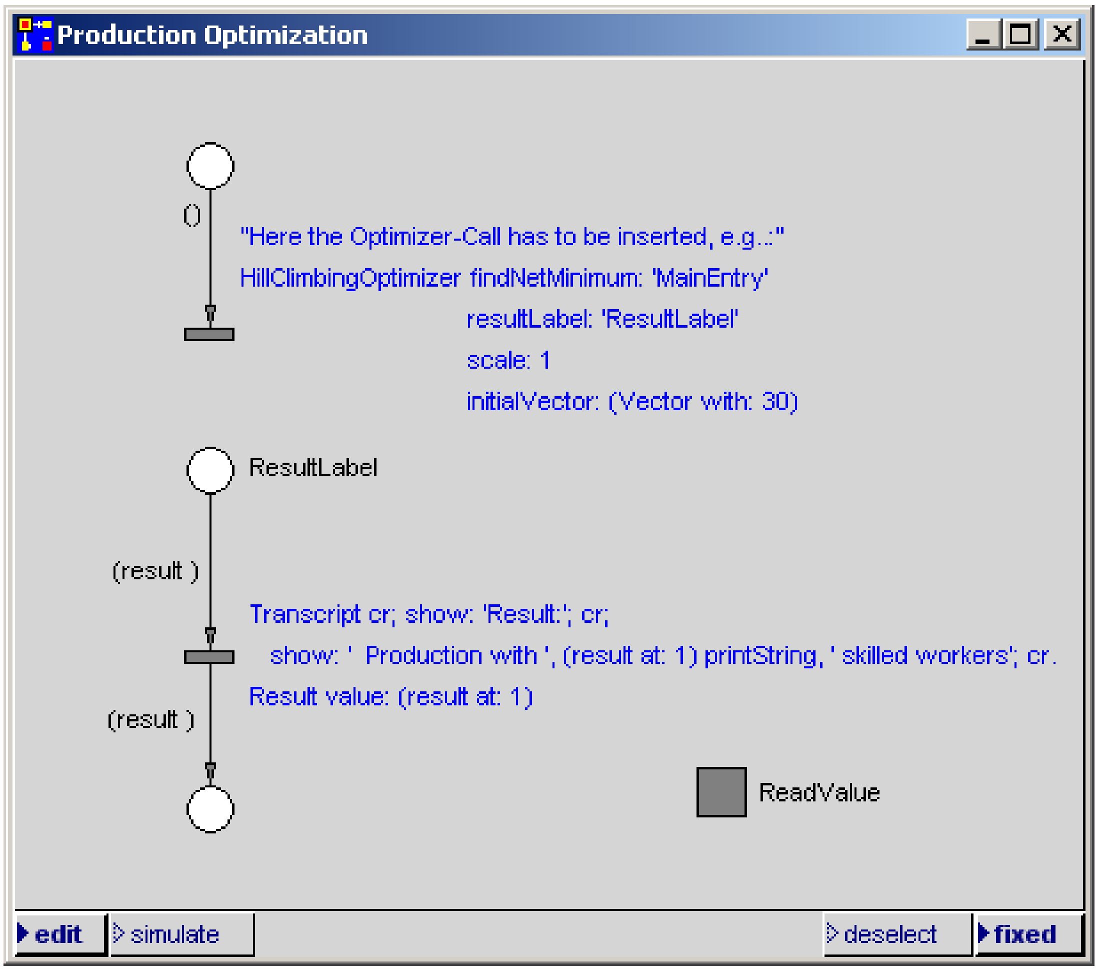
7.2 Creating a model

While in earlier sections we have described the (syntactical) construction of the models step-by-step, in the following we will present only an abbreviated representation, assuming that the user will model the demonstration graphics and texts. Only those individual steps which have not been presented in earlier sections, or expansions of earlier descriptions, are presented more explicitly. This does not, of course, include the internal (semantic) description of the model, which is presented using the graphic and text illustrations.

Begin once again with a new model, which has the name ProductionOptimization. As described in Section 6.5., an optimizer is called up in the network window. This brings you to the graphic shown in Figure 6.9 with a new window name.

The action codes must be changed in this network window, as shown in Figure 7.1. The optimum should be based on 30 skilled workers. Scaling is done in whole numbers, as each order is performed by exactly one worker.

Output of the result is produced by the second action code. First, using Transcript with a cascade, the result is written to a transcript window. Then the result is output to a yet-to-be-defined AlternativeBarGauge, which is saved in the global variable **Result**. The AlternativeBarGauge window (its so-called View) is opened lower down. In translation of the action code (Menu option **accept** in the **re-MT**-Menu of the Actionscode), **Result** is declared as a global variable.



```
"Here the Optimizer-Call has to be inserted, e.g.:"
HillClimbingOptimizer findNetMinimum: 'MainEntry'
                      resultLabel: 'ResultLabel'
                      scale: 1
                      initialVector: (Vector with: 30)
```

```
Transcript cr; show: 'Result:'; cr;
  show: '  Production with ', (result at: 1) printString, ' skilled workers'; cr.
Result value: (result at: 1)
```

Figure 7.1: Calling up the optimizer for production

Instead of a HillClimbingOptimizer, you may choose to use one of the other two scalable optimizers in PACE. In the case of a genetic optimizer, the action code could look as follows:

```
GeneticOptimizer findNetMinimum: 'MainEntry'
                 resultLabel: 'ResultLabel'
                 scale: 1
                 origin: (Vector with: 1)
                 range: (Vector with: 30)
```

The minimum, i.e., the smallest number of skilled workers, with which the above conditions can be met should fall between the boundaries 1 (origin) and 30 (range).

For threshold acceptance, the code would look like this:

```
ThresholdAcceptionOptimizer findNetMinimum: 'MainEntry'
                            resultLabel: 'ResultLabel'
                            scale: 1
                            initialVector: (Vector with: 10)
                            initialThreshold: 3
```

It is possible to integrate all three versions within the transition and to use them at will.

In the transition's **ri.MK**-Menu, select the menu option **code versions** and select **action code** in the sub-menu. This opens a window in which the versions of the action code can be managed. In the left side of the window, call up the **ri.MK**-Menu and select menu option **add**. This opens an input window for the version name under which the current code is to be saved (Figure 7.2).

If you want to replace the current Transition action code with a saved code, highlight the name of the code in the left window and select menu option **restore** in the menu of the left window (**ri.MK)**.

The program uses the current date and time as a default name. In the case at hand, we are not considering management of the versions, but rather saving the three alternative optimizer calls. So the default name is replaced with the name HillClimbing.

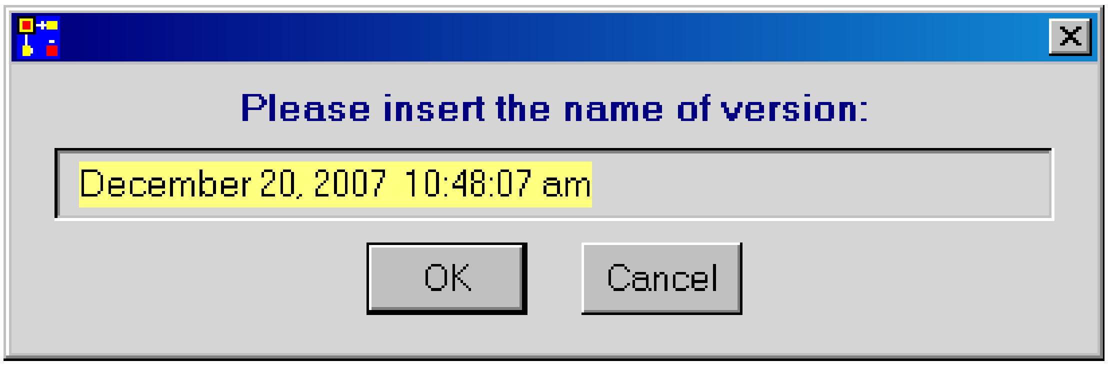
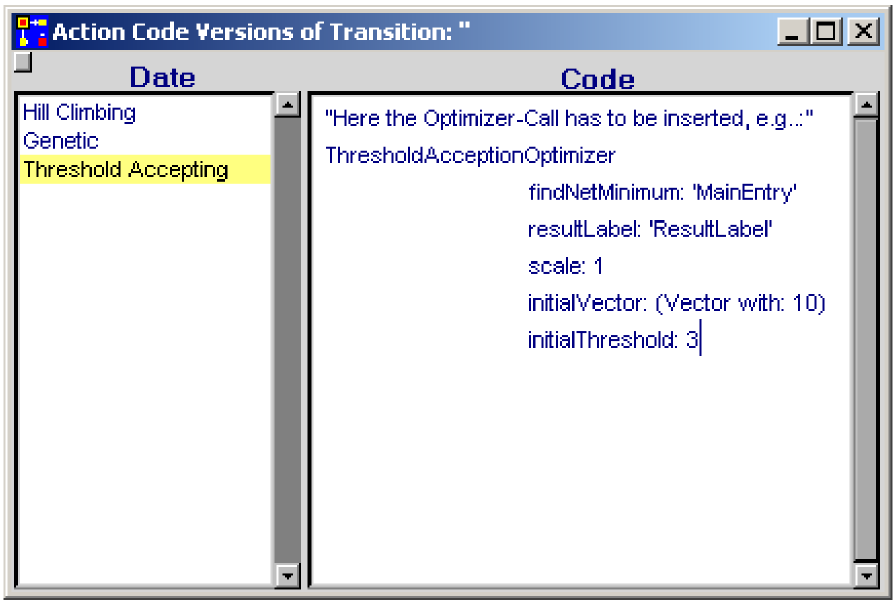

Please insert the name of version:
December 20, 2007 10:48:07 am
OK
Cancel

Figure 7.2: Inputting the version name

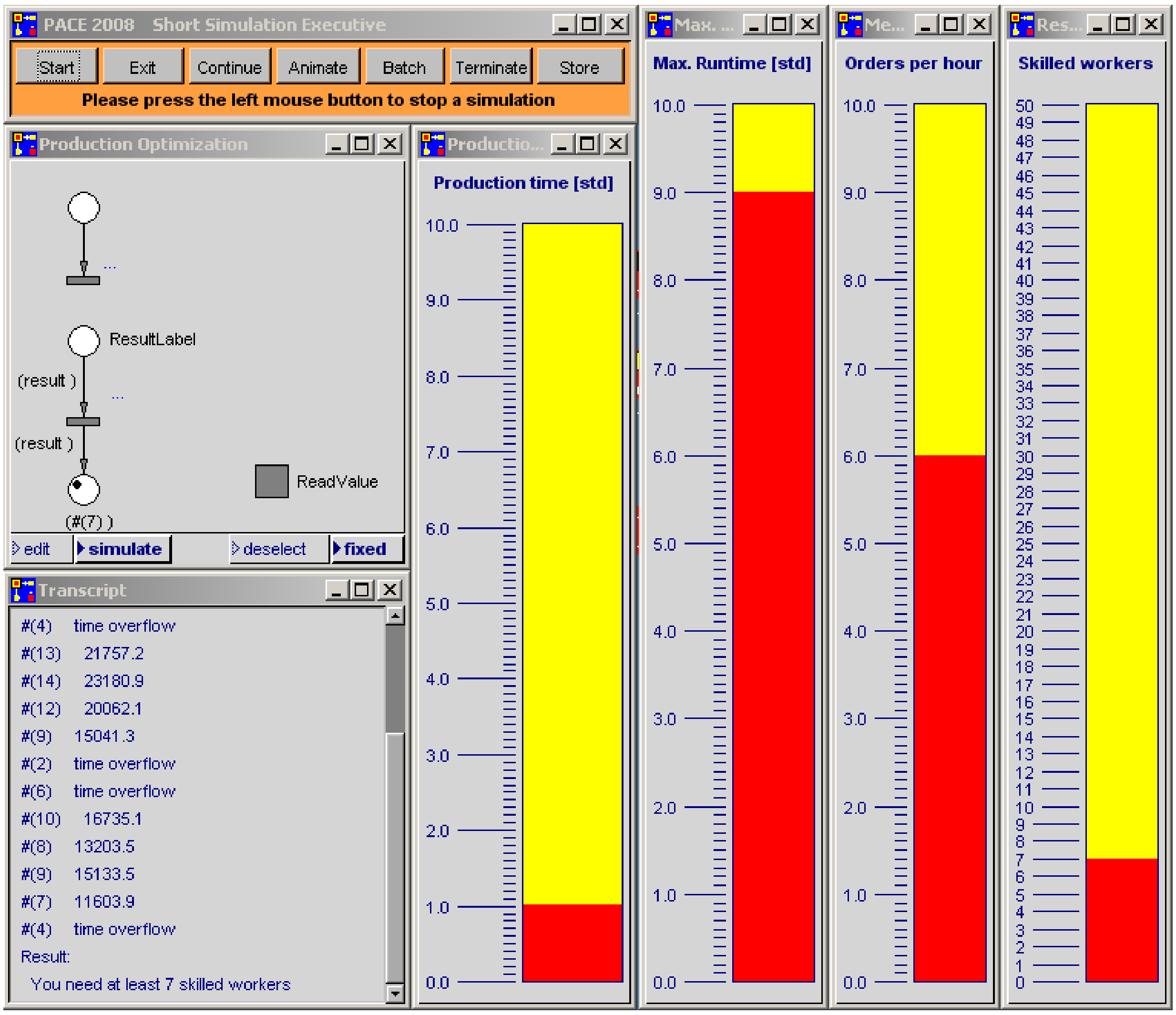
Similarly, one can save the other two optimization calls, resulting in the window shown in Figure 7.3.



```
"Here the Optimizer-Call has to be inserted, e.g.:"
ThresholdAcceptionOptimizer
                    findNetMinimum: 'MainEntry'
                    resultLabel: 'ResultLabel'
                    scale: 1
                    initialVector: (Vector with: 10)
                    initialThreshold: 3
```

Figure 7.3: Agreement of initial tokens and values

To reduce the size of the window shown in Figure 7.1 on the desktop, both transition codes can be hidden with the Three-Dots-Menus.



```
#(4)    time overflow
#(13)    21757.2
#(14)    23180.9
#(12)    20062.1
#(9)    15041.3
#(2)    time overflow
#(6)    time overflow
#(10)    16735.1
#(8)    13203.5
#(9)    15133.5
#(7)    11603.9
#(4)    time overflow
Result:
 You need at least 7 skilled workers
```

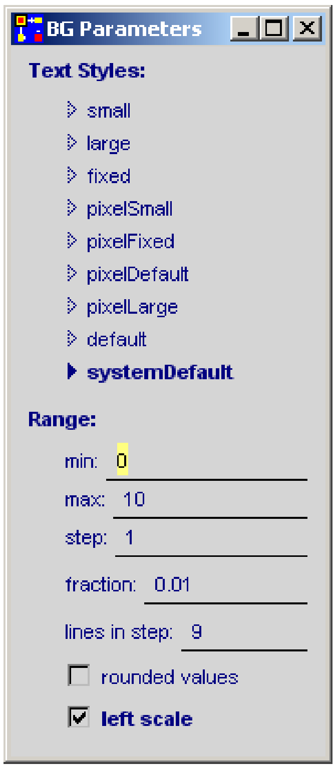
Figure 7.4: Work interface for production optimizing

Next it's time to create the work interface for the optimization model (Figure 7.4).
To guide the simulation, a short horizontal executive bar is used. In addition to the AlternativeBarGauge **Result**, three additional AlternativeBarGauges (**Production time, Max. runtime** and **Mean**) are used to set and manage the conditions to be considered in the production.

The scales and other characteristics of an AlternativeBarGauge may be set in its Parameter window. In the **ri.MK** window select the menu option **parameter**. The parameter window for **Production time** is shown in Figure 7.5.

Use **Text Styles** to select the fonts for the scaling.

The **Range**-specifications set the values of the scale. **min** and **max** set the minimum and maximum values in the scale. **step** sets the value increments for the scale. **frac-**


BG Parameters

Text Styles:

- small
- large
- fixed
- pixelSmall
- pixelFixed
- pixelDefault
- pixelLarge
- default
- **systemDefault**

Range:

min: 0

max: 10

step: 1

fraction: 0.01

lines in step: 9

☐ rounded values

☑ **left scale**

Figure 7.5: Parameter window for an AlternativeBarGauge

tion specifies the precision with which values are to be represented. Finally, using **lines in step**, you can specify how many lines should be added between "steps" to refine the scale representation. Checking **rounded values** allows you to specify that only multiples of the „step“ value may be set when adjusting a value. **left scale** allows you to control if the scale will be to the right or left of the bar.

We recommend that you experiment with the Parameter windows for different graphics which can be shown via the View menu on the PACE main navigator. This will help you explore how the different settings work.

As shown in Figure 7.4, you can add a one-line comment above the bar in AlternativeBarGauges. Call up the BarGauge menu again and select **label**. Enter the comment text into the input window which opens and save it with the **Return** key.

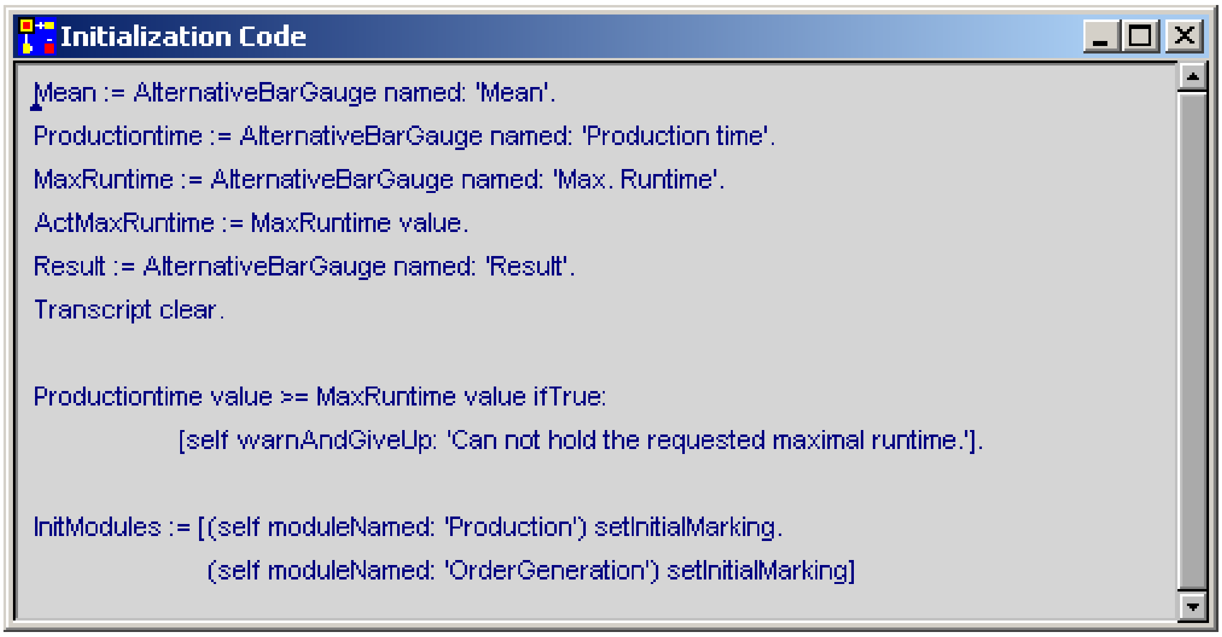
If the parameter setting is needed later in another model, it can be saved and later loaded into another AlternativeBarGauge. This is done with menu options **store** and **restore** in the menu of the Bar window. **store** writes the scale into a file in the **ioutils** subdirectory of the directory from which PACE was started. **restore** opens a selection window with the available scales. Most in/output windows in PACE use these functions to ease scaling of graphics.

The AlternativeBarGauges shown in Figure 7.4 are attached to the model, as before, using the initialization code. Figure 7.6 shows the initialization code.

In the first five lines, the four AlternativeBarGauges are assigned to four global variables, and the maximum allowable runtime is stored in the global variable **ActMaxRuntime**. The global variables are later used to access the BarGauges. The sixth line clears the Transcript window so output of the results (see Figure 7.4) is written into a blank window.

The next two lines check if the production time is larger than the requested maximum runtime. If it is, an error message is returned and the simulation run is ended.

The last two lines were added during modeling of the **ReadValue** module. Since the lines in the block (block code) are to be executed in various places in the model, the block is assigned to the global variable **InitModules**. The starting conditions for the yet-to-be-created modules **OrderGeneration** and **Production** can be recreated by calling up the block with the message **InitModules value.**



```
Mean := AlternativeBarGauge named: 'Mean'.
Productiontime := AlternativeBarGauge named: 'Production time'.
MaxRuntime := AlternativeBarGauge named: 'Max. Runtime'.
ActMaxRuntime := MaxRuntime value.
Result := AlternativeBarGauge named: 'Result'.
Transcript clear.

Productiontime value >= MaxRuntime value ifTrue:
            [self warnAndGiveUp: 'Can not hold the requested maximal runtime.'].

InitModules := [(self moduleNamed: 'Production') setInitialMarking.
              (self moduleNamed: 'OrderGeneration') setInitialMarking]
```

Figure 7.6: Initialization code for production optimization

The **ReadValue** module is shown in Figure 7.9. The PACE-Net function MainEntry is called up by the HillClimbingOptimizer with one argument, which is the only element of the vector **paramVector**. It is the number of workers with which the production, resp. the net procedure, is to be executed.

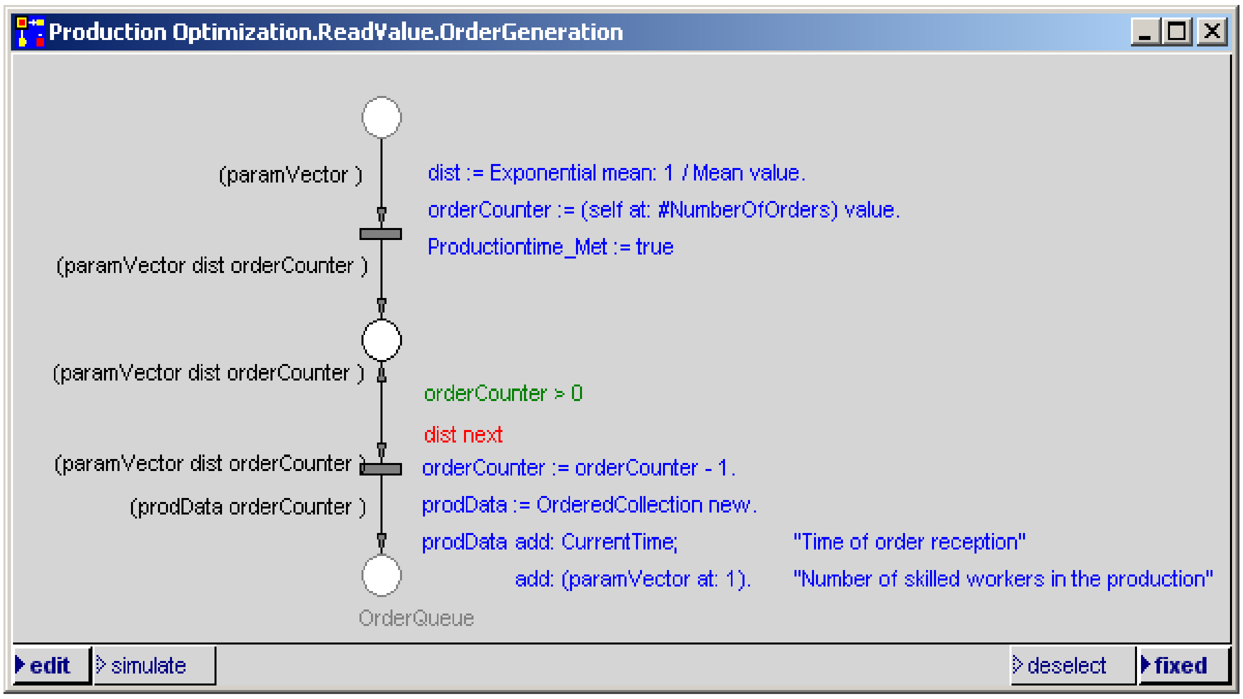
To catch negative values which may occur in the HillClimbingOptimizer, these are immediately steered to the right (Condition code: paramVector first <= 0). The optimizer is returned a very large value which is far above the expected minimum (see below).

For positive values, the condition code paramVector first > 0 in the transition sets the starting conditions for the simulation. Next, the number of workers is output in the transcript window to document the optimization run. Then the simulation time is reset and the modules **OrderGeneration** and **Production** are initialized as described above. Finally, the capacity of the place 'worker', which is positioned in the production module, is set to the number of workers.

Since orders arrive statistically distributed, i.e. they can heap up in between of some boundaries, it is not enough to consider only one order. Instead the set of orders in the place **OrderQueue** which develops during the production has to be investigated. For this reason, the **OrderGeneration** module generates a large number of orders whose arrival times are exponentially distributed. The mean value of the exponential distribution is set using the AlternativeBarGauge **Mean**.

Figure 7.7 shows the **OrderGeneration** module. The exponential distribution is generated in the first transition from the top, setting the time interval between two orders. Then the connector variable **orderCounter** is initialized with the total number of orders which are to be performed. In the model, the net variable **#NumberOfOrders** in the **OrderGeneration** module is present with 10,000. In the global variable **Productiontime_Met** the program notes whether the preset maximum runtime was exceeded at any time during execution of the orders. If so, you can minimize simulation time by halting the simulation run with the just-used number of workers and returning to the optimizer.

The second transition of the module is connected to the place just above it with a double connector. If the transition fires, then as long as the condition **orderCounter > 0** is met, a token is removed from the above place and reset for the next order. The transition switches again after the delay time determined by the next value of the exponential distribution **dist next**. In addition, a token for the next order is sent to the order queue **OrderQueue**. In the course of this, the action code is executed. There the order counter is lowered by 1, and the current time and number of currently-specified workers is stored in a so-called OrderedCollection **prodData**, which runs through the net as an attribute of a token representing the order. An OrderedCollection is a special array with dynamically changeable length. The current value of the connector variable **orderCounter** is the number of orders still to be generated.


Production Optimization.ReadValue.OrderGeneration
(paramVector)
(paramVector dist orderCounter)
(paramVector dist orderCounter)
(paramVector dist orderCounter)
(prodData orderCounter)
OrderQueue
edit
simulate
deselect
fixed

```
dist := Exponential mean: 1 / Mean value.
orderCounter := (self at: #NumberOfOrders) value.
Productiontime_Met := true
```

```
orderCounter > 0
dist next
orderCounter := orderCounter - 1.
prodData := OrderedCollection new.
prodData add: CurrentTime;           "Time of order reception"
         add: (paramVector at: 1).   "Number of skilled workers in the production"
```

Figure 7.7: The module OrderGeneration

Figure 7.8 shows the **Production** module, in which a waiting queue is modeled as in the car wash described earlier. In contrast to the car wash, here several objects may be processed simultaneously.

In connection with the **Production** module, the **ReadValue** module checks if the maximum production time has been exceeded. The difference **CurrentTime - prodData first** delivers the runtime of the order. If it is larger than the maximum runtime allowed specified in the AlternativeBarGauge, then the number of workers has been set too low and the global variable **Productiontime_Met** is set to **false**.

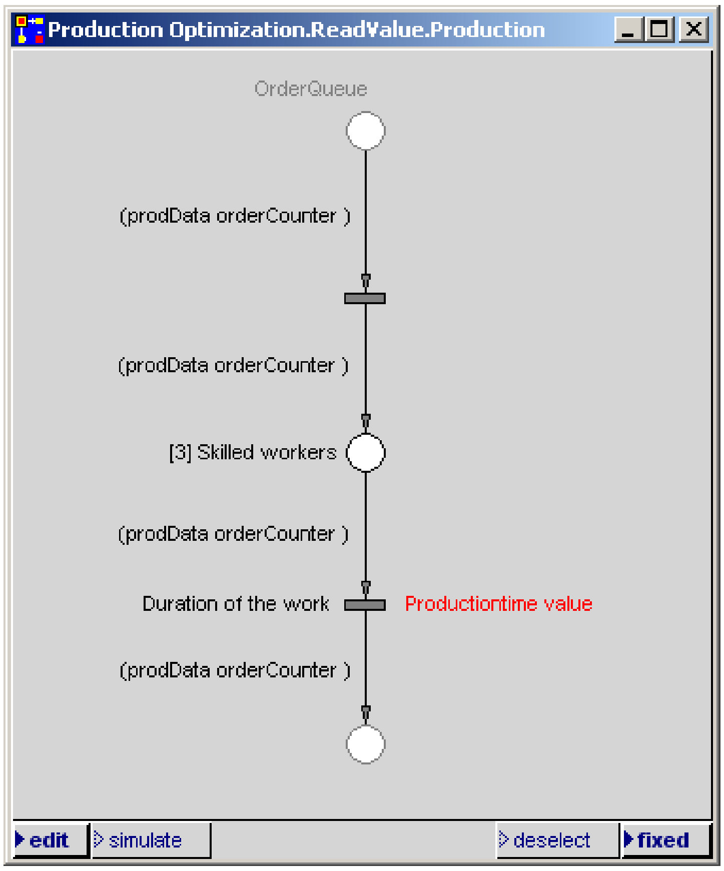
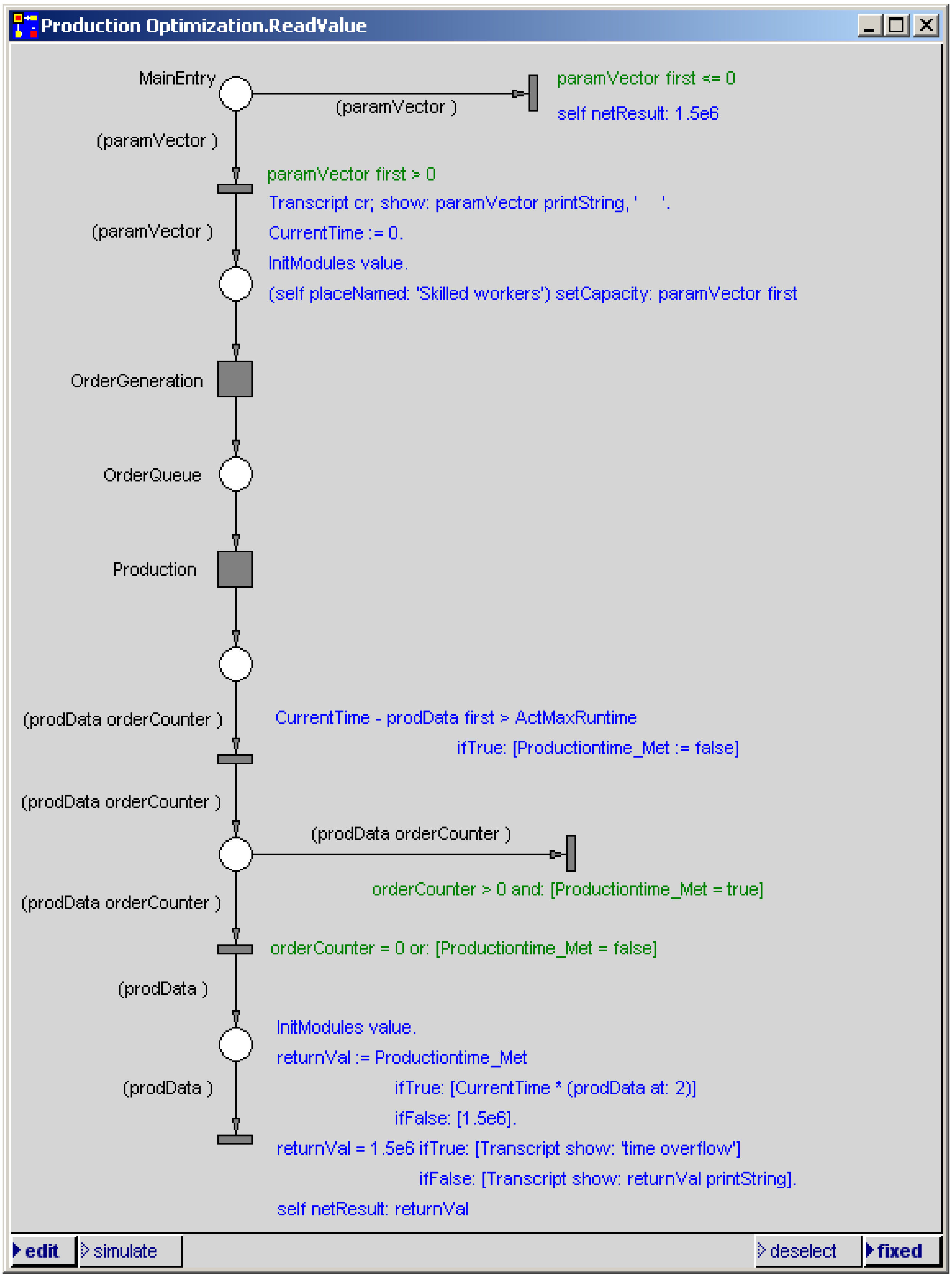


Figure 7.8: The module Production

In the following step, a token runs to the right and is deleted if more orders remain to be done and the production time parameters have been met. If not, the token runs below. Here the generation of further tokens (orders) is first stopped when the Block **InitModules** is called up, and then the function value is calculated.

The total time for the completion of the 10,000 assignments is CurrentTime. Multiply this time by the number of skilled workers specified in (prodData at: 2) to get a value proportional to the cost of the work. This may be used as the function value for cost optimization. If the production time cannot be met, a large replacement value 1.5e6 is returned as the function value.

The last step in the simulation run is to complete the result specified in the beginning of the module. Depending on whether the production time specifications were met or not, the Transcript window will show the calculated function value or the string **time overflow**.



paramVector first <= 0

self netResult: 1.5e6

paramVector first > 0

```
Transcript cr; show: paramVector printString, '    '.
CurrentTime := 0.
InitModules value.
(self placeNamed: 'Skilled workers') setCapacity: paramVector first
```

```
CurrentTime - prodData first > ActMaxRuntime
                    ifTrue: [Productiontime_Met := false]
```

orderCounter > 0 and: [Productiontime_Met = true]

orderCounter = 0 or: [Productiontime_Met = false]

```
InitModules value.
returnVal := Productiontime_Met
            ifTrue: [CurrentTime * (prodData at: 2)]
            ifFalse: [1.5e6].
returnVal = 1.5e6 ifTrue: [Transcript show: 'time overflow']
               ifFalse: [Transcript show: returnVal printString].
self netResult: returnVal
```

Figure 7.9: The ReadValue module for production optimization

7.3 Experimenting with the Model

Planning is often based on consideration of mean values. This is acceptable only if runtime and order queues are not limited. If, for example, you set both the production time and mean value of orders per hour to 1 and the maximum runtime to 10, the current model requires only 2 skilled workers to carry out the order processing. This value is also produced when considering mean values.

This changes dramatically when the maximum allowable runtime is limited. If the maximum allowable runtime is close to a value of 1, then 7 skilled workers are required to execute the order processing. The requirement for faster delivery of a product, i.e., faster processing of an order, would lead to considerably higher costs as expected.

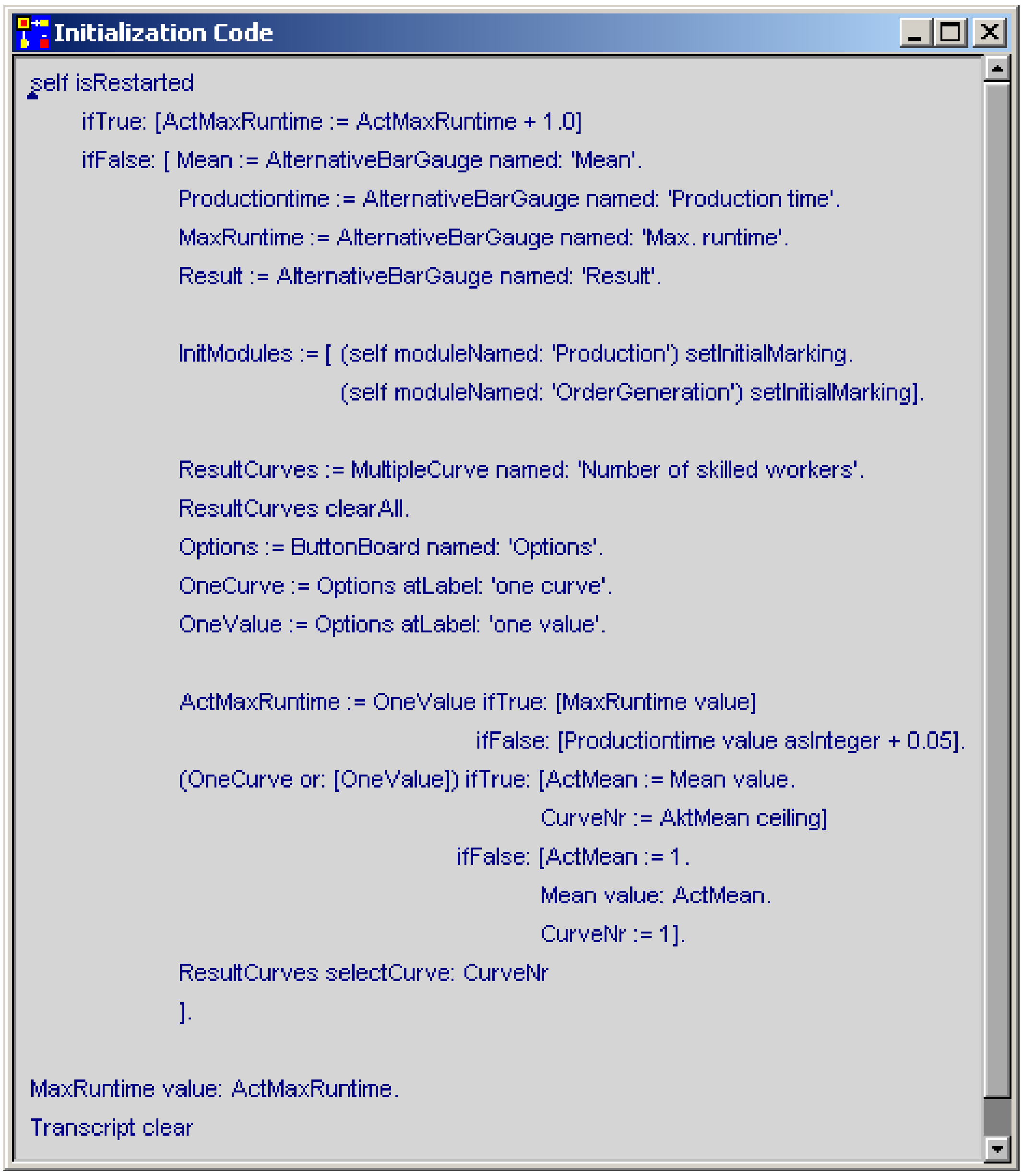
An overview of the situation is best achieved with charts or suitable graphics. To achieve an overview of the relationship between the number of skilled workers and the maximum runtime for order distributions with different mean values, the model in Section 7.2 must be enlarged slightly to carry out the experiments. Here the options described in Section 6.3, which allow the whole model to be executed repeatedly, can be used.

Figure 7.14 shows the expanded modeling interface. It adds the so-called ButtonBoard **Options** and a multiple-curve window to the previous interface (Figure 7.4). With **Options** you can specify,

- Button: **one value**
  the appropriate number of skilled workers to be calculated for a combination of production time, maximum runtime and orders per hour.
- Button. **one curve**
  a curve is to be calculated and drawn which represents the number of skilled workers for a fixed production time and a fixed mean number of orders per hour. The maximum production time traverses the values from the specified production time up to a value of 10 hours in one-hour increments.
- Button: **all curves**
  the curve specified in the preceding point is to be drawn for all mean values of orders in the range from 1 though 10.

Figure 7.10 shows the expanded initialization code. The majority of the earlier code (see Figure 7.6) is included in the false-branch of the conditional message which begins with **self isRestarted**. Added to that is a global variable **ResultCurves** which stores the multiple curve 'Number of skilled workers'. After this assignment, all possibly existing curves in the curve window are deleted with **clearAll**. Then the **Options** ButtonBoard is assigned to a further global variable **Options**. The status of the buttons '**one curve**' and '**one value**' of the ButtonBoard **Options** are stored in additional global variables **OneCurve** and **OneValue**. If the button is selected, the button delivers the value **true**; if not, the value is **false.**

Then the current runtime **ActMaxRuntime** is assigned. If the 'one value' button is set, the value of the **Max. Runtime** bar is assigned. If not, a value is used which is mini-



```
self isRestarted
      ifTrue: [ActMaxRuntime := ActMaxRuntime + 1.0]
      ifFalse: [ Mean := AlternativeBarGauge named: 'Mean'.
                 Productiontime := AlternativeBarGauge named: 'Production time'.
                 MaxRuntime := AlternativeBarGauge named: 'Max. runtime'.
                 Result := AlternativeBarGauge named: 'Result'.

                 InitModules := [ (self moduleNamed: 'Production') setInitialMarking.
                                  (self moduleNamed: 'OrderGeneration') setInitialMarking].

                 ResultCurves := MultipleCurve named: 'Number of skilled workers'.
                 ResultCurves clearAll.
                 Options := ButtonBoard named: 'Options'.
                 OneCurve := Options atLabel: 'one curve'.
                 OneValue := Options atLabel: 'one value'.

                 ActMaxRuntime := OneValue ifTrue: [MaxRuntime value]
                                           ifFalse: [Productiontime value asInteger + 0.05].
                 (OneCurve or: [OneValue]) ifTrue: [ActMean := Mean value.
                                                    CurveNr := AktMean ceiling]
                                           ifFalse: [ActMean := 1.
                                                    Mean value: ActMean.
                                                    CurveNr := 1].
                 ResultCurves selectCurve: CurveNr
                 ].

MaxRuntime value: ActMaxRuntime.
Transcript clear
```

Figure. 7.10: Expanded initialization code

mally larger than the value of the bar **Productiontime**. The production time is the lower limit for the runtime.

In the remaining lines of the false-block, the mean value of the exponential distribution **ActMean** and the assigned curve color are set, which is implicitly defined by the curve number of the curve **CurveNr**. If only one value is to be calculated or one curve to draw, the **Mean** bar provides this mean value and the curve number is defined as the next largest integer (ActMean ceiling). Alternately, all curves are to be drawn, and both ActMean and CurveNr receive a starting value of 1.

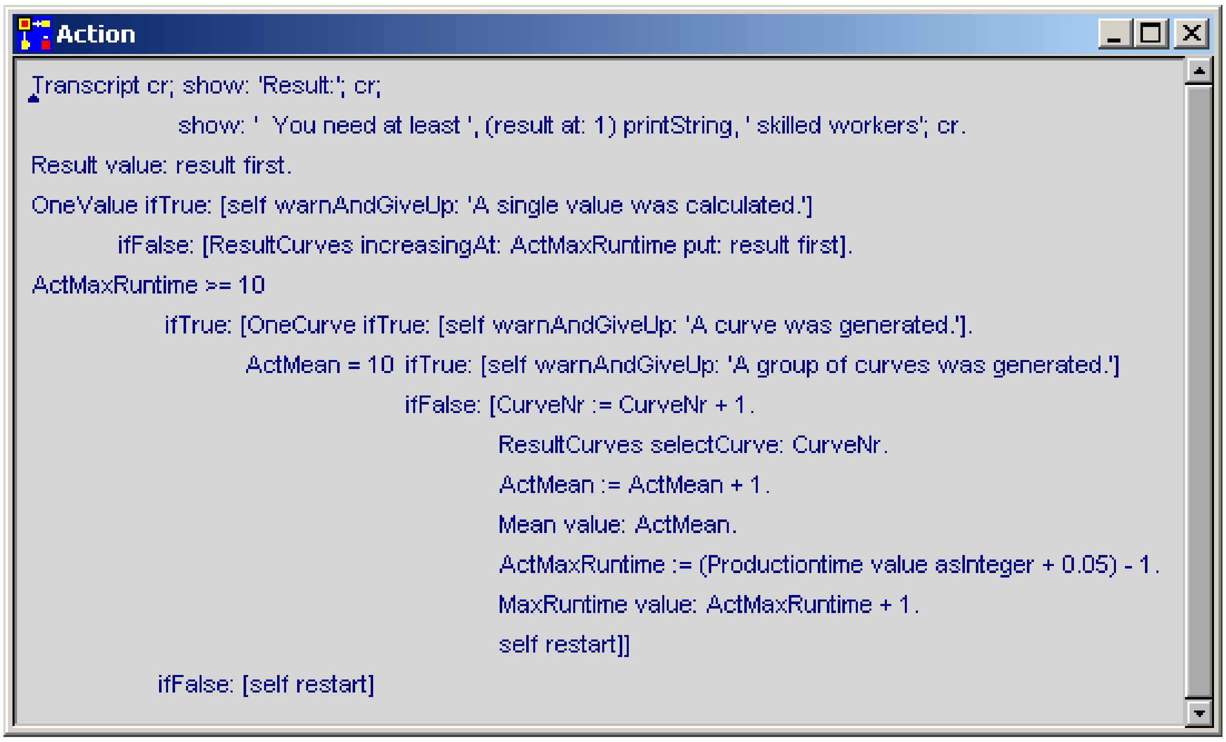
The last line of the block sets the specified curve for further outputs to ResultCurves.

In the true-block, the next value for the abscissa in the next run of the model is calculated by increasing the current maximum runtime **ActMaxRuntime** by 1. After the conditional message, the **Max.Runtime** bar is set to a new value and the contents of the transcript window is deleted.

In addition to changing the Initialization code, the action code of the transition must be altered, which is executed when a token which the optimizer has inserted in the place **ResultLabel** (Figure 7.11) is processed.

The first three lines were already introduced in Section 7.2, Figure 7.1.

In the conditional message which evaluates **OneValue**, the simulation run is ended if true; if not true, the next value to **ResultCurves** is output.



```
Transcript cr; show: 'Result:'; cr;
           show: '  You need at least ', (result at: 1) printString, ' skilled workers'; cr.
Result value: result first.
OneValue ifTrue: [self warnAndGiveUp: 'A single value was calculated.']
      ifFalse: [ResultCurves increasingAt: ActMaxRuntime put: result first].
ActMaxRuntime >= 10
         ifTrue: [OneCurve ifTrue: [self warnAndGiveUp: 'A curve was generated.'].
               ActMean = 10  ifTrue: [self warnAndGiveUp: 'A group of curves was generated.']
                          ifFalse: [CurveNr := CurveNr + 1.
                                ResultCurves selectCurve: CurveNr.
                                ActMean := ActMean + 1.
                                Mean value: ActMean.
                                ActMaxRuntime := (Productiontime value asInteger + 0.05) - 1.
                                MaxRuntime value: ActMaxRuntime + 1.
                                self restart]]
         ifFalse: [self restart]
```

Figure 7.11: Expanded Transitions code

If **ActMaxRuntime** >= 10, a curve is finished, otherwise, using the command **restart**, the next run of the model is started and the next value of the current curve is calculated. In the true-branch, the simulation is either ended (if Actmean = 10) or switched to the next curve (ActMean := ActMean +1).

We still need to describe how the ButtonBoard **Options** in Figure 7.12 is created. From the **Views**-menu on the PACE's main toolbar, select **button board**. A window with two input lines will open, with which you can define the form of the ButtonBoard. The preset values which appear when the window opens are precisely those which are needed here (3 buttons in one row). To create the ButtonBoard (Figure 7.12), press Return.

The window is given the name **Options** (**mi.MK**, **relabel as…**). To replace the default names **label** shown in Figure 7.12 with the names **a value, a curve, all curves**, (Figure 7.14), position the cursor on a button and call up the button's menu (**ri.MK**). Select the menu point label and enter the appropriate name in the input window which opens, then press the **Return** key.

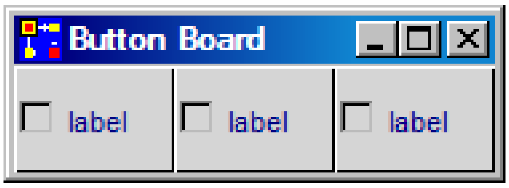
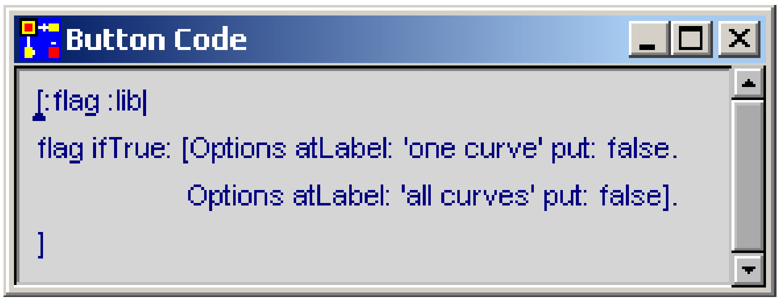


Figure 7.12: Newly created ButtonBoard

So that only one button is pressed at a time, all other buttons must be released when one is pressed. To that end, each button is assigned a code which is executed when the button is pressed. For the left-hand button, for example, the procedure is as follows:

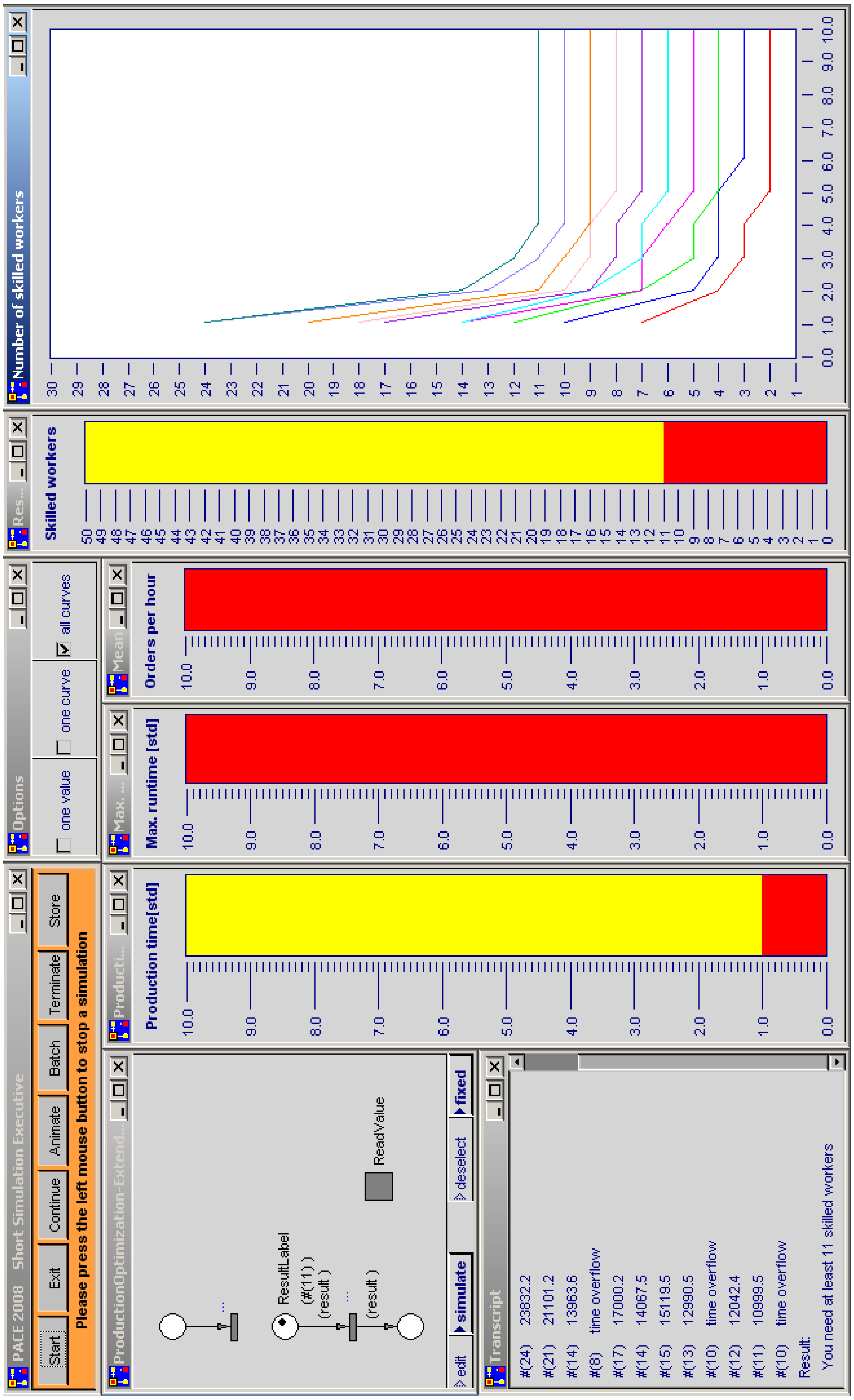
Set the cursor on the button and call up the button menu again (**ri.MK**). This time, select menu option **block**. A block will open in which the delete code may be entered. After this input, the block will look as follows (Figure. 7.13):



```
[:flag :lib|
flag ifTrue: [Options atLabel: 'one curve' put: false.
             Options atLabel: 'all curves' put: false].
]
```

Figure 7.13: Code to delete a Button

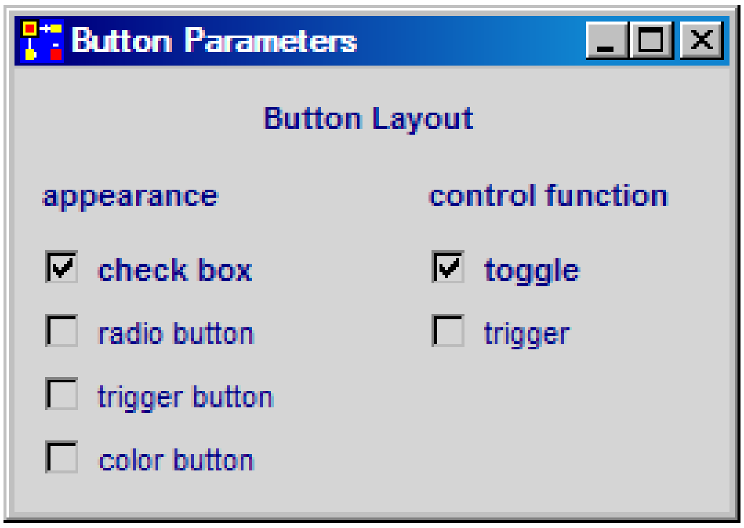
flag contains the current button status. If **flag = true**, then the button is pressed. In this case, the other two buttons are assigned the value **false**, i.e., the selection of the other buttons is reset.



```
#(24)   23832.2
#(21)   21101.2
#(14)   13963.6
#(8)    time overflow
#(17)   17000.2
#(14)   14067.5
#(15)   15119.5
#(13)   12990.5
#(10)   time overflow
#(12)   12042.4
#(11)   10999.5
#(10)   time overflow
Result:
You need at least 11 skilled workers
```

Figure 7.14: Expanded Model Overview

The control function of the button must be altered to avoid releasing the button through repeated pressing of an already-pressed button (thus leaving no button pressed at all). For each button, call up the button menu and select **parameters.** That will call up the window shown in Figure 7.15. Change the control function by selecting **trigger.**


Button Parameters

Button Layout

appearance

- [✓] check box
- [ ] radio button
- [ ] trigger button
- [ ] color button

control function

- [✓] toggle
- [ ] trigger

Figure 7.15: Default-Layout for a Button

When you have appropriately specified the other two buttons, the ButtonBoard is complete.

Figure 7.14 shows all curves for an order runtime of one hour. One can see that the required number of skilled workers rises sharply for short response times (maximum processing times – run time) of less than an hour, and for response times of several hour converges only slowly toward the result calculated with the mean value of the exponential distribution.

8. What's left to do?

As the example shows, modeling in PACE is a combination of net description and program code which controls processes in the net. Inscriptions and extra-codes usually require only a few Smalltalk constructs, such as assignment, conditional messages, etc., and several special messages such as show:, value, value:, restart, etc.

While creating nets is usually learned easily, creating inscriptions in Smalltalk often causes problems, particularly if the user is not familiar with other programming languages already. For that reason, we've included a Smalltalk primer which describes the requisite language constructions and provides multiple examples. We strongly suggest you to review Chapters 3 through 5 of the Smalltalk Primer to make model creation easy and even fun.