Yasper

:

Yet Another Smart Process EditoR

User Guide

What is Yasper?

Yasper is a tool to *specify* and *execute* models of discrete-step processes.

A Yasper process model shows the steps of a process and the order dependencies between them in one or more diagrams. The diagram technique supports alternative and parallel paths, repetitions of steps, and contention for resources between steps. The diagram isn't merely a sketch, but provides an exact formal specification of the order dependencies; this allows *simulation*. A simulation is an execution of consecutive process steps that satisfies the specification. Yasper supports two modes of simulation: *manual* mode, in which the user selects the next step to execute in the diagram, and *automatic* mode, which randomizes the choice of steps and produces an aggregated report with relevant statistics.

Yasper was developed in collaboration between TU Eindhoven and Deloitte.
For details of its status, availability and use, see the Yasper website:

http://www.yasper.org/

Yasper was written to provide modeling convenience, rather than new and unique new modeling features: the features it supports are well known, well analyzed, and support exists for them in other tools. What is more, discrete process modeling tools in general tend to support techniques such as state machines, flowcharts, and activity diagrams, which are very closely related to Yasper models.

How to read this User Guide

This user guide assumes you have decided to give Yasper a try, and attempts to provide you with everything you need know in order to use it. It does not assume any prior knowledge of the techniques it uses; however, where appropriate, some comparisons with similar tools and techniques will be made for the benefit of the reader who is already familiar with them. Depending on your background knowledge you may wish to skip certain parts of this guide.

*Chapter 1 d*escribes Yasper models. It details Yasper's modeling constructs, how they fit together, how they appear to the user, and what they mean in terms of process execution.

Chapter 2 describes the Yasper editor. It details the user interface functionality related to the specification of models. It also describes the means available to import or export Yasper models from/to other applications.

Chapter 3 describes manual execution mode. It details the user interface functionality specific to manual execution and describes the relationship between this mode and edit mode.

Chapter 4 describes automatic simulation mode. It details the user interface functionality related to setting simulation parameters and executing a simulation run, and defines the meaning of the statistics reported.

Table of contents

1 Process models in Yasper

This chapter describes the process modeling features supported by Yasper.
If you just want a description of how to operate the Yasper tool, skip to chapter 2 (the editor), 3 (manual process execution) or 4 (automated process execution).

Yasper's process models are *Petri nets*. Like most Petri net tools, it extends basic nets, defined by C.A. Petri in 1962, with additional constructs.

Section 1.1 describes basic Petri nets: networks of *transitions* (process steps or events) connected through *places* (conditions that must be fulfilled before or after a process step). Examples of typical modeling patterns are given.

Section 1.2 adds *choice*, a construct known from flowcharts and other techniques.

Section 1.3 adds *roles*, an alternative to using places for modeling resources.

Section 1.4 adds hierarchical modeling with the subnet construct.

Section 1.5 adds special types of connections: combined arcs, inhibitor arcs and reset arcs.

Section 1.6 describes arc weights.

Section 1.6.2 and 1.6.3 add processing time and processing cost.

Section 1.6.4 adds *token case*. With this feature, process execution can distinguish between different cases being processed at the same time. This makes it easier to model and simulate workflow-like processes.

Sections 1.7 and 1.8 describe Yasper's rudimentary support for *token color* and *store*s.

1.1 Basic nets

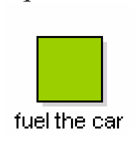
This section defines Yasper's support for basic Petri nets. For a rigid introduction of Petri nets and their properties, the reader is referred to the many introductory books and articles that have appeared on Petri nets.

1.1.1 Basic net elements

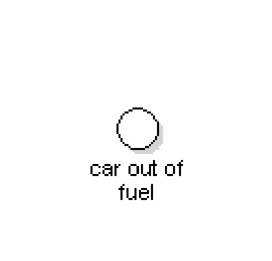
Models express the flow structure of processes. The simplest kind of model only uses three constructs and is called a *basic net*.

1.1.1.1 Transitions and places

The most important modeling unit represents a process step. It appears in the diagram as a green square.



Process flow structure means that not every step can occur at every moment: they are subject to certain conditions. In Yasper, these conditions are made explicit.

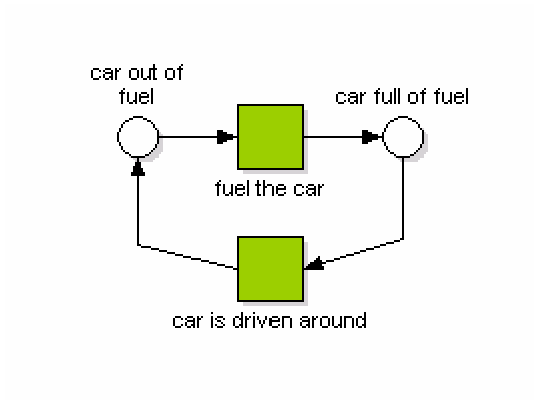


The squares represent actions, processes or events; generally speaking, *transitions* between states of affairs in the world.

The circles represent conditions, and together, the possible states of affairs as far as they are relevant to the process flow of the process being described. They are known as *places.*

1.1.1.2 Arcs and nets

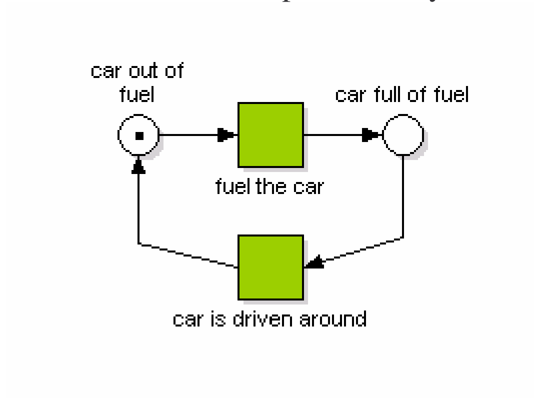
Transitions and places are connected to form a *net*; for example,



The arrows are known as *arcs*.

1.1.1.3 Tokens and markings

A state of affairs is represented by marking the conditions that hold:

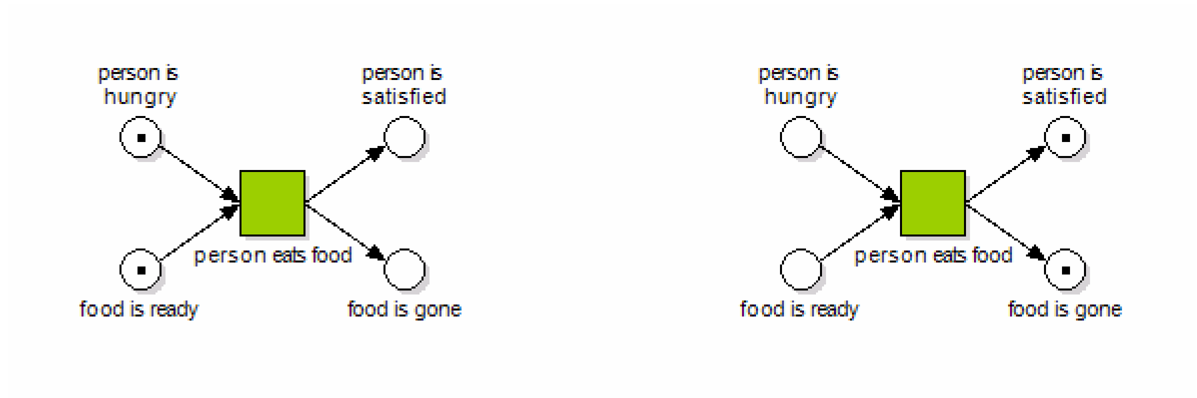


The dots are known as *tokens*; a distribution of tokens over places is a *marking*.

1.1.2 Process execution in basic nets

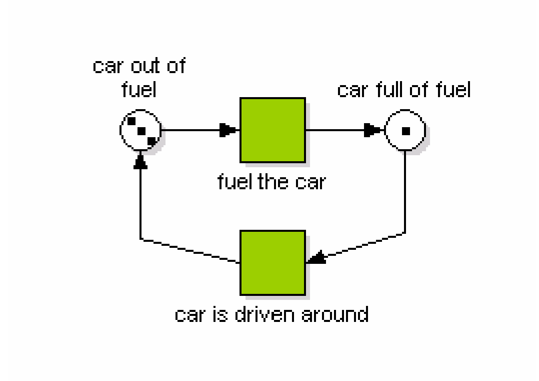
Process execution is defined by a single rule: a transition can occur whenever all places that have arcs to it contain a token; its occurrence will remove a token from each of them, and put a token in each of the places to which it has an arc.

For example, in the above diagram, *fuel the car* can only occur when *car out of fuel* is marked with a token, and when it occurs, the token moves into *car full of fuel*.
As soon as the token arrives there, *car is driven around* becomes possible; when it occurs, the token moves back into *car out of fuel*. So we see the model is not very realistic: a car cannot be driven without running out of fuel, and it cannot be fueled before it has run out. But the point of the example is that process execution is completely defined by this one transition execution rule.



In this model, *person eats food* can only happen when a person is hungry and food is available; and when it does, the person is no longer hungry, but satisfied, while the food is no longer ready, but gone.

It is legal for multiple tokens to be in the same place:



In this example, the tokens represent cars: three still out of fuel, one ready to drive. Generally speaking, in models using this technique, tokens usually describe objects, while places describe states these objects can be in. A whole marking describes the overall state of the system.

Similarly, any number of persons and food portions can be introduced into the other example.

Note that execution is nondeterministic: it is precisely determined what happens when a transition executes, but when multiple such executions are possible at the same time, it is not determined which one will happen. More on this in section 1.6.4.4.

1.1.3 Modeling with basic nets

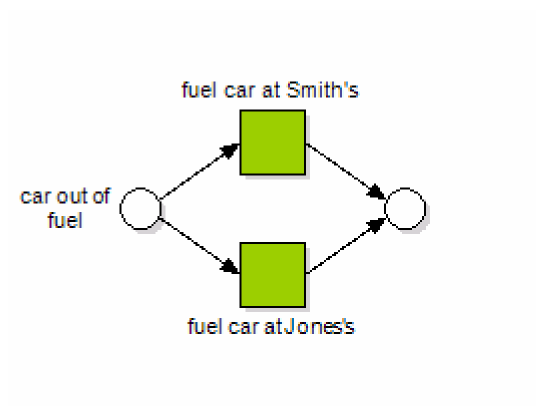
This section presents some examples to illustrate typical modeling patterns.

The intention is to provide some basic notions and examples that later sections will build on to introduce additional Yasper features.

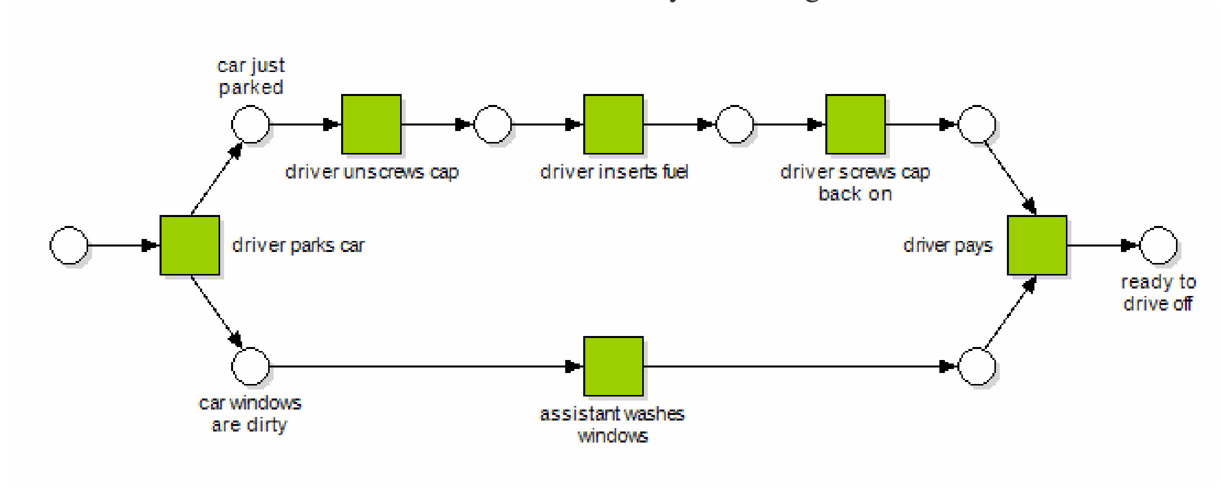
Full tutorials on the use of Petri nets can be found elsewhere.

1.1.3.1 Three useful types of branching in basic nets

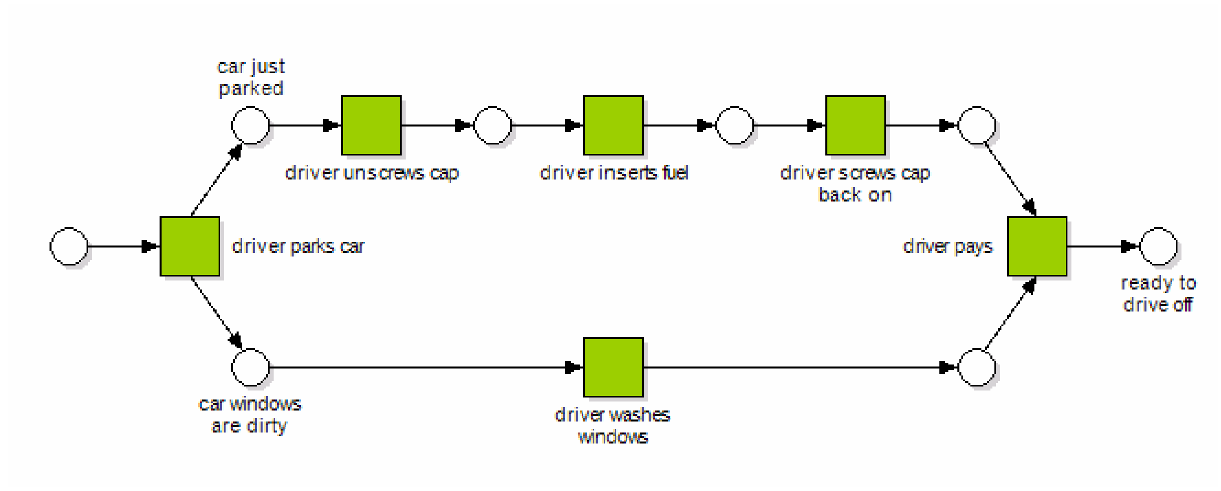
Alternative paths can be described by branching on places:



Parallel execution, on the other hand, is described by branching on transitions:

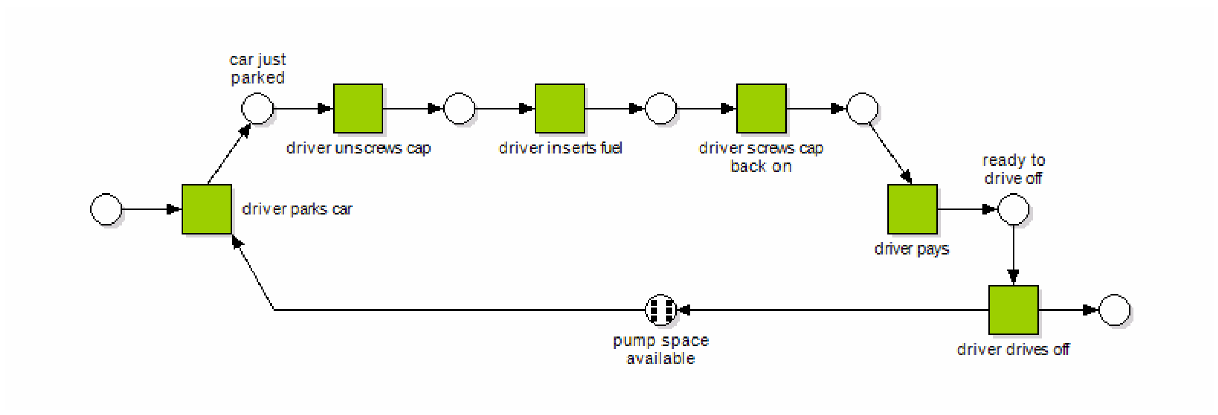


This introduces *nondeterminism*: the order in which the upper and lower branch execute is undefined. For example, the assistant may finish before the driver has unscrewed the cap, or may start washing while the driver is inserting fuel and only finish after the driver is ready, or any other combination. In fact, any situation in which the order of steps is undefined must be modeled with parallel branches:



In this example, there is no implication that things are done in parallel; it just models the situation in which the windows can be washed at any time while the other three steps must be performed in the order given.

Branching on transitions can also be used to describe resource contention:



The execution rule given above determines that whenever six cars have parked and remain somewhere in the process, *driver parks car* will not be possible until one of them drives off, thereby releasing the space it occupied. All incoming cars would be kept waiting, according to this model.

1.1.3.2 Well-handled nets and proper completion

As the examples show, these few basic constructs and the one execution rule give us a very simple, yet very powerful technique to describe process flow logic. Nice aspects are that the flow logic is fully defined by the execution rule, and that it is fully displayed in the diagram.

The three examples have some things in common:

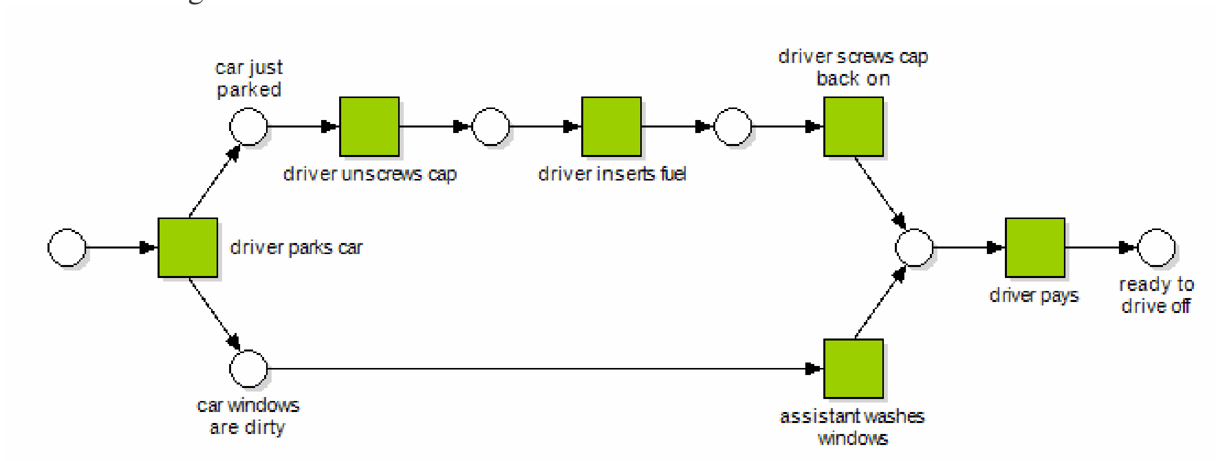
- all branch points form pairs of splits with matching joins of the same type; if a net has this property, it is called *well-handled;*[1]
- they describe *workflows,* processes in which individual cases (represented by tokens) flow from a fixed starting point to a fixed end point; if such a net can handle such business cases indefinitely and concurrently without ever locking up or amassing tokens anywhere, we say it *completes properly.*[2]

An exact definition of proper completion is given in section 1.6.4.5.

[1] This is in accordance with Petri net literature, e.g. *Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques* (Van der Aalst, 1998). The term well-structured may seem more natural, but is already in use for a different notion.
[2] Proper completion generalizes the *soundness* property defined in *Workflow management: Models, Methods, and Systems*, (Wil van der Aalst and Kees van Hee, 2002), which requires the net to be empty initially and eventually.

In the practice of modeling, proper completion is an important correctness criterion.
A net with a clear begin and end point that lacks this property may well point out a real problem with the process in question, but more often, is an incorrect model of the process. It turns out to be very easy for human modelers – including the authors of Yasper - to make mistakes even in simple models, and checking for proper completion turns out to reveal most such errors.

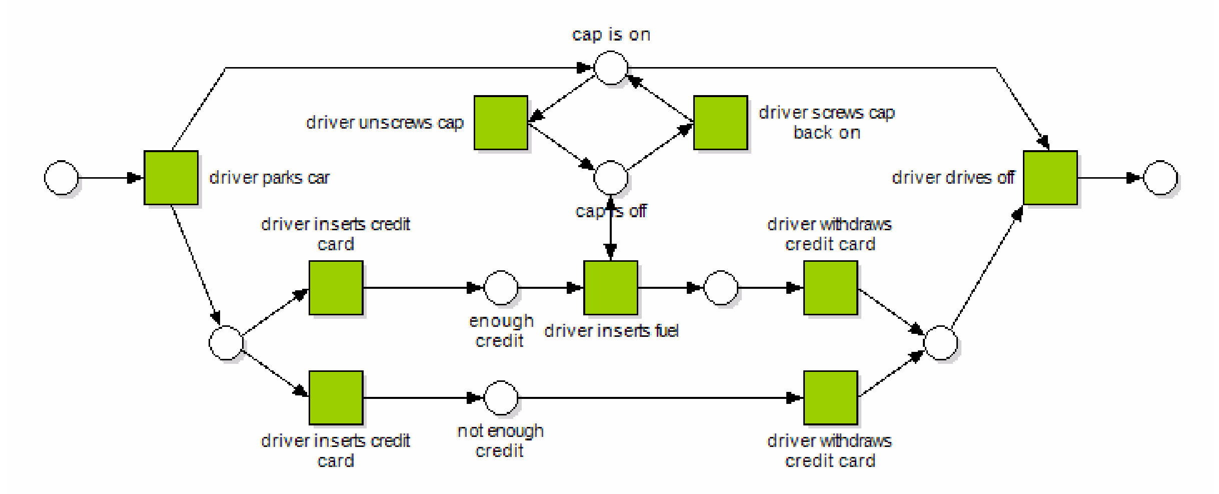
Well-handled nets always complete properly, so it is a good design principle to stick to neatly bracketed split-join pairs whenever possible. Doing so makes problems easy to spot, as can be seen in this beginners' error:



Every driver parking a car will pay twice and end up with two cars!
The cause is obvious: a transition split is mismatched with a place join.

However, in well-handled nets, the only form of parallelism is completely hierarchical: a process can start multiple concurrent processes and wait for all of them to finish, but communication or synchronization between different subprocesses is impossible. In practice, a large class of processes can only be modeled with nets that properly complete, but are not well-handled.

This situation can arise as soon as the constraints on the order of process steps are anything other than a choice between fixed paths, even when they are quite simple:


cap is on
driver unscrews cap
driver screws cap back on
driver parks car
driver drives off
cap is off
driver inserts credit card
driver withdraws credit card
enough credit
driver inserts fuel
driver inserts credit card
not enough credit
driver withdraws credit card

At this level of complexity, verifying proper completion already becomes a problem to the human eye. Proper completion can in fact be verified automatically in many cases. However, Yasper doesn't yet offer such verification, because it has a much more insightful way to check for improper completion: manual or automatic simulation. The user sees the tokens flowing through the net and spots deadlocks or bottlenecks where they arise. Simulation turns out to be an extremely convenient tool in designing correct non-well-handled nets, and appears to catch nearly all of the modeling errors that arise in practice.

1.1.4 Transitions and places must alternate

In Petri nets, arcs always run between transitions and places: it is illegal to connect two transitions or two places directly. Such arcs could be introduced as shortcuts, on which the intermediate place or transition has been omitted.

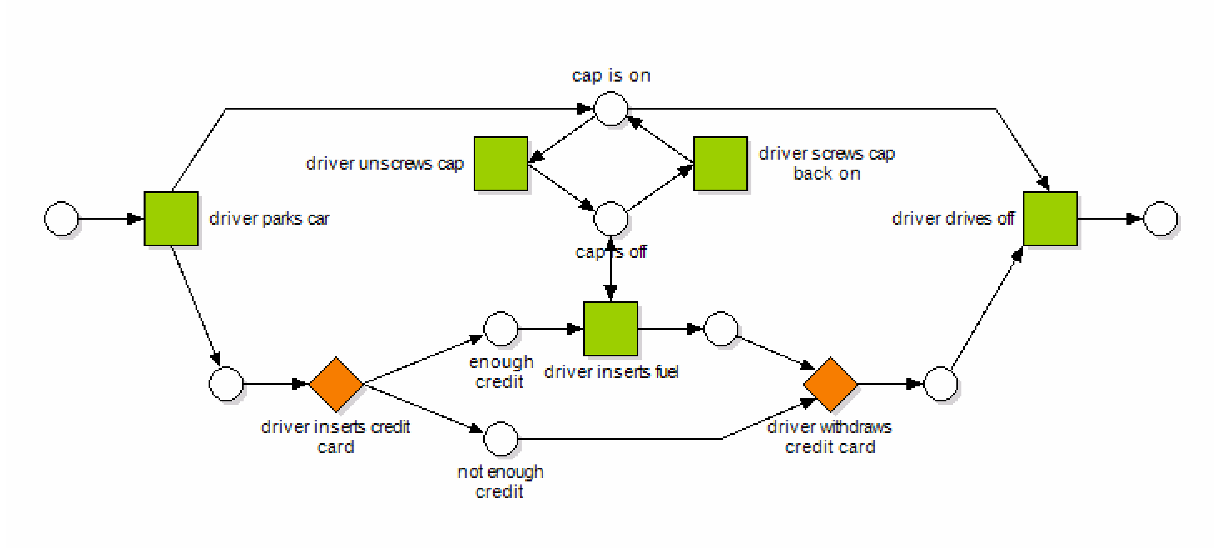
Although we have experimented with this feature, the present Yasper release does not support such shortcuts. It is tidier to always make all transitions and places explicit, since branching can happen on both.

In practice we often see nets in which all branching happens on places: such models express alternative paths, but no parallelism or resource contention. A net with this property is called a *state machine*. For them, it makes sense to always omit the transitions and connect the places directly; this is in fact how state machines are usually drawn. Historically, state machines predate Petri nets, and they are still used as a basis for popular specification techniques, such as UML statecharts.

Many process modeling techniques exist today, often based on state machines (e.g. UML statecharts) or flowcharts (see the next section). Some of these techniques can easily be described with Petri nets: e.g., UML activity diagrams and the XLANG process models of Microsoft BizTalk are effectively well-handled Petri nets without initial tokens; as we have seen, this implies that they cannot directly express interprocess communication or resource contention.

1.2 Choice elements

There are two places in which the preceding example looks quite awkward: two of the process steps appear twice. The reason is that the steps in question are subject to varying conditions: for *driver inserts credit card,* its results, and for *driver withdraws credit card,* its prerequisites. The process execution rule does not allow either, and duplication is the only way to resolve this. Clearly it is more natural to just write:

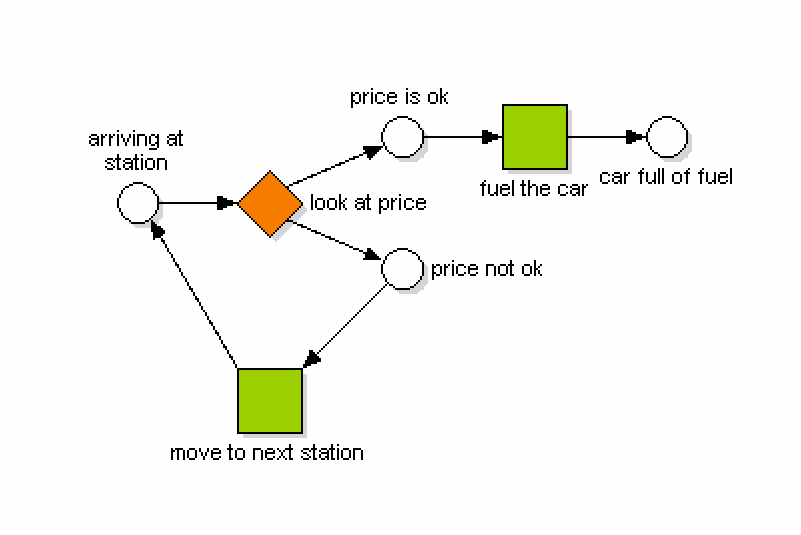


The new symbol is called a *choice* or *XOR* element, a construct used in *flowcharts*, a technique developed by Goldstine and Von Neumann in 1946-1947,[3] and in modern derivatives such as UML activity diagrams and BPMN.

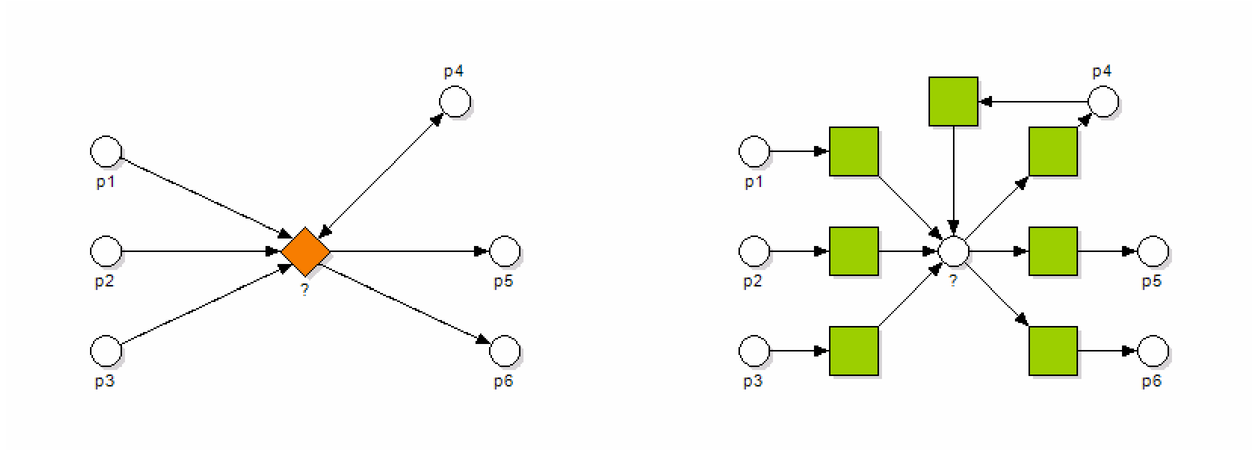
The process execution rule is extended as follows: a choice consumes a token from one of its input places, if any, and produces a token in one of its output places, if any.

Again, it is best practice to use XORs in pairs, so as to keep models well-handled, but this is not always possible. For instance, they often appear in loops:

[3] See pp. 266-267 in Herman.H. Goldstine, The Computer from Pascal to Von Neumann, Princeton University Press, 1972, ISBN 0-691-08104-2



An XOR can be defined as a place surrounded by non-branching transitions:

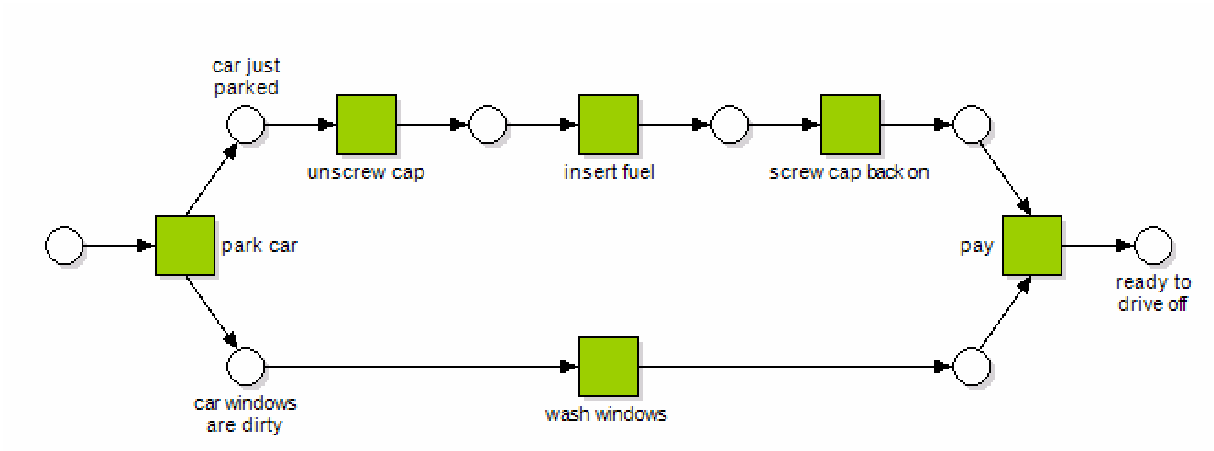


At present, Yasper doesn't quite treat XORs in that way, since it allows roles (cf. section 1.3) and output weights (cf. section 1.6.1) to be associated with them, and treats their execution atomically in manual simulation (cf. section 3.4).

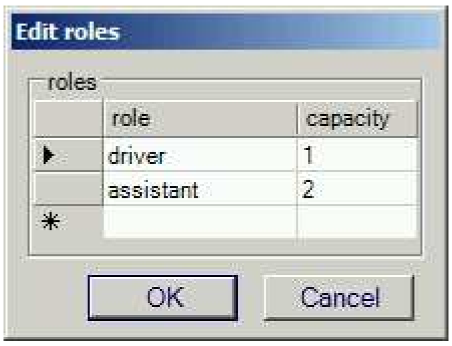
1.3 Roles

In business processes or production processes, efficient resource allocation is often a major driving force behind process design. Yasper offers a special facility for this purpose: the execution of a transition can be assigned to one or more *roles*.

Consider, for instance, the parallel execution example in section 1.1.3.
We may wish to express that in general there are two types of agents available to perform the tasks involved: drivers and assistants. In Yasper we can define *driver* and *assistant* as roles, and tag individual transitions with the roles that can execute them. The general model would look like this:

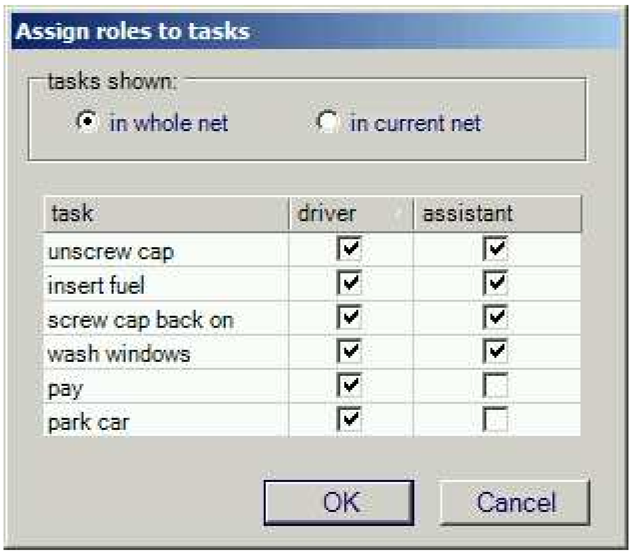


Roles do not appear in the diagram, but are defined separately (see section 2.11):


Edit roles

roles

	role	capacity
▶	driver	1
	assistant	2
*		


Assign roles to tasks

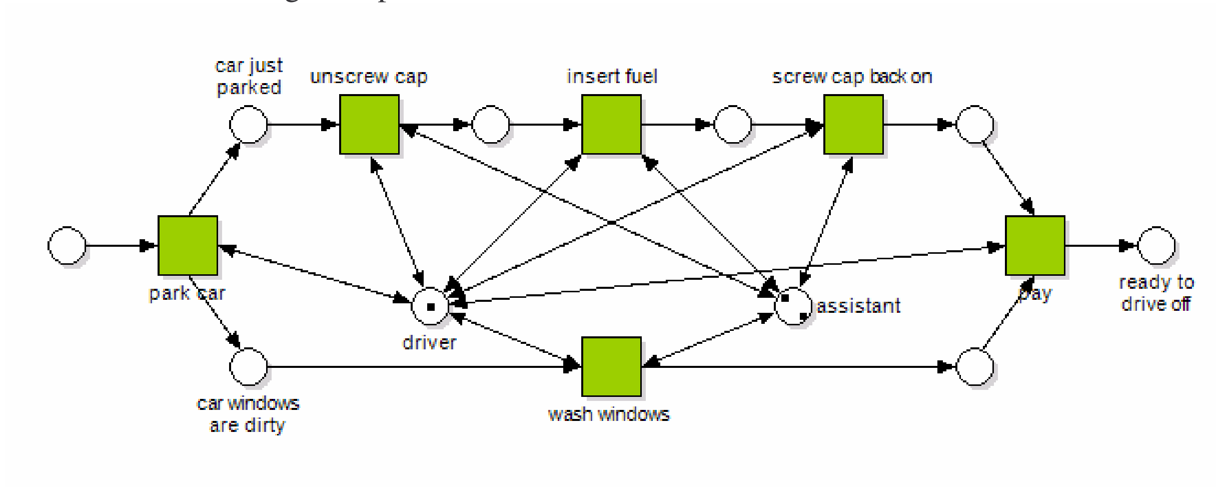
tasks shown: [✓ in whole net] in current net

task	driver	assistant
unscrew cap	✓	✓
insert fuel	✓	✓
screw cap back on	✓	✓
wash windows	✓	✓
pay	✓	
park car	✓	

Note that roles have capacities: there is one driver and there are two assistants.

Process execution is influenced by roles in the expected way: a transition with assigned role(s) can only execute when an instance of a role is available to perform it, and while it executes, will keep that instance from doing something else.
For example, the windows can only be washed if a driver or an assistant is available,
and no more than three cars can have their windows washed at the same time.

Although resources can also be modeled with places, a role is not a place.
Consider the following example:



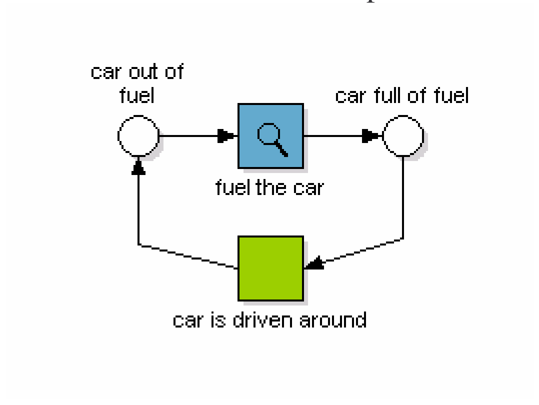
The capacities and assignments to transitions are the same, but execution is different: in this model, *wash windows* occupies both a driver and an assistant, while in the previous model, one of either suffices.[4]

Whether to model resources with places or with roles depends on the purpose of the Yasper model. Manual and automatic simulation have special support for roles; for example, role utilization is aggregated in the automatic simulation report. See sections 3.6 and 4.5 for details.

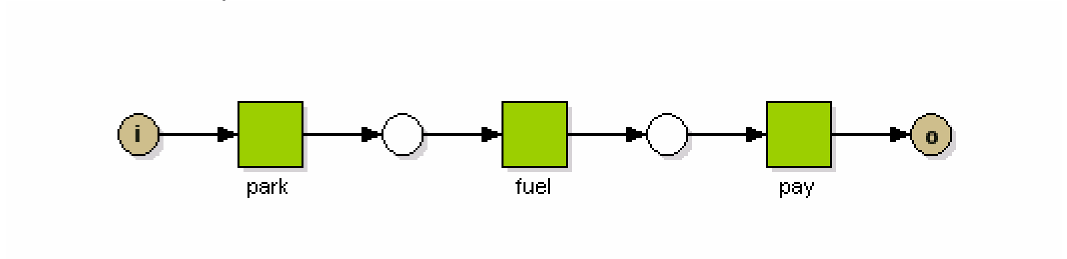
[4] Note that neither example is realistic: in practice, each car comes with its own driver, so drivers should not be modeled as resources at all.

1.4 Hierarchical models

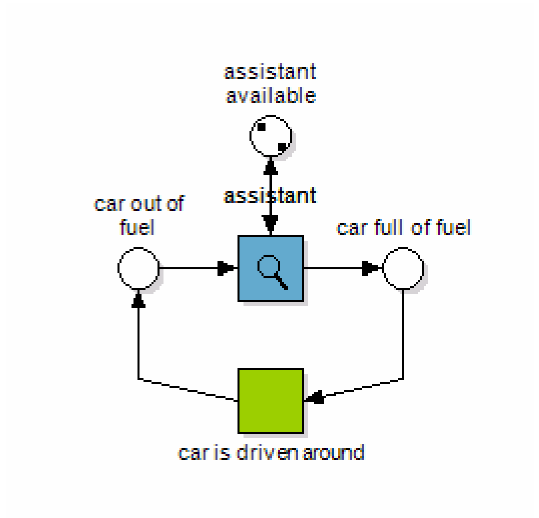
Models can grow big rather quickly. It is possible to spread the contents of a net over multiple pages by means of the *subnet* construct. In the surrounding net, a subnet is like a transition, in that it must be connected to places:

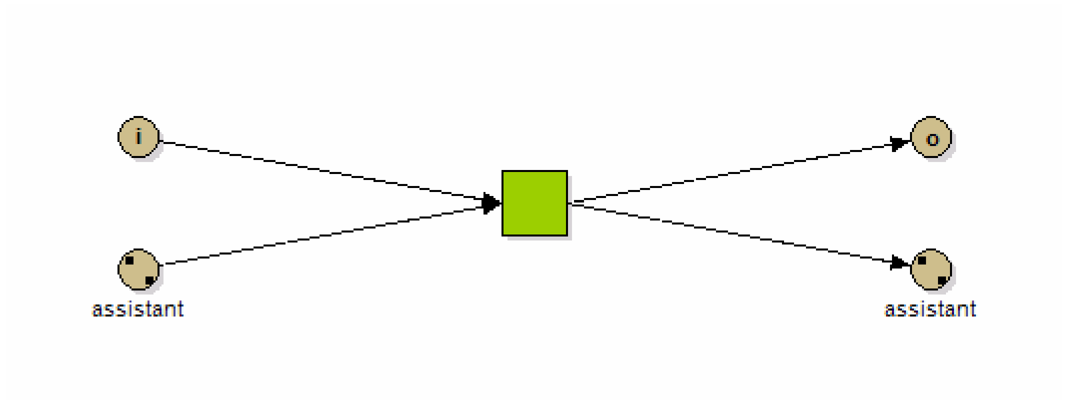


The contents of the subnet are those of a normal net, except that they may contain *pins*. A pin represents an arc connecting the subnet in the surrounding net. For example, the content of the subnet above may look like this:



The left and right pins correspond to the incoming and outgoing arc, respectively. Ambiguities can be clarified by naming:





Note that two pins can refer to the same place, and that pins cannot hold tokens: the tokens shown here are those of the place they refer to, *assistant available*.

Subnets can occur within subnets: in that case, a pin can refer to a pin in the surrounding net.

Execution with subnets is straightforward: all pins are identified with the places they ultimately refer to.

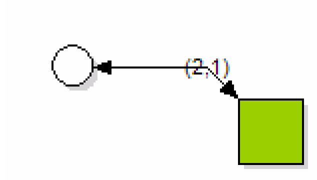
1.5 Special arc types

There are three special types of arcs*:* *combined arcs*, *inhibitor arcs*, and *reset arcs*.

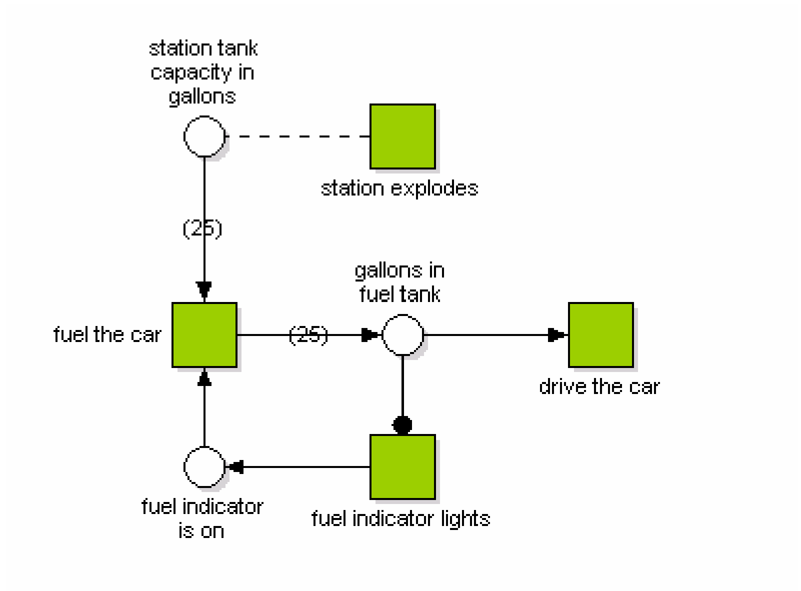
1.5.1 Combined arcs

Most definitions of Petri nets allow multiple arcs to run between the same transition and place. In Yasper, all input and output arcs between the same transition and place are bundled into a single arc.

The *biflow* arc represents a pair of input and output arcs; it already occurred in some of the previous examples. An arc can represent any number of input and output arcs; in general, both numbers appear on the arc:

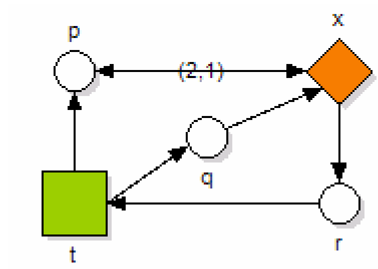


This is rare in practice, but multiple inputs or multiple outputs are more common:



Here, *fuel the car* always brings 25 gallons of fuel from the station tank to the car tank.

A combined arc with one or more multiplicities is usually equivalent to multiple single arcs. In the present Yasper version this is not the case for arcs on XORs.



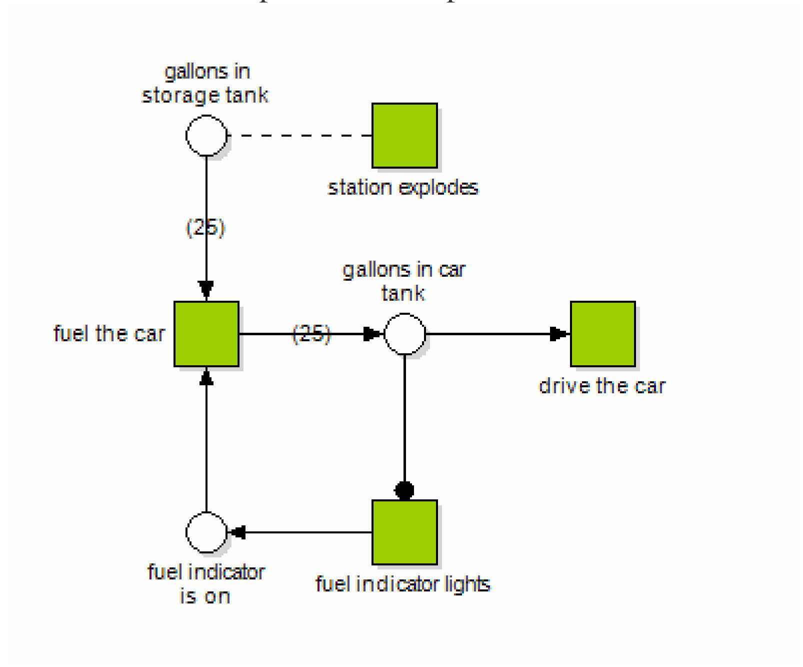
If the arc between p and x behaved like three separate arcs, an execution of x would always pick a single token from p or q, and put a single token into p or r. Instead, when x decides to consume from p, it always takes two tokens from it. It does not guarantee to put a token back into p in that case; it may put it into r instead.

This behavior is odd and may change in the next Yasper version – see section 1.6.

The arcs we have seen thus far are called *flow arcs*, since they represent tokens flowing between transitions (or XORs) and places.

1.5.2 Inhibitor arcs

Let us return to the previous example.



The arc with the dot on the end is an *inhibitor arc*. It specifies a constraint on execution, namely, that the transition at its end can only execute when there are no tokens in the place at the other end. In the example, the fuel indicator lights when the gas tank is empty.

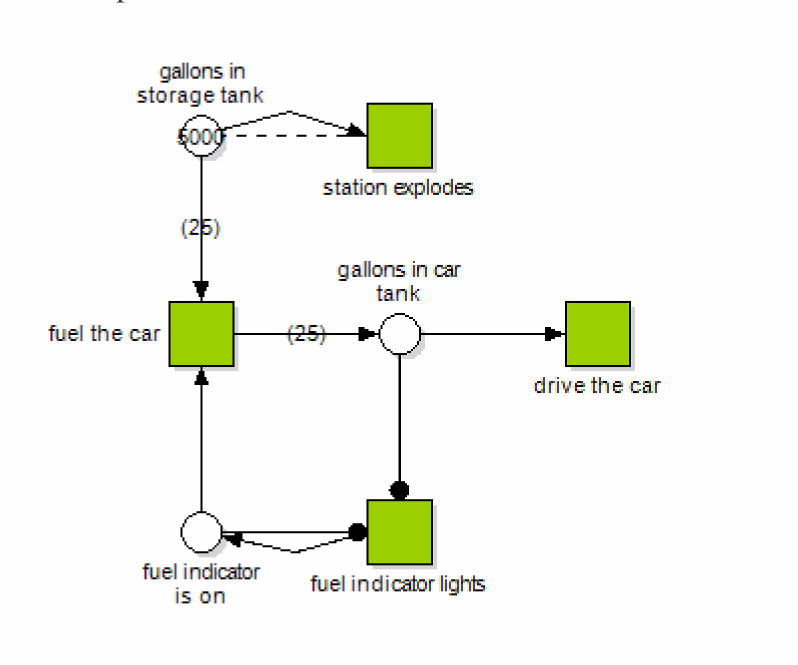
The inhibitor arc is very powerful[5], but it takes skill to apply them correctly; in fact, there is a problem with the way it is used in this model – see the following section.

1.5.3 Reset arcs

A reset arc does not express a constraint, but instead specifies that when the transition at its end executes, it empties the place at the other end.[6] In the example, after the fuel station explodes its storage tank will be empty.

Note that there is, again, a problem with the way in which it is used in the example model.

The two problems mentioned can be corrected as follows.



In the original, once the fuel indicator can go on, it can continue to do so, producing an arbitrary number of tokens in *fuel indicator is on*. But we only have one fuel indicator, so once it is on it cannot switch on again.

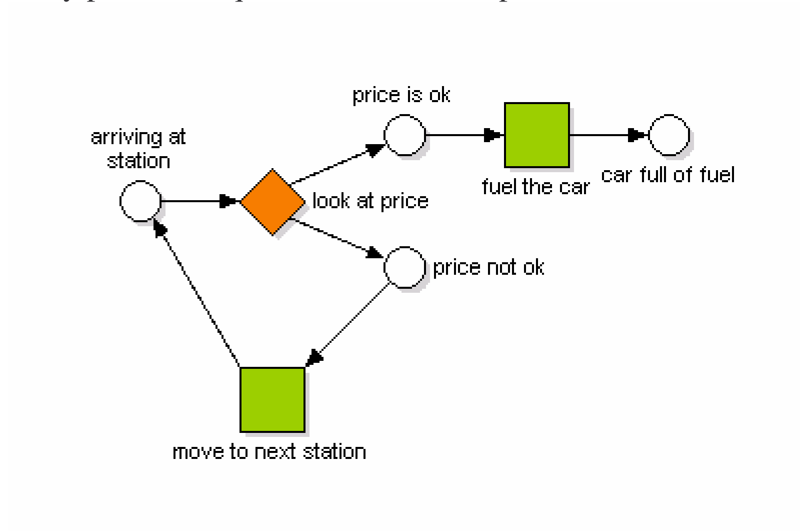
The second, smaller error is that the station can only explode when its tank is nonempty. It also seems a good idea to start out with a nonempty storage tank, otherwise, nothing will ever happen.

[5] Arbitrary computations on numbers of tokens can be expressed with inhibitor arcs: they make Petri nets Turing complete.
[6] *Reset arc* is the name used in the Petri net literature. It can be argued that a reset arc doesn't really perform a *reset*, since it clears the tokens from a place, even when that place wasn't empty initially.

1.6 Extensions for automatic simulation

1.6.1 Arc weights

During simulation, when a XOR split is executed, an output places must be chosen. By default, every place has equal likelihood to be picked.



In this example, the modeler may wish to indicate that when we look at a price, it has a 90% chance of being acceptable. To this end, the modeler can attach *arc weights* to the output arcs of a XOR. It is not possible to do the same for input weights.

Arc weights only take effect in automatic simulation; during manual simulation, the output place is picked by the user.

Arc weights are not displayed in the diagram, and must be edited by opening the property sheet of the XOR and setting the arc weights there. They serve a different purpose than arc multiplicities, discussed in the previous section. In the next version of Yasper we may simplify this situation, and use arc multiplicities on XORs to express arc weights.

1.6.2 Processing time

Two parameters can be set on each transition to specify the time its execution takes

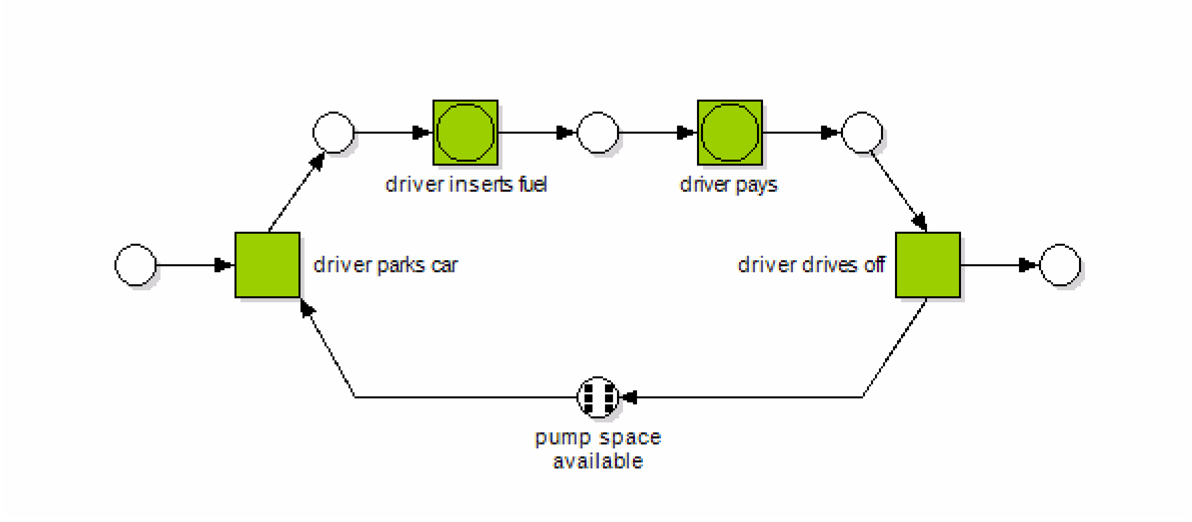
- its mean processing time
- the mean deviation of its processing time

During automatic simulation, the actual processing time spent is determined by these two parameters. If the given deviation is 0, the processing time is always equal to the given mean. Otherwise, whenever the transition starts execution, a processing time is chosen randomly such that on average the chosen value will be the given mean, and the standard deviation from the mean will be the given deviation. This is achieved by using the so-called *gamma distribution*.

Note that processing time solely depends on these two parameters of the transition being executed; no other information can be used. E.g., to express that the same action can be executed by a slow and a fast machine, use two transitions.

Processing time specifications can strongly influence the paths a simulation will take. It is easy to construct nets that are perfectly viable in principle, yet completely lock up with certain choices of processing times. Often such a lockup turns out to reflect a real bottleneck in the process being modeled, and experimenting with processing times is a powerful tool in designing efficient process flows.

In the diagram, timed transitions (with *mean* > 0) are drawn with a circle; the exact parameter values are not shown.



1.6.3 Processing cost

Processing cost is also configurable with two parameters per transition:

- its fixed (setup) cost **f**
- its cost per time unit **v**

The cost of a process execution is **f + v*t**, where **t** is its processing time.

Processing cost cannot influence process execution at all – there is no provision in Yasper to make execution paths depend on cost. Cost calculations only appear in the automatic simulation report, where they provide indications of the expected overall cost of a particular process configuration.

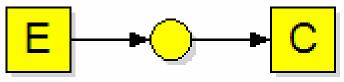
Processing cost parameter values are not shown in the diagram.

1.6.4 Token case

The examples we have seen so far are *workflows*: the process takes tokens from a starting point to an end point. Such tokens represent cases being handled by the workflow. This is a very common type of model.

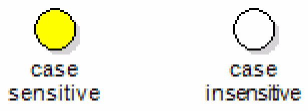
Yasper's automatic simulation feature (see chapter 4) was designed for workflows.
It generates workflow cases, runs them through the model, and produces a report detailing how the average case was handled.

Automatic simulation requires that the start, continuation and end of workflow cases are clearly indicated in the model. This must be done with three specifically designed constructs: *emitors*, *case sensitive places*, and *collectors*, respectively. The smallest model on which automatic simulation will work has one of each:



1.6.4.1 Case sensitive places

A *cased* token belongs to a specific workflow case; an *uncased* token doesn't.

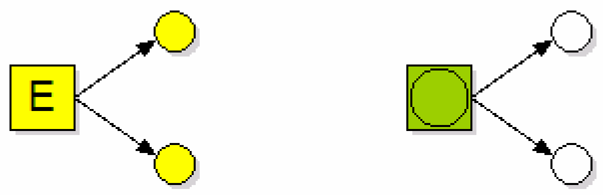


In Yasper, a place can be marked as *case sensitive*, which means it can only hold cased tokens. Any other place can only hold *uncased* tokens.

1.6.4.2 Emitors

An emitor is a source of tokens. During simulation, it generates tokens one at a time.
Every token generated by an emitor establishes a workflow case: a workflow case is defined by the emitor that generated it and the generation step at which it was generated.

Like a transition, an emitor emits tokens on all its output places at once.
It has the time parameters of a timed transition to control token generation during simulation.
An emitor cannot have input places. It is, therefore, much like a timed transition without input places.

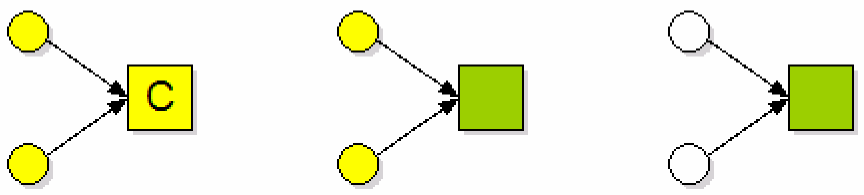


Normal transitions and XORs without input places can be created in the editor.
However, simulation refuses to run when they are present: it requires all tokens to be generated with emitors. This helps prevent accidental modeling errors.

1.6.4.3 Collectors

A collector is a sink for tokens. It consumes them like a normal transition.
A collector cannot have output places.

Collectors mark the expected end points of workflow cases. Automatic simulation reports on workflow cases flowing from emitors to collectors (cf. section 4.1).



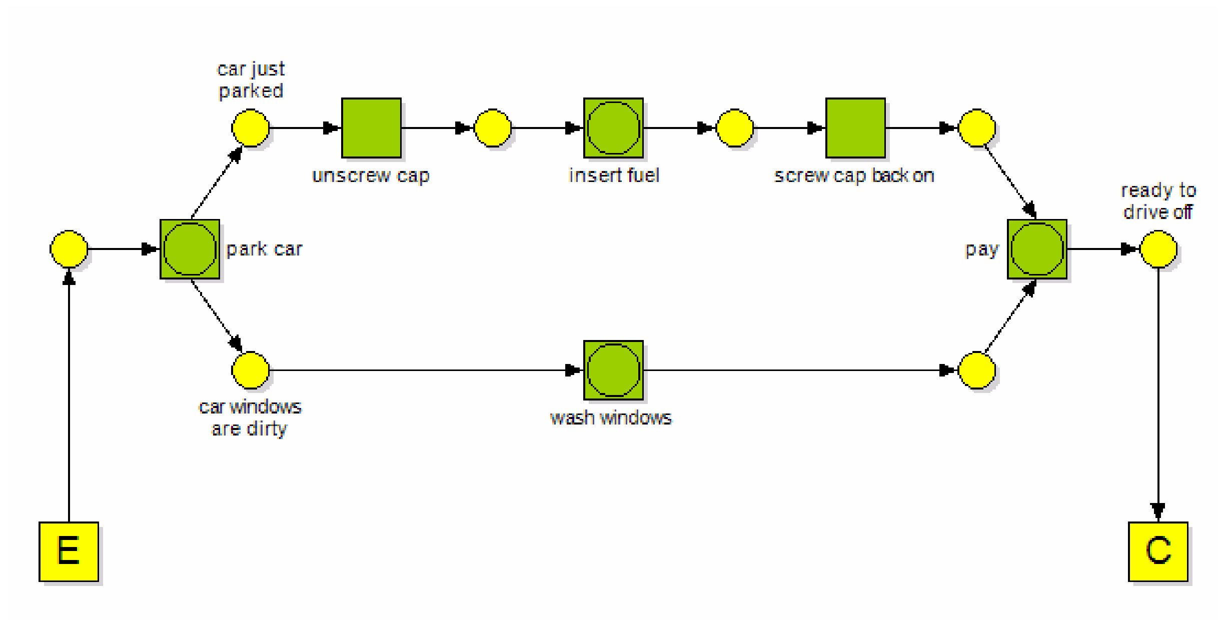
Normal transitions and XORs without output places can be created in the editor.
Simulation will still run when they are present. However, the termination of a workflow must be marked explicitly with a collector. Simulation refuses to run when there is no path along cases places from an emitor to a collector.

The simulator supports models with multiple emitors and/or multiple collectors. This is mostly useful for simulating models in parallel.

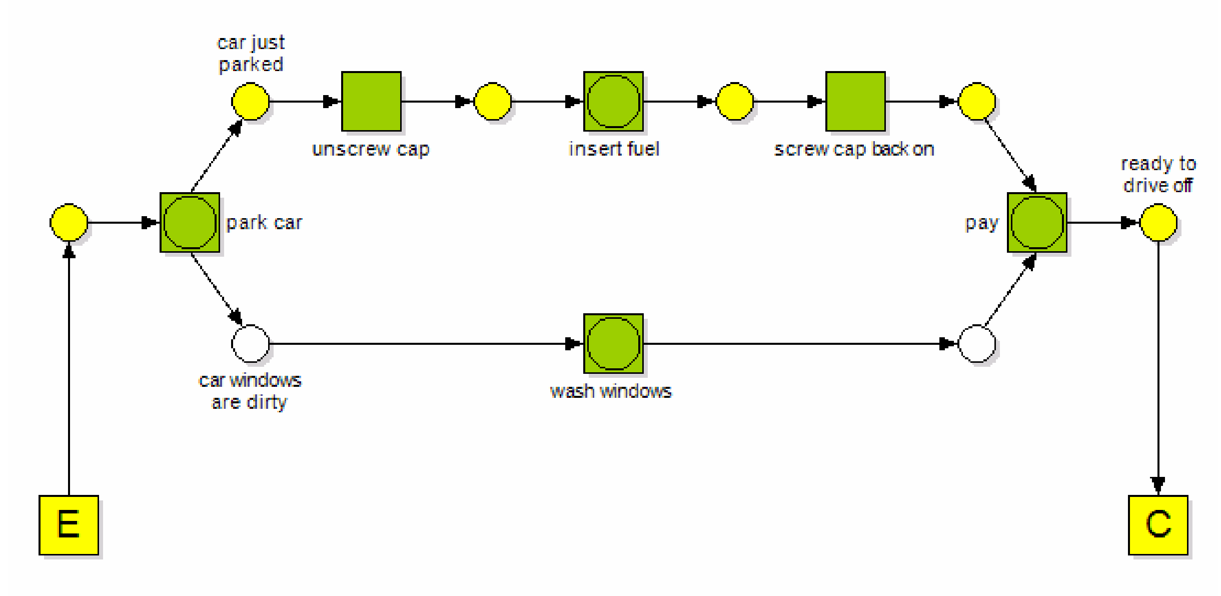
1.6.4.4 Case sensitive execution

Transition execution works on a case-by-case basis. To be exact, the execution rule is adapted as follows: all tokens consumed from case sensitive places must be of the same case, and that case will be passed on into any case sensitive output places.

The examples so far were workflows, so they are easily modified for simulation.

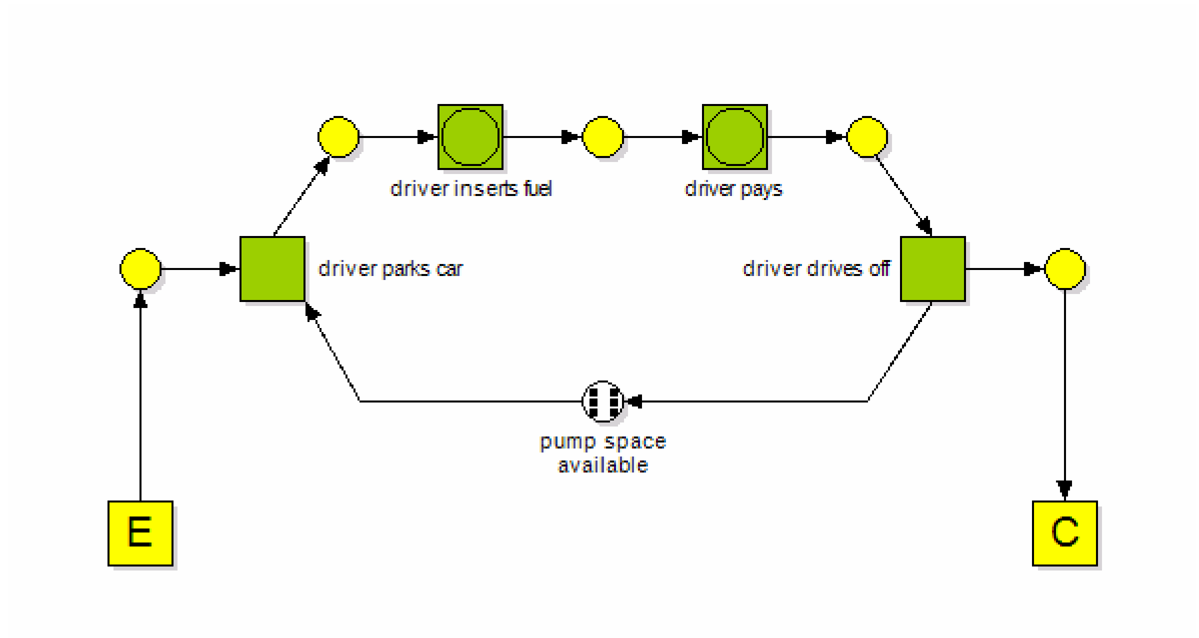


The case-by-case behavior of transitions becomes relevant as soon as multiple cases enter the model. In this example, *pay* is only possible when fueling and washing have both finished for the same car. Contrast this with the following:



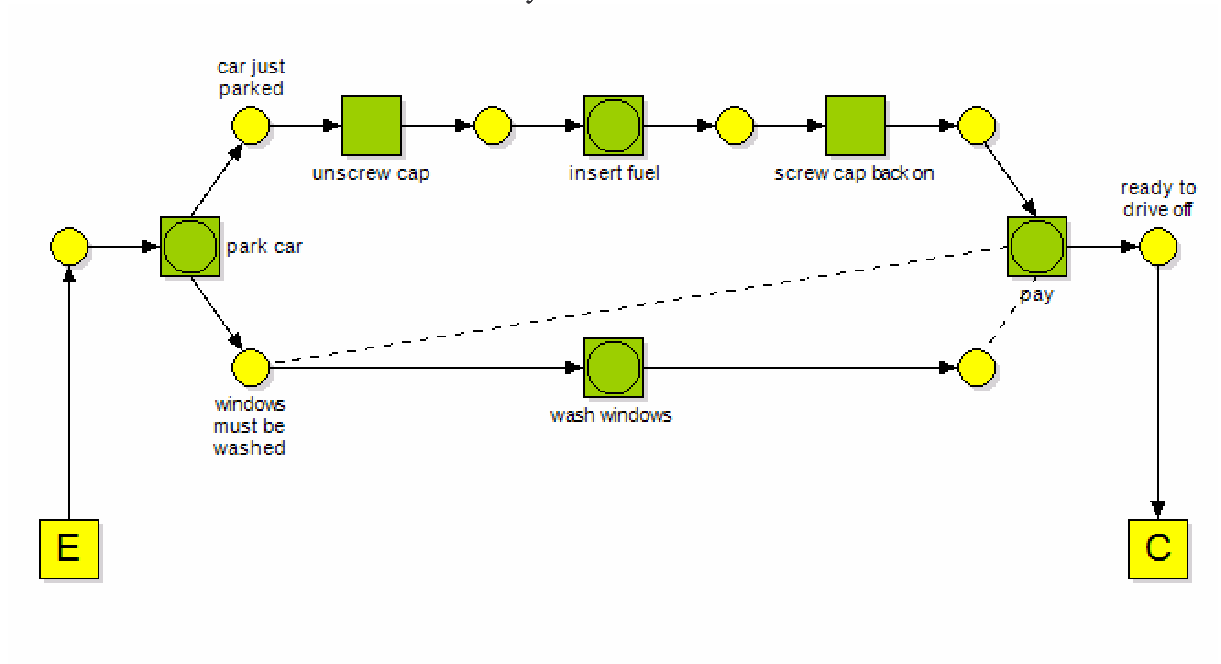
Here, *pay* is possible whenever fueling has finished and washing has finished for some car. Clearly, not the intended semantics, and if *wash windows* takes a relatively long time, the difference will be clearly noticeable in the simulation report.

A common use of case insensitive places is to model resource contention.



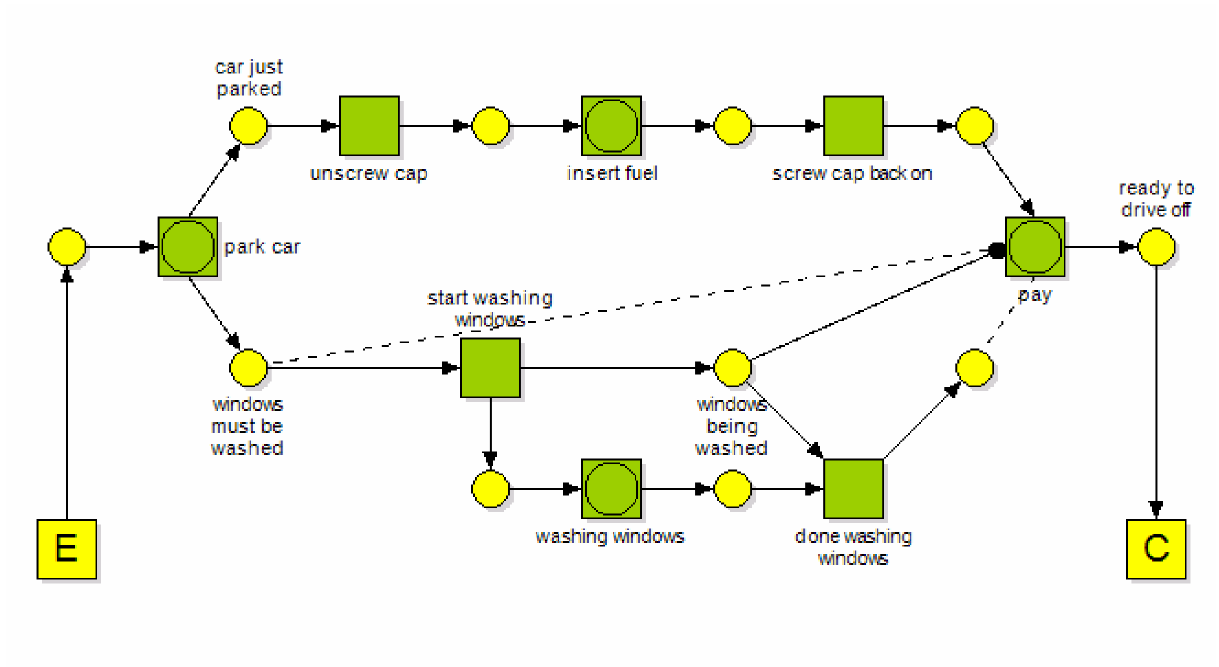
Here, *pump space available* must be case insensitive, since it describes how cases affect each other. If it was case sensitive, *driver parks car* could never execute.

Inhibitor and reset arcs also work case-by-case:



Here, we pay and drive off, whether or not the windows have been washed. Since the resets are by case, payment for one car doesn't affect whether the windows of other cars are washed.

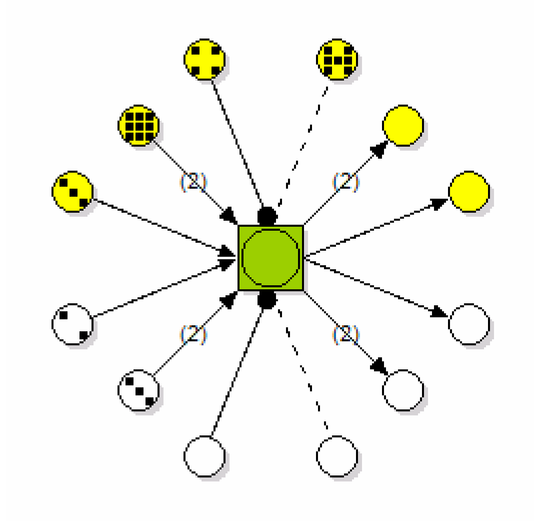
However, *pay* also proceeds if the car was being washed, leaving a token behind in wash windows that will never be cleaned up: this model does not *complete properly* (as defined in section 1.1.3.2). Not only will this cause problems for the simulation report, we actually want to wait when the car is being washed.



Note that the inhibitor only suspends payment while the *same* car is being washed.
To suspend it while *any* car is being washed, *windows being washed* must be case insensitive.

We now provide an exact definition of what it means for a transition or XOR to execute. As in the case of basic Petri nets, this definition completely defines process execution – there are no other constraints or possibilities.

In general, a transition may have any combination of cased and uncased places attached to it with any combination of arc types, and with any assignment of roles, time and cost.



Execution of a transition can start whenever the following conditions are met:

- if any roles are assigned to it, a role is available
- all case insensitive places connected to it with inhibitor arcs, if any, are empty

- every input place contains as many tokens as there are arcs from it to the transition (counting multiplicities), all having the same case on the case sensitive input places, such that
- no case sensitive places connected to it with inhibitor arcs contain any tokens with that case

When execution of a transition starts, the following happens, atomically:

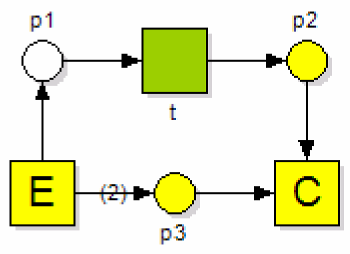
- a set of tokens on the input places as just described is consumed (removed from their input places)
- if any roles are assigned to the transition, one is occupied (i.e. its availability, initially equal to its capacity, decreases by 1)
- all tokens are removed from case insensitive places connected to the transition with reset arcs
- if any cased tokens were consumed, all tokens with that case are removed from case sensitive places connected to the transition with reset arcs
- the processing time for this job is determined
- the job is started

Note that a transition without input places can still consume (the set of tokens consumed is empty).

When the processing time of a job has expired, the following happens, atomically:

- if it occupied a role, it is released (i.e. its availability increases by 1)
- if starting the job consumed any cased tokens, to every output place as many tokens are added as there are arcs to it from the transition (counting multiplicities), with the same case if the place is case sensitive, and without case otherwise
- if starting the job did not consume any cased tokens, to every case insensitive place as many tokens are added as there are arcs to it from the transition (counting multiplicities), while no tokens are added to any case sensitive places

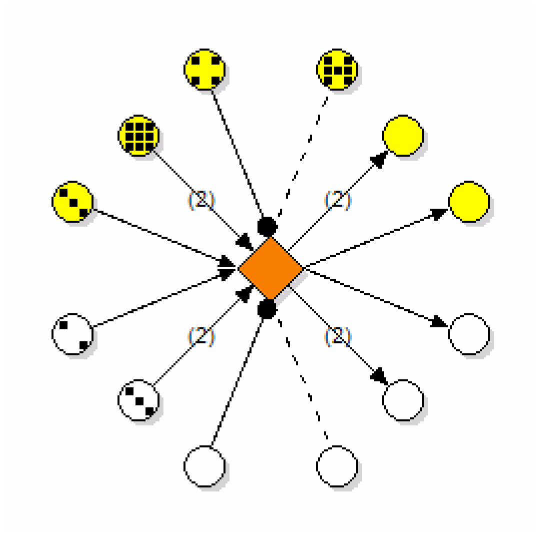
This last rule means that flow arcs in models can be superfluous and misleading:



Here, whenever *t* fires, no tokens will be put into *p2*. The arc between *t* and *p2* might just as well be omitted.

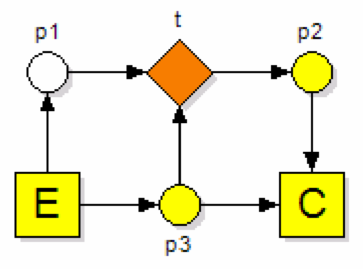
The emission of cases by an emitor is identical to the expiration of a job, except that the emitor generates a new case and puts cased tokens for that case into its cased output places, and will then use its processing time parameters to determine the next time to generate a case.

E.g., when the emitor in the example generates a case, it puts two tokens with that case into *p3* and an uncased token into *p1*.



For XORs the rules are identical, except that XORs consume from only one of the input places, if any, and produce to only one of the output places, if any.

Like a normal transition, an XOR may fail to produce a token in a cased output place:



Here, when *t* consumes from *p1* it will not put a token into *p2*, but when it consumes from *p3*, it will.

Multiple transitions may simultaneously be able to start and/or finish jobs, and a transition may simultaneously be able to start and/or finish different jobs. In Yasper's manual and automatic simulation modes, finishing always takes precedence. Furthermore, in automatic simulation mode, starting jobs for the earliest generated case always takes precedence. Other than that, execution order is free; in manual simulation mode, the user gets to choose, while automatic simulation mode chooses randomly.

The state of execution at any particular point is completely described by the state of all tokens: for each token its state is described by its location (a place, XOR, or transition), the executing role (if any), and the remaining processing time (if being processed by a timed transition). Each execution step either puts a token inside a transition or collector (consuming 0 or more tokens from places) or releases one from a transition or emitor (producing 0 or more tokens into places).

1.6.4.5 Case sensitive proper completion

When a cased token arrives at a collector, we say the collector *collects* the case. If no other token for the case remains, we say the case *completes.* A model in which every possible generated case is guaranteed to complete has *proper completion*.

In nets with proper completion, cases cannot continue forever, and cannot terminate without completing (i.e. their last token disappears through an outputless transition or XOR). Proper completion does not rule out the possibility of cases being collected multiple times.

1.7 Token color

In *colored Petri nets*, tokens can have values that can be tested and modified within the net. For instance, a transition can be specified to consume only integer tokens with a value smaller than 10, and to increment the value on tokens consumed.[7]

Color allows data to be associated with a process model.

The *token case* feature (section 1.6.4) can be regarded as a specific, limited application of token color. Yasper does not presently support any other use of color.

[7] Process modeling tools such as CPN Tools (http://wiki.daimi.au.dk/cpntools/) and ExSpecT (http://www.exspect.com/) are based on colored Petri nets.

1.8 Data stores

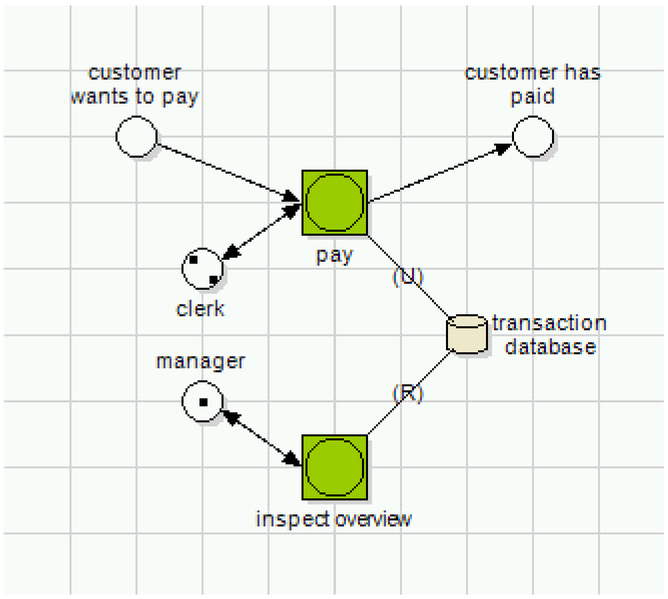
A natural way to associate data with a process is to say that the data reside in a data store, and the process affects the data in a particular way. A typical example is a process interacting with a relational database by issuing SQL query statements.

Yasper allows such data stores to be represented explicitly.[8] A store is like a place in that it is connected with tasks (transitions or XORs).[9] For arcs connected to stores, the normal arc types do not apply, since tokens cannot be added to or removed from a store. Instead, it can be specified whether the task can

- create
- read
- update and/or
- delete

(the) value(s) in the store. Any combination can be specified, as long as at least one of the options is used.

Specifying this information (known collectively as a *CRUD matrix*) is a well-established technique; it allows basic consistency checking on the use of data.[10]



In this example, *pay* modifies the transaction database, while *inspect overview* reads it, but does not modify it.

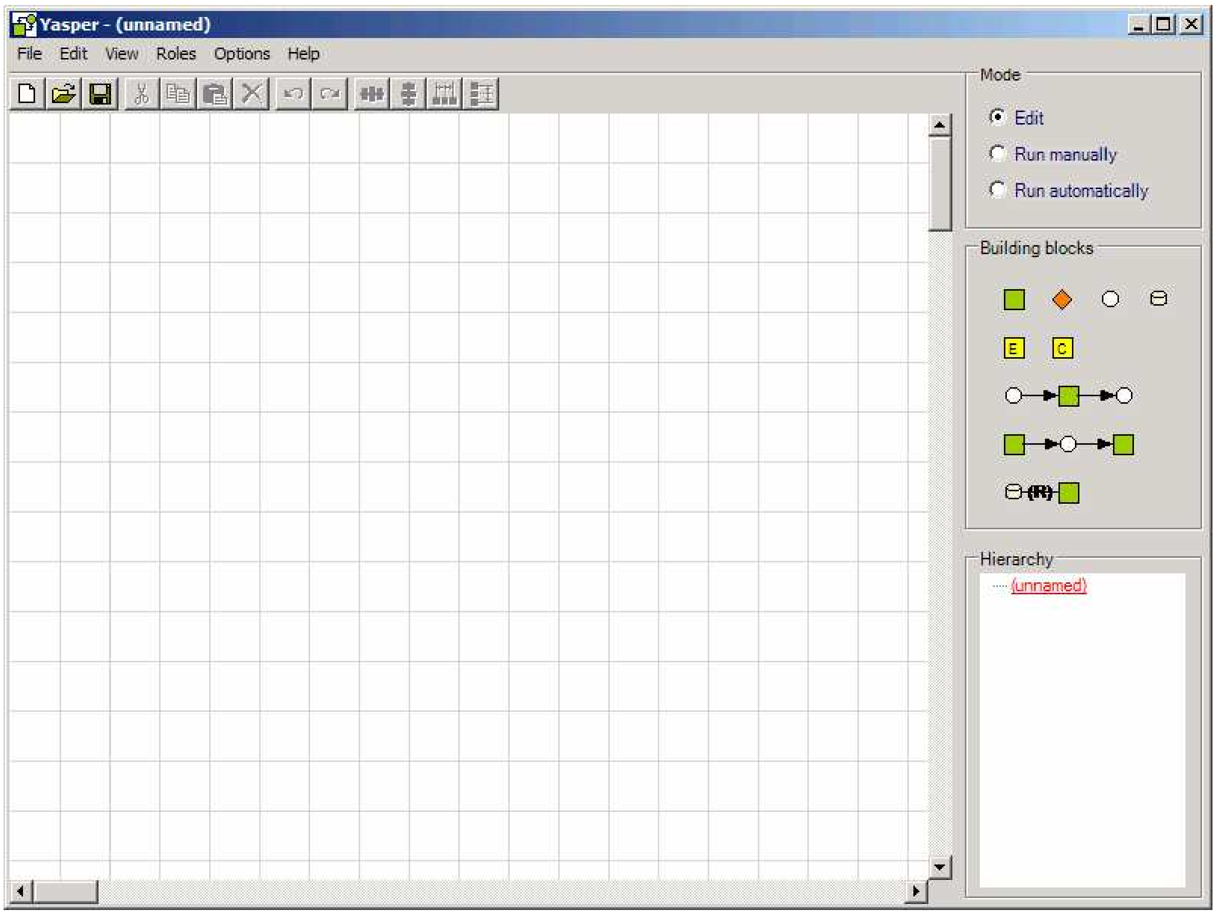
[8] The store concept was taken from ExSpecT.
[9] In colored Petri nets, the contents of the store can be specified exactly as a token value.
[10] Yasper does not provide any direct support for such checks.

The present version does not use the specifications of stores and store arcs in any way; simulations ignore them. A future version of Yasper can support full details in this specification through a plug-in mechanism (see previous section).

2 The Yasper editor

2.1 Starting with an empty model

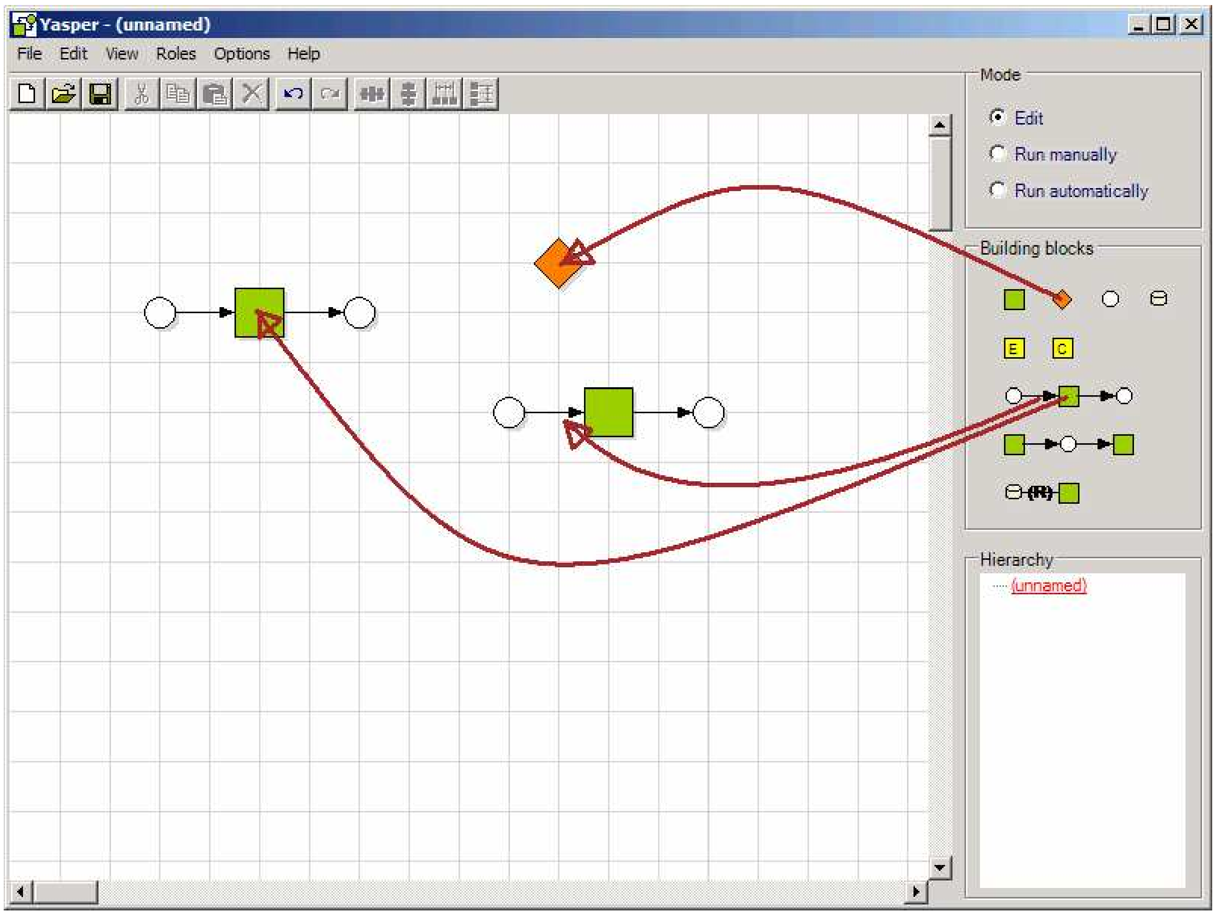
Start Yasper from the Windows Start menu or by double-clicking on the Yasper desktop short-cut. The main Yasper window comes up in edit mode.



A quick overview of the interface elements shown and their purpose in editing:

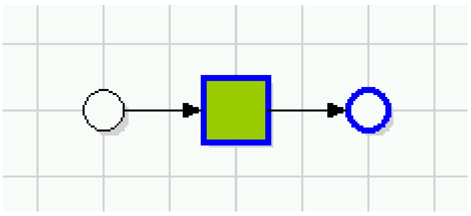
- the diagram canvas: to modify the model with direct manipulation
- the *File* menu: to load and save models in various file formats
- the *Edit* menu: to execute certain modifications (e.g. alignments) and copy/paste
- the *View* menu: to configure how models are displayed
- the *Roles* menu: to specify roles (they are not shown in the diagram)
- the *Options* menu: to configure how the editor behaves
- the toolbar icons: to directly execute many *File* and *Edit* menu operations
- the mode buttons: to switch between editing and simulation modes
- the building blocks: to add elements to the diagram
- the hierarchy view: to navigate and operate on the subnet hierarchy

Initially, the model is empty. To start editing, pick up a building block with the mouse and drop it onto the diagram canvas:


Yasper - (unnamed)
File Edit View Roles Options Help
Mode
Edit
Run manually
Run automatically
Building blocks
E
C
(R)
Hierarchy
(unnamed)

2.2 Selecting and deselecting elements

All diagram manipulations work on a *selection* of elements; the selected elements are shown with blue borders. Both *nodes* (transitions, places, stores, subnets) and *arcs* (connections) can be selected.



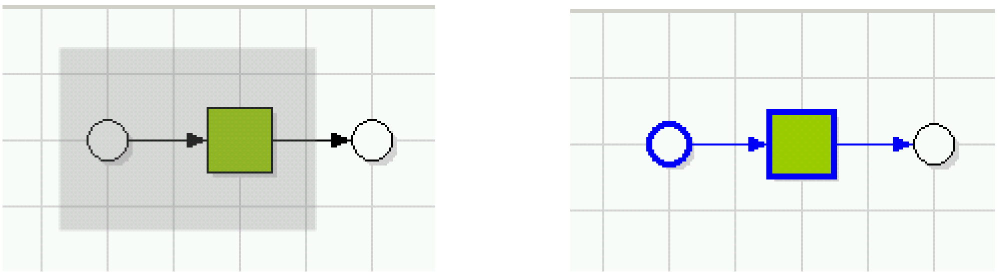
To select an element, click on it.

To add an element to the selection, shift-click on it.

To remove an element from the selection, shift-click on it.

To select all elements in the diagram, choose *Select All* in the Edit menu, or press *Ctrl-A*.

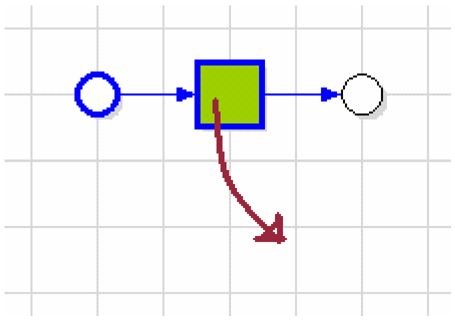
To select a range of elements, click in the background, and drag the mouse across.
A gray rectangle appears; the overlapping elements will be selected.

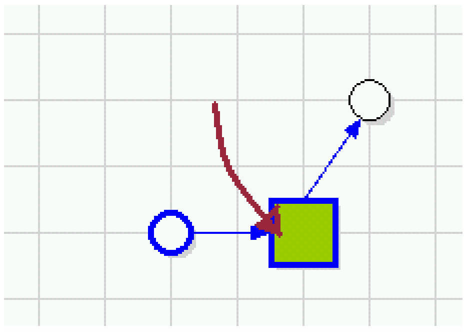


To add the elements to the existing selection, hold *Shift* while dragging.

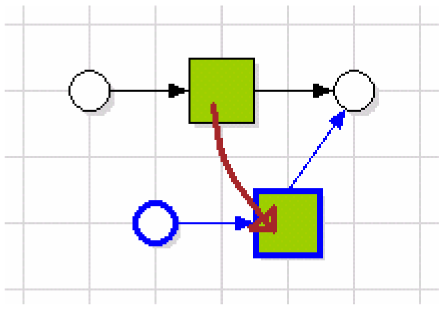
2.3 Moving elements

To move the selected elements, click on one of them and drag the mouse in the desired direction.





To copy the selection and move the copy, hold down the *Ctrl* key while dragging.



Arcs always stay connected.

After moving elements, automatic repositioning may automatically occur:

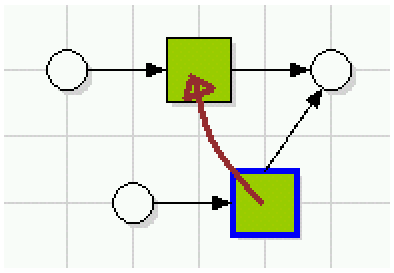
- merging (see next section): a node is swallowed by another;
- nodes are centered on gridline crossings,
  if *Nodes snap to grid* is enabled in the *Options* menu;
- the selection is moved away, if one or more nodes overlap with other nodes;
- nodes that move outside the diagram boundaries are pushed back on.

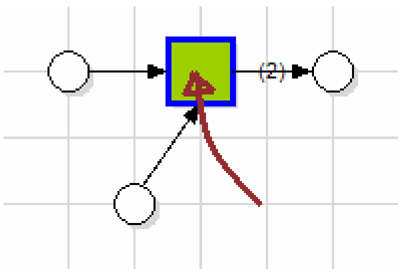
Moves can also be executed with the arrow keys. This allows precise placement.

2.4 Merging elements

Moving a node onto another will fuse them together, under the following conditions:

- the two nodes are compatible
- the node is the only element being moved
- node merging is enabled in the *Options* menu





Arcs can be fused as a side effect of nodes being fused.

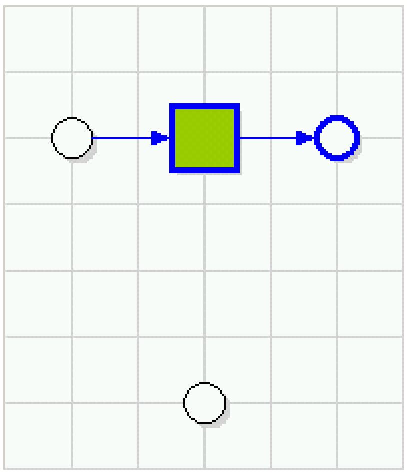
2.5 Deleting elements

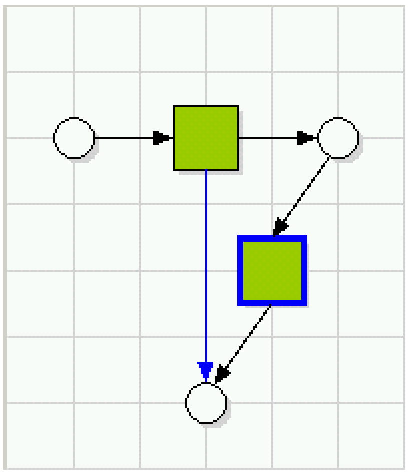
To delete the selected elements, select *Delete* from the *Edit* menu or click the *Delete* toolbar icon, or press the *Delete* key. For a single element, another option is o right-click it and select *delete*.

Deleting a node also deletes all arcs attached to it, since arcs must always be connected on both sides.

2.6 Connecting nodes

To create connections from the selected node(s) to another node, *Ctrl*-click on that node, which must not be part of the selection (if it is, the node is copied instead, cf. section 2.3).



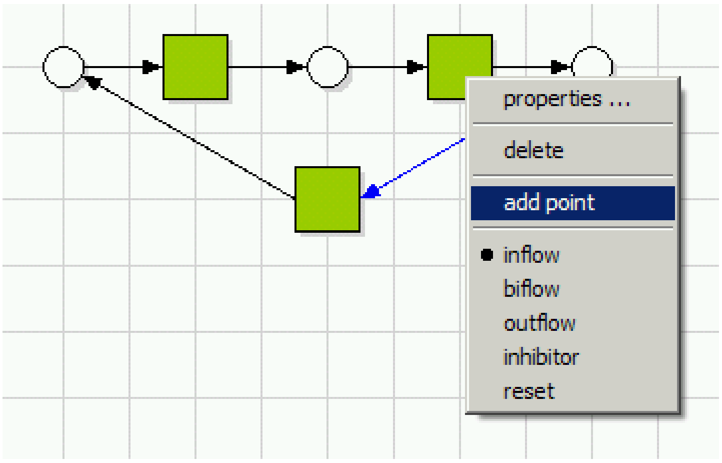


Only valid connections will be made.

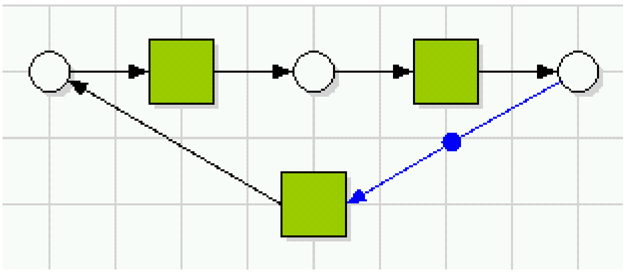
Intermediate nodes are inserted when necessary.

2.7 Arc support points

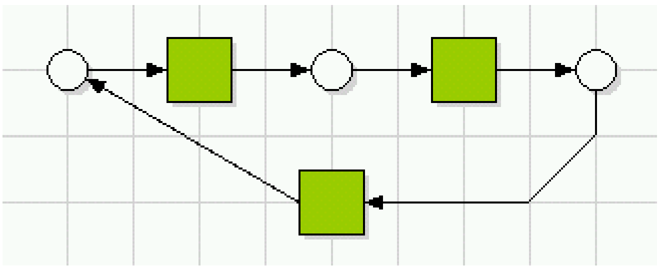
To add an intermediate point[11] to an arc, right-click on it and select *add point*.



To move a support point, left-click on it and drag.



Arbitrarily many support points can be added to an arc.

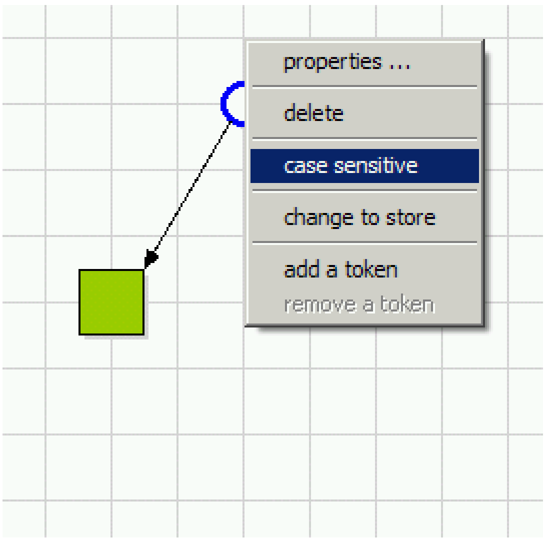


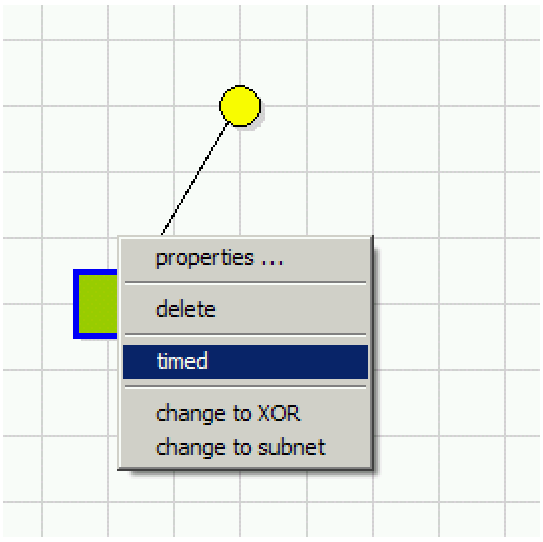
To delete a support point, right-click on it and select *delete point*.

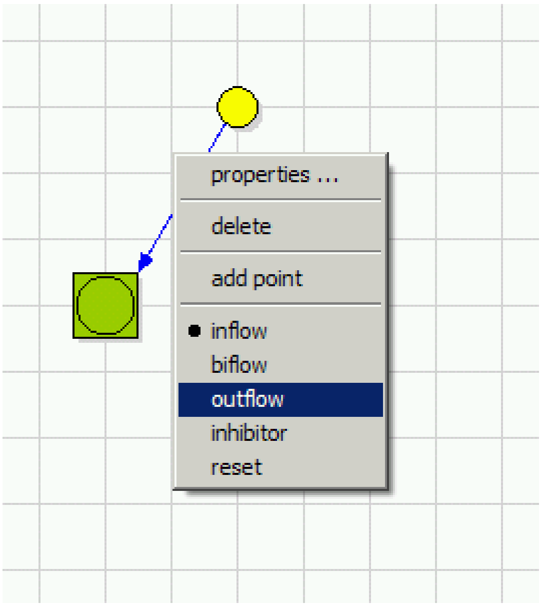
[11] No standard term appears to exist for the intermediate points of polylines; we use *support point*.

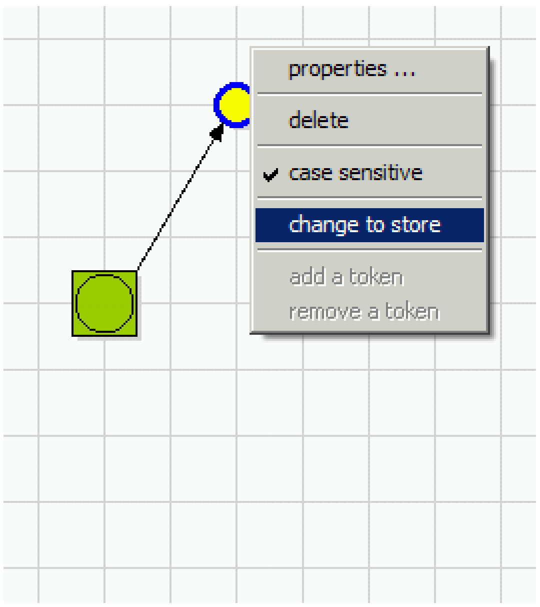
2.8 Changing the type of an element

To change the type of an element, right-click on it and select the new type.









Changing the type of a node automatically adjusts the types of the attached arcs, when necessary.

The available node and arc types and their meanings are discussed in chapter 1.

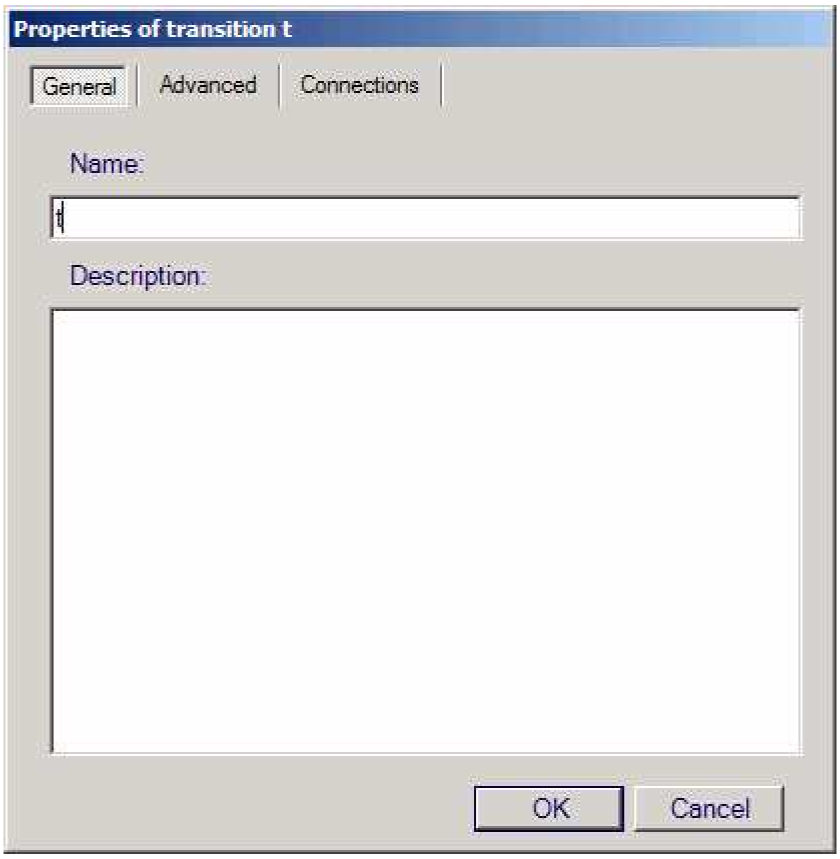
2.9 Specifying additional properties

Model elements have additional properties that can be set and inspected in the element's property sheet.

To bring up the property sheet for an element, right-click on the element and select *properties …*, or double-click on it (except on subnets and pins).

To bring up the property sheet for the current net, right-click in the background and select *properties …* or double-click in the background.

Every element has a name and a description. They can be entered in the property sheet. The name will appear in the diagram if *Show names* is enabled in the *View* menu. Names are optional and do not have to be unique.


Properties of transition t

General | Advanced | Connections

Name: t

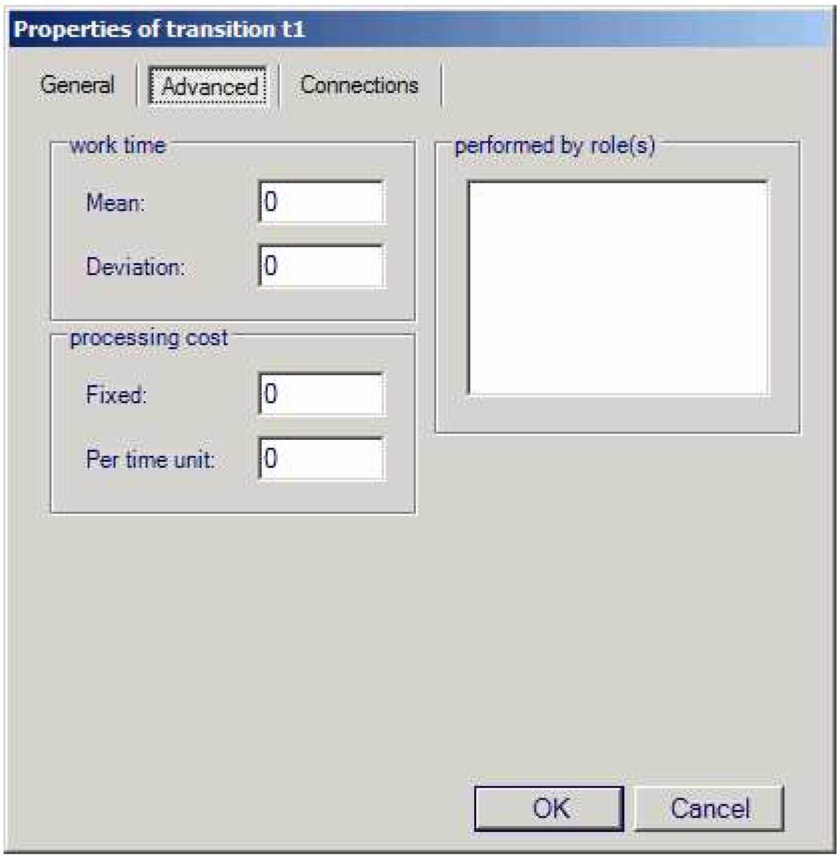
Description:

OK | Cancel

Some elements have additional properties. To inspect or modify them, click on *Advanced* and fill out the relevant form field(s). You will be prevented from entering invalid information.

The various advanced properties are the following.

2.9.1 Transition properties


Properties of transition t1

work time

Mean:	0
Deviation:	0

processing cost

Fixed:	0
Per time unit:	0

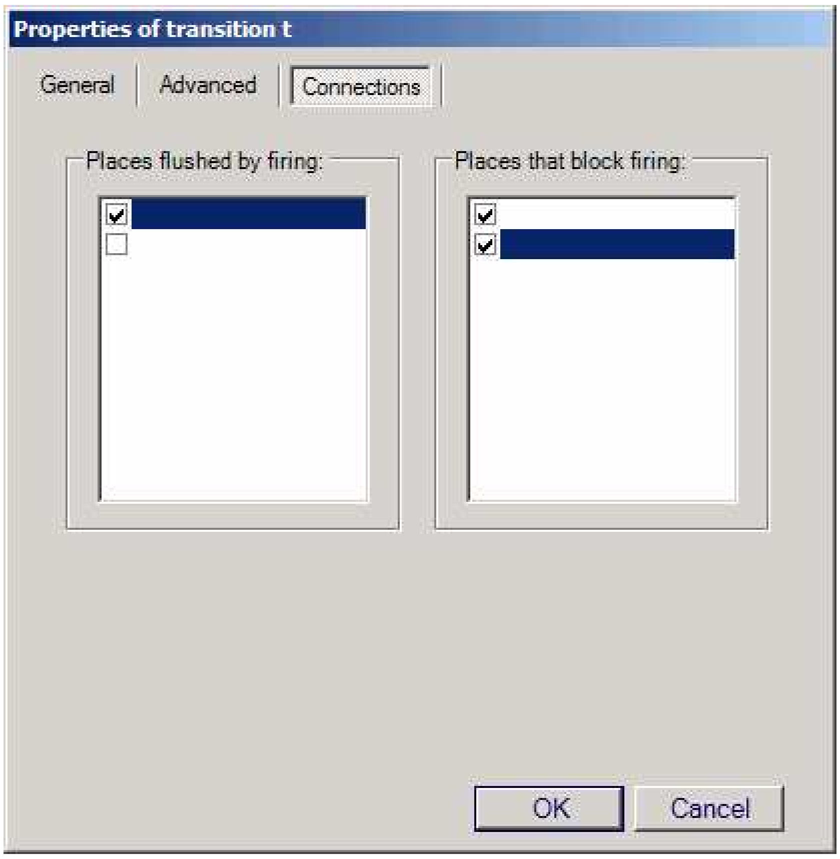
performed by role(s)

For a transition, the parameters for processing time and cost (see sections 1.6.2 and 1.6.3) can be entered here. They must be decimal numbers.[12]

For role assignment see section 2.11.2.

The *Connections* tab can be used to add and remove reset and inhibitor arcs.
Whether the arcs in question appear in the diagram is configurable in the *View* menu.

[12] Caveat: even though Yasper's interface language is English, the decimal separator to use here is the one set in your Windows settings. This may change in the next version of Yasper.



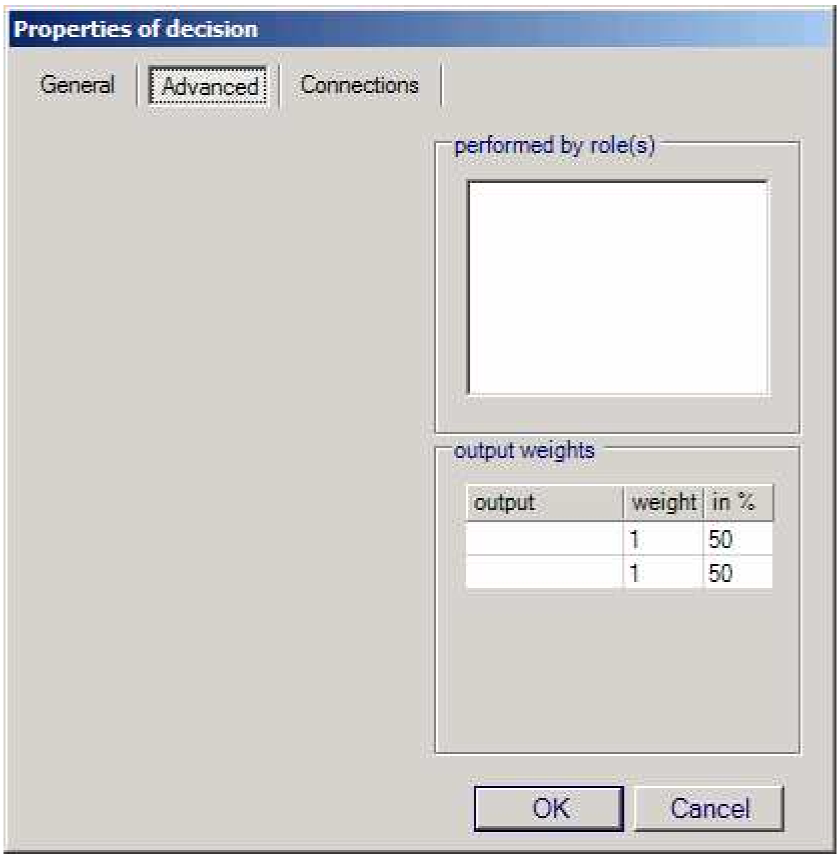
Places flushed by firing:

- [✓]
- [ ]

Places that block firing:

- [✓]
- [✓]

2.9.2 XOR properties



output	weight	in %
	1	50
	1	50

XORs allow relative output weights to be set (see section 1.6.1).
Like processing time and cost, output weights can have decimal separators.

Type or edit a number in the *weight* column to adjust it; the percentages are recomputed automatically.

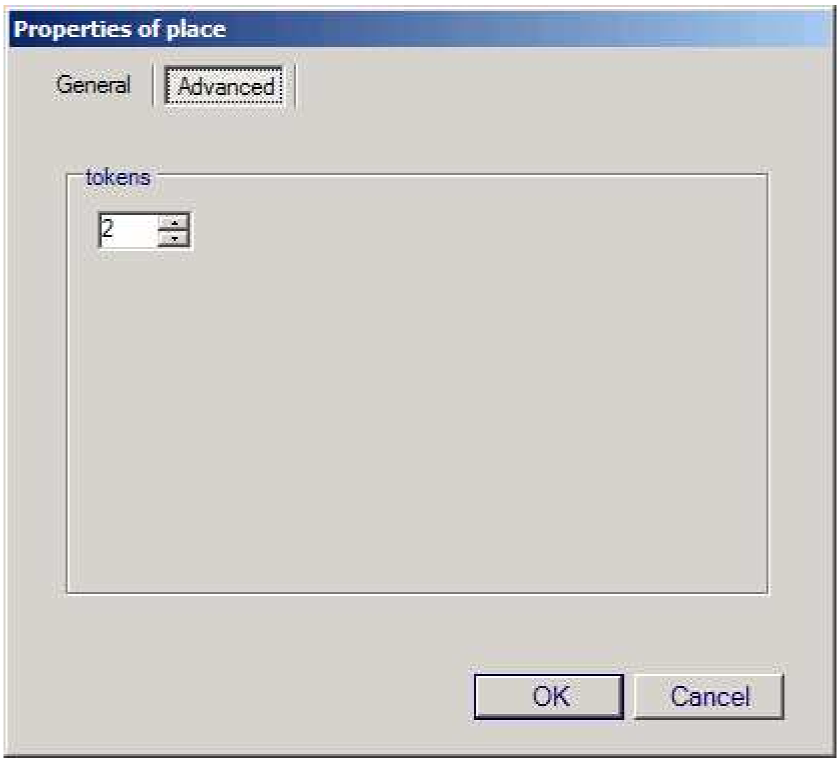
Type or edit in the output column to adjust or set the name of the place in question.

2.9.3 Place properties

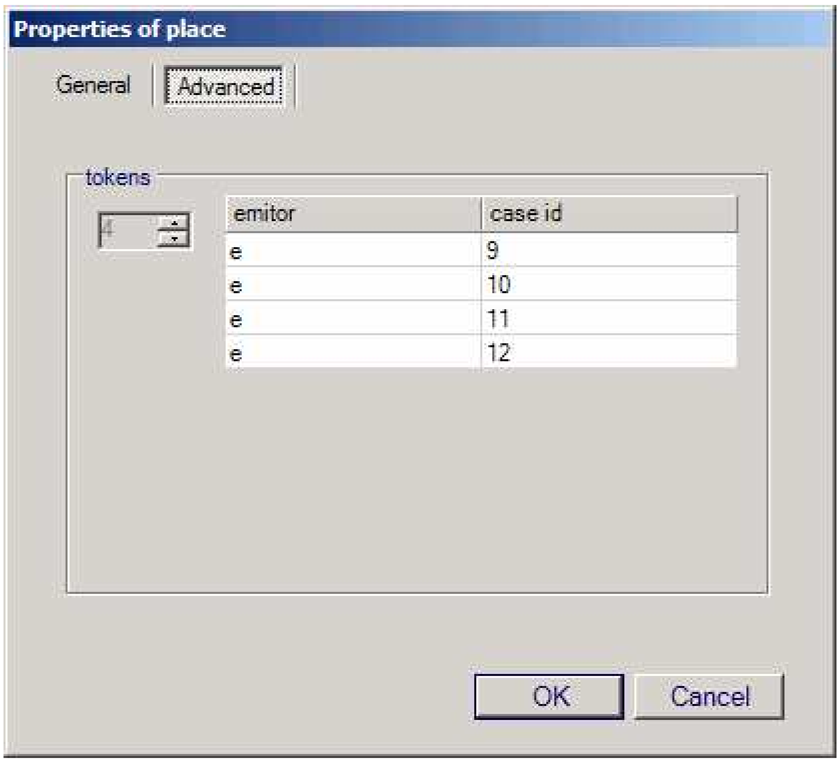
For places, the *Advanced* tab displays their tokens.

To change the number of tokens in the place, type it directly, or click on the tiny triangles next to it. A value can be entered for a token, but this is of limited use, since values are meaningless to Yasper[13] – see section 1.7.

[13] Also, values are not included in the saved document; this will change in a future Yasper version.



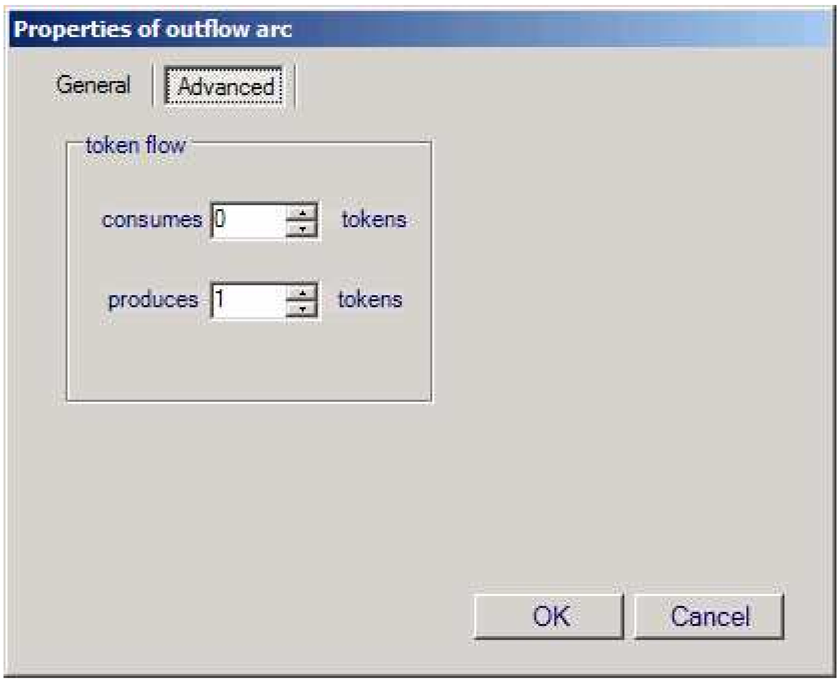
For case sensitive places, tokens are displayed, but cannot be edited.
The case of tokens is shown: emitor name and case sequence number.



emitor	case id
e	9
e	10
e	11
e	12

2.9.4 Arc properties

To set the number of consumed and/or produced tokens on a flow arc, open the *Advanced* tab of its property sheet and adjust the numbers shown.


Properties of outflow arc

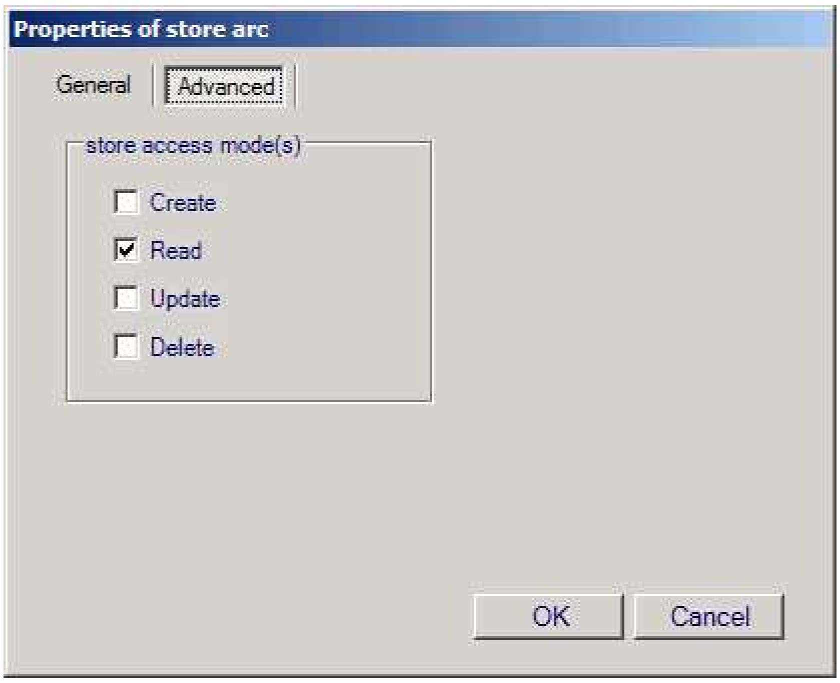
token flow

consumes 0 tokens

produces 1 tokens

The numbers cannot both be 0. With the numbers set to *1,0* or *0,1*, the arc is an inflow or outflow, respectively; otherwise, it is a *combined arc* (see section 1.5.1).

On arcs attached to stores, the kind(s) of store access performed can be set:


Properties of store arc

store access mode(s)

- [ ] Create
- [✓] Read
- [ ] Update
- [ ] Delete

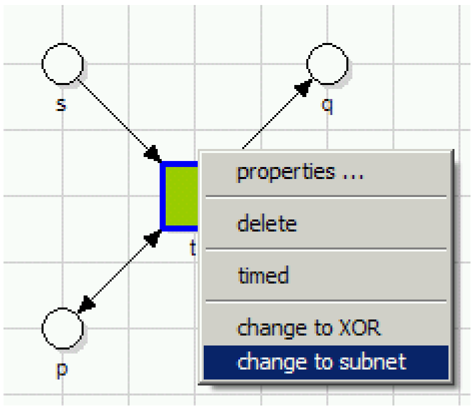
At least one must be selected.

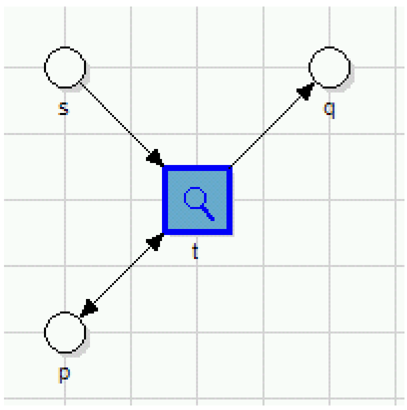
2.10 Working with subnets

Subnets distribute models across multiple screens; full details in section 1.4.

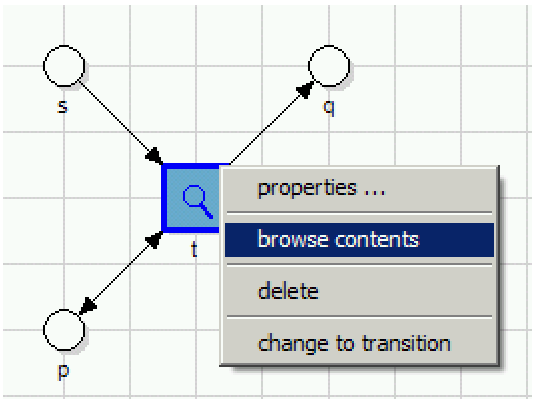
2.10.1 Creating and navigating subnets

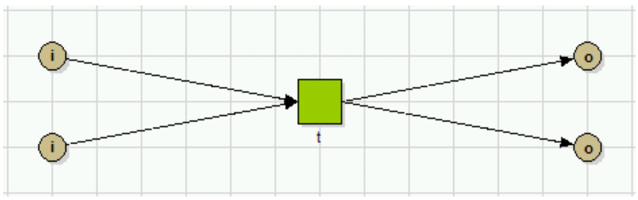
To convert a transition to a subnet, right-click on it and select *change to subnet*.





To move into the subnet, double-click on it, or right-click and select *browse contents*.





To move up to the surrounding (sub)net, double-click on a pin.

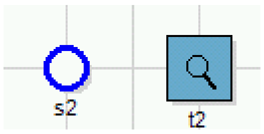
When a subnet is deleted or converted into a transition, its contents are destroyed.

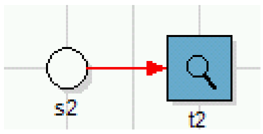
2.10.2 Connecting across subnet boundaries

Transitions inside a subnet can be connected to places outside. To show such a connection in the subnet content, a *pin* exists within the subnet for every arc attached to the subnet in the surrounding net.

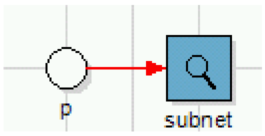
Pins can be manipulated like places, with two restrictions.[14] First, pins cannot be copied or merged. Second, at most one arc can be attached to a pin, directly or indirectly; attempts to attach another will fail.

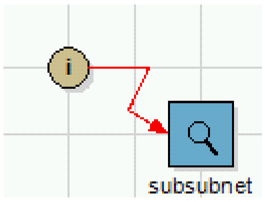
To add a pin to a subnet, connect the subnet to a pin or place.

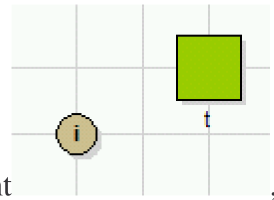




An arc on a subnet is shown in red whenever its pin inside is not connected to a transition, directly or indirectly. For example, in



with content


with content


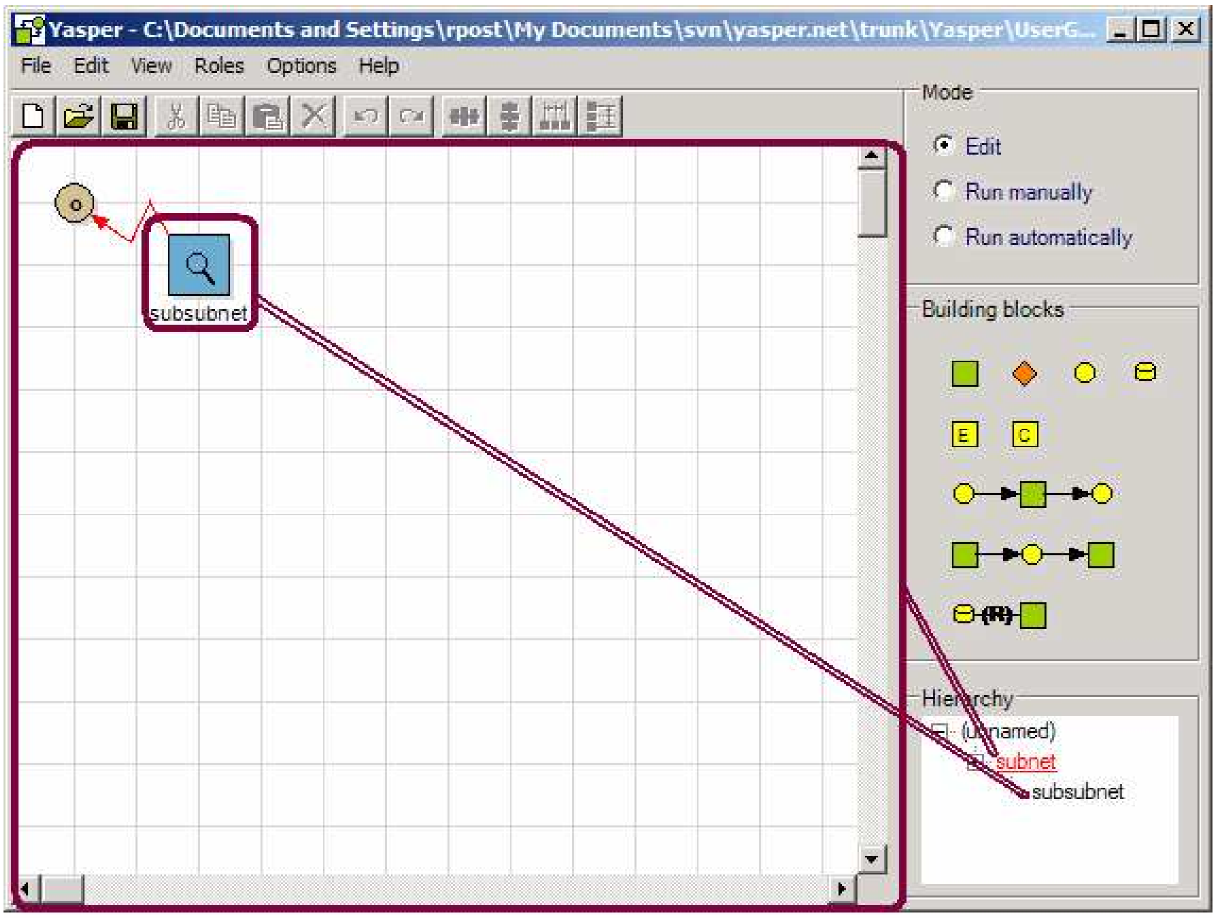
,

creating an arc between *t* and the pin will turn the two red arcs black.

2.10.3 Working with the tree view

At the bottom right, Yasper displays a tree view of the subnet hierarchy. The last example, with a subnet inside a subnet, is displayed as follows.

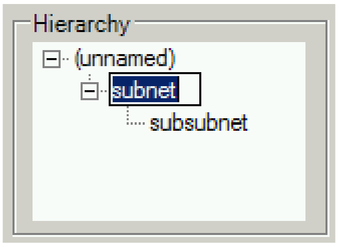
[14] These restrictions will be removed in a future version of Yasper.


Yasper - C:\Documents and Settings\rpost\My Documents\svn\yasper.net\trunk\Yasper\UserG...
File Edit View Roles Options Help
Mode
Edit
Run manually
Run automatically
Building blocks
E
C
R
subsubnet
Hierarchy
(unnamed)
subnet
subsubnet

The net presently displayed is marked in the tree view: its name is in red and underlined.

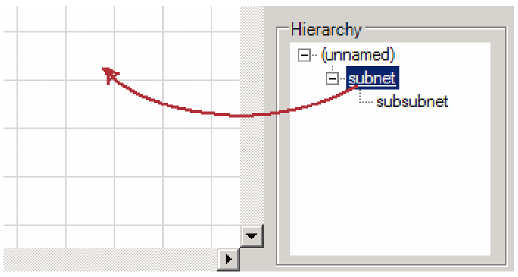
To navigate to a (sub)net of choice, click on its item in the tree view.

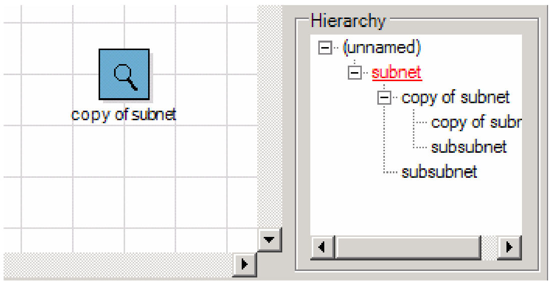
To rename a net, click on its tree view item, wait, then click on it again.[15] A rectangle will appear around it. Type or edit the name.



To copy a (sub)net into the current net, drag its tree view item onto the diagram canvas. Since nets may not directly or indirectly contain themselves, recursion is cut short.

[15] You may have to click three times.


Hierarchy
(unnamed)
subnet
subsubnet

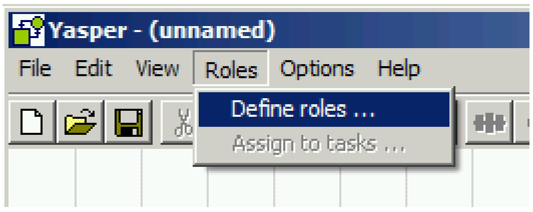

copy of subnet
Hierarchy
(unnamed)
subnet
copy of subnet
copy of subr
subsubnet
subsubnet

2.11 Working with roles

Roles provide a way to model resources required to execute a task (see section 1.3).

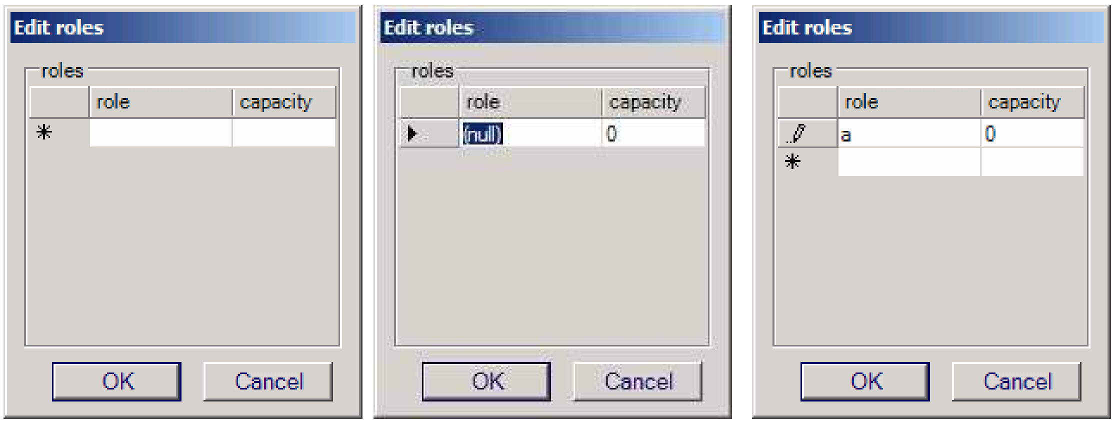
2.11.1 Defining roles

On the *Roles* menu, select *Define roles …*



The *Edit roles* dialog is shown.

To add a new role, click in the *role* column and type a name for it.



To delete a role, click on the leftmost, gray part of its row and press the *Delete* key.

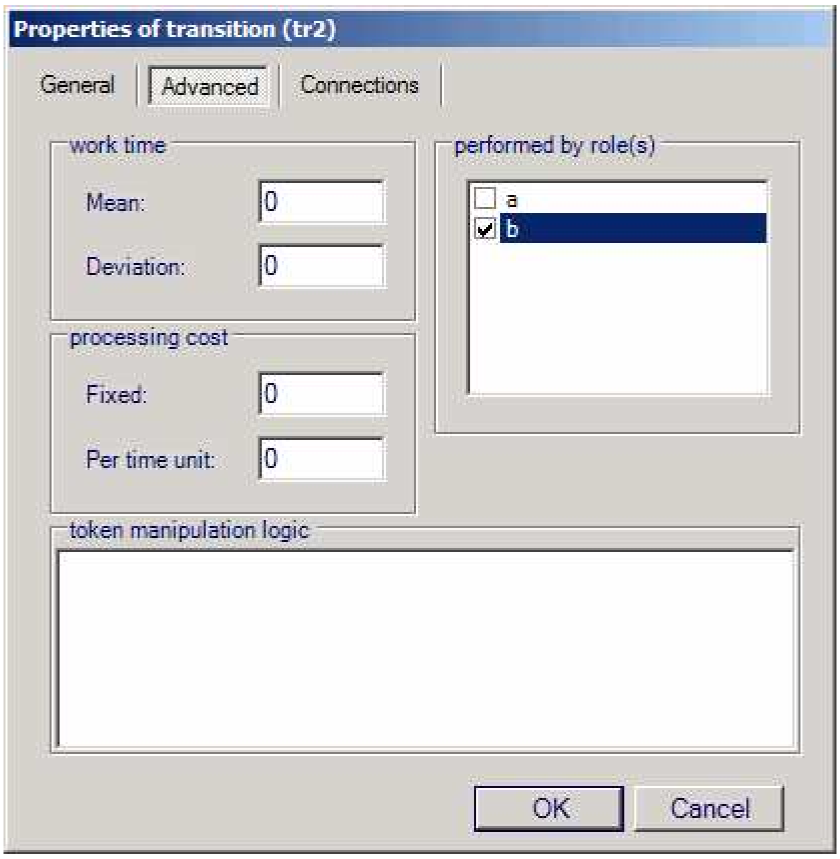
To set a role's capacity, type the number in the capacity column. A capacity must be a positive integer; *0* is invalid and will be automatically corrected to *1*.

Unlike the names of other elements, role names are mandatory and unique. Yasper automatically adjust the names entered to ensure this.

2.11.2 Assigning roles to transitions

Once one or more roles have been defined, they can be assigned to transitions and/or XORs.

To assign/deassign roles for an individual task (i.e., transition or XOR), double-click on it in the diagram, select its *Advanced* properties (see also section 2.9.1), and check the role(s) to be assigned.


Properties of transition (tr2)

General | Advanced | Connections

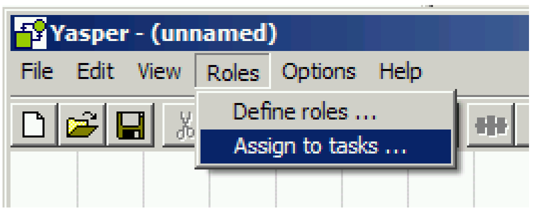
work time
Mean: 0
Deviation: 0

processing cost
Fixed: 0
Per time unit: 0

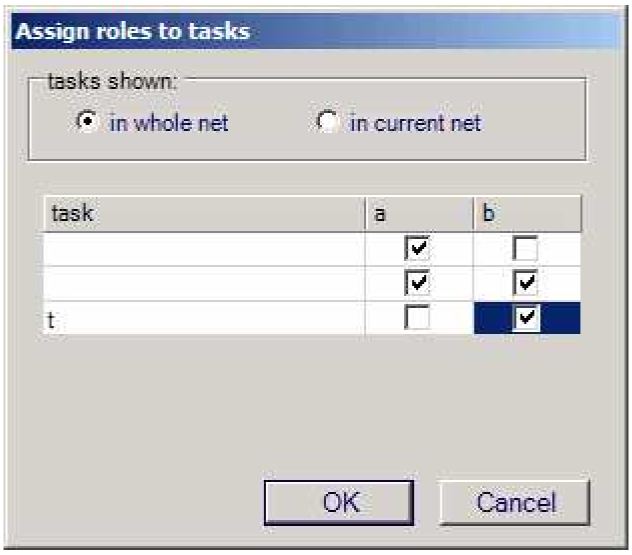
performed by role(s)
☐ a
☑ b

token manipulation logic

For an overview of all role assignments, select *Assign to tasks …* in the *Roles* menu.



The role assignment dialog is shown. It has a column for each role and a row for each task (i.e. transition or XOR).


Assign roles to tasks

tasks shown: [✓ in whole net] in current net

task	a	b
	✓	
	✓	✓
t		✓

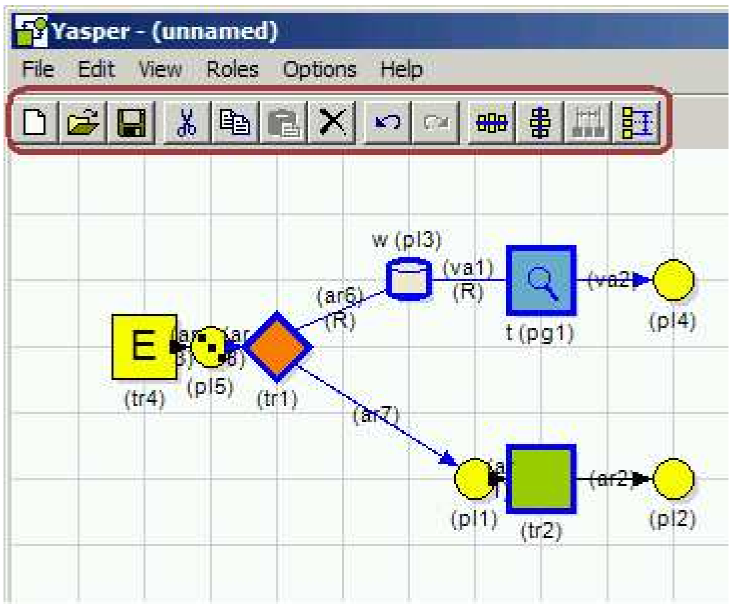
To assign a role to a task, check the relevant checkbox.
To deassign it, uncheck it.

To limit the rows to tasks in the (sub)net being displayed in the diagram, click on the *in current net* button.

If tasks have not been named, it may be practical to set *Show identifiers* in the *View* menu prior to bringing up this dialog.

2.12 Toolbar operations

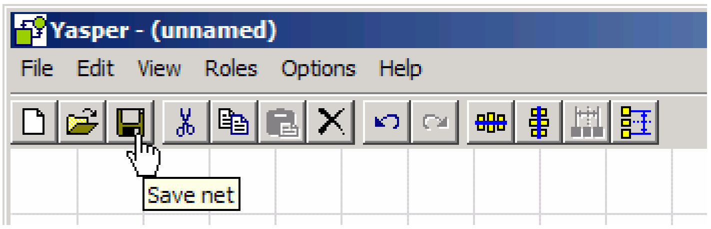
The toolbar below the main menu items provides one-click access to frequent operations. The toolbar items are just shortcuts: each item corresponds to a menu item in the *File* or *Edit* menu, explained in sections 2.14 and 2.13, respectively.



An item is grayed out if the operation cannot be performed at the present time.

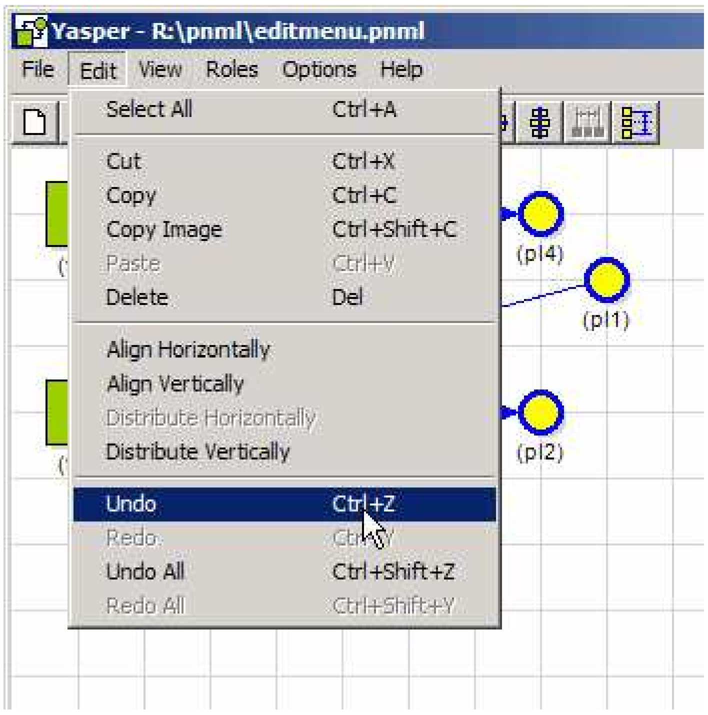
To execute an operation, click the item in question.

To get a textual description, put the mouse over an item and leave it still.



2.13 *Edit* menu operations

The *Edit* menu provides the standard operations and a few more.



2.13.1 Undoing changes

Yasper remembers all changes made in the editor since the last time the model was saved to file.

To undo a change, select *Undo* in the *Edit* menu, click the *Undo* toolbar icon, or press *Ctrl-Z.*
To redo a change, select *Redo* in the *Edit* menu, click the *Redo* toolbar icon, or press *Ctrl-Y.*

To *Undo* as many times as possible, the shortcut *Undo All* is available.
To *Redo* as many times as possible, *Redo All.*

2.13.2 Copying and pasting

To copy the elements selected in the diagram, select *Copy* in the *Edit* menu, click the *Copy* icon on the toolbar, or press *Ctrl-C.*

The copy is not always exact, since it is adjusted to form a valid model:

- if an arc is selected, the nodes it connects are in the copy, even when they were not selected;
- if a pin is selected, it is added to the copy as a place.

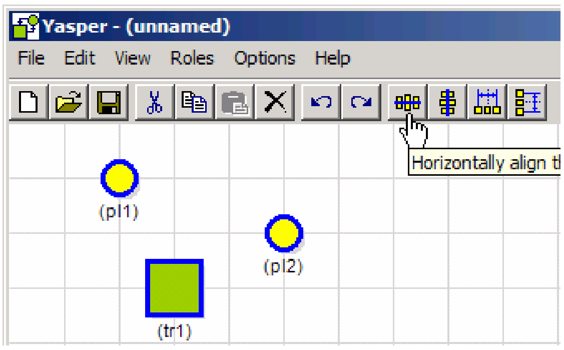
To paste a copy into the model, select *Paste* in the Edit menu, click the *Paste* icon on the toolbar, or press *Ctrl-V*. Pasting works in other (sub)nets and even in other Yasper instances, but not in other applications.

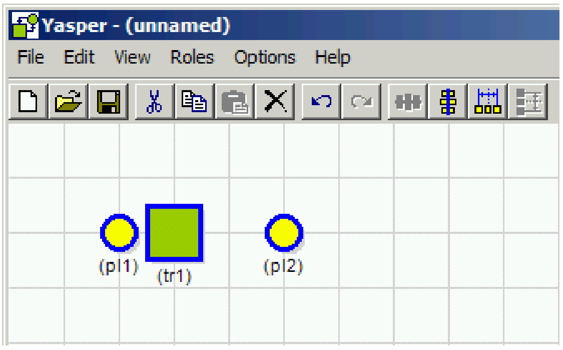
After pasting, the new elements are selected so they can be moved into position.

To copy an image of the current net that can be pasted into other applications,[16] use *Copy Image* on the *Edit* menu. The image will contain all elements of the current (sub)net, with the selected elements marked; gridlines are not shown.

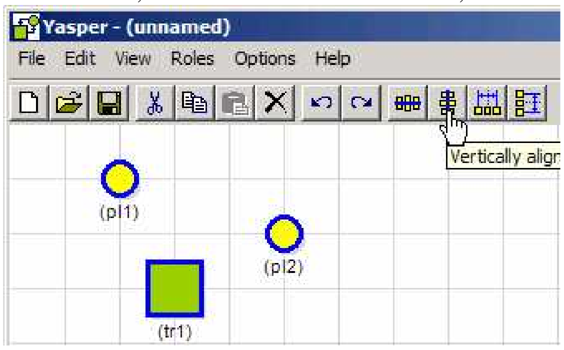
2.13.3 Aligning elements

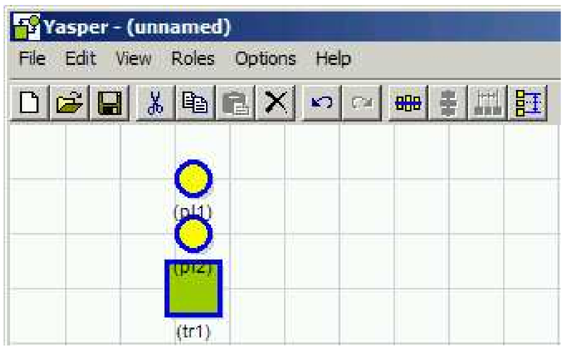
To shift the selected elements so they are on the same horizontal line, select *Align Horizontally* on the *Edit* menu, or click the toolbar icon, .





To shift the selected elements so they are on the same vertical line, select *Align Vertically* on the *Edit* menu, or click the toolbar icon, .





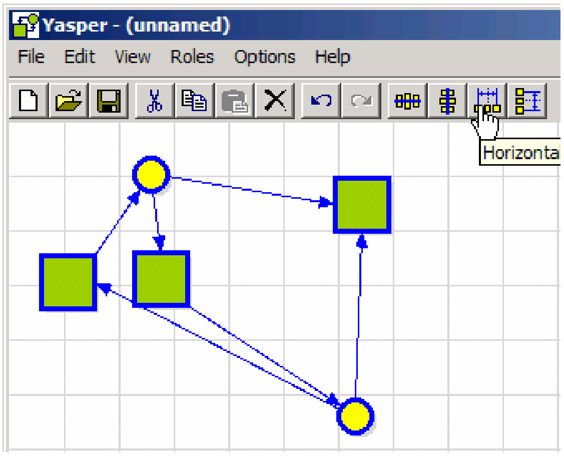
Arc support points (see section 2.7) are not affected by the operation.

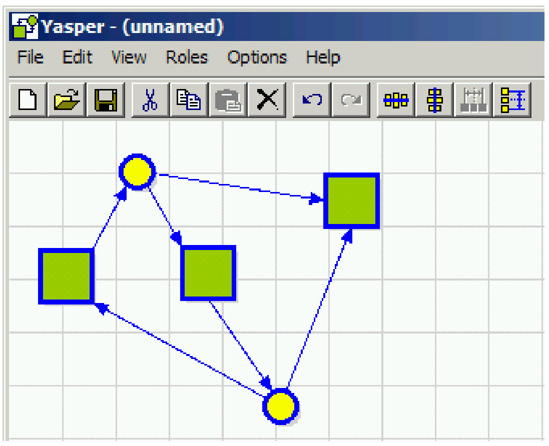
Another way to keep node alignment tidy is to set *Nodes snap to grid* in the *Options* menu (section 2.16).

[16] The Microsoft Office clipboard sometimes refuses to accept an image; when this happens, clear the clipboard before issuing *Copy Image*.

2.13.4 Making elements evenly spaced

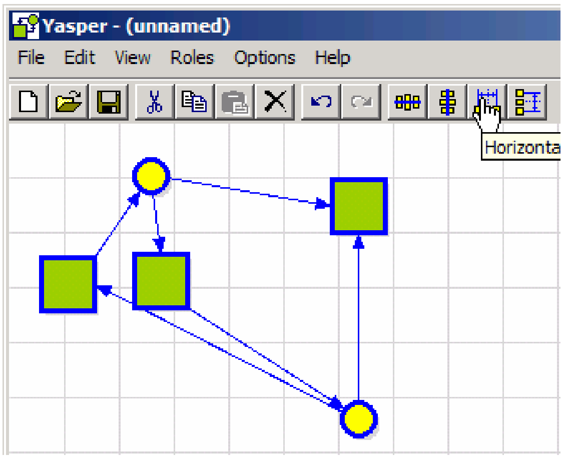
When elements are in a horizontal sequence, it often looks tidier to distribute them to be evenly spaced. To perform this operation on a set of elements, select *Distribute Horizontally* on the *Edit* menu, or click on the toolbar icon, .

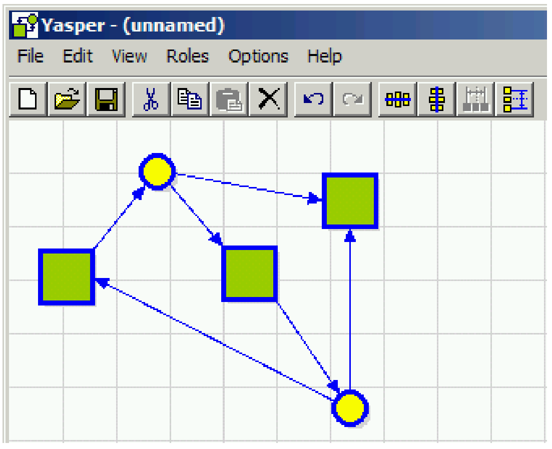




This works well to bring out the logical ordering between elements even when they are not actually horizontally aligned.

Nodes on the same height will remain on the same height:

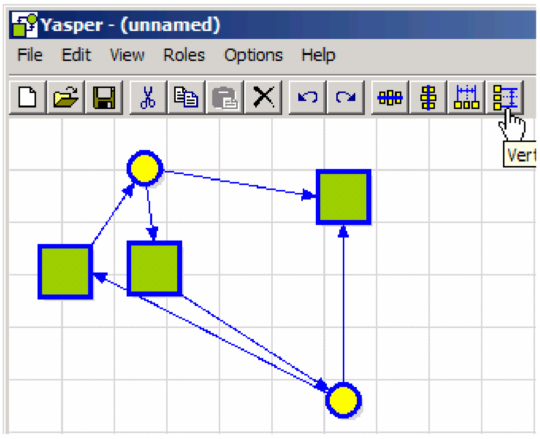




This operation completely ignores arcs: arcs do not affect how nodes are displaced, and arc support points stay where they are.

Horizontal distribution is orthogonal to horizontal alignment: distribution shifts nodes horizontally, while alignment shifts them vertically. They are often combined.

To apply the same operation vertically, select *Distribute Vertically* on the *Edit* menu, or click on the toolbar icon, .



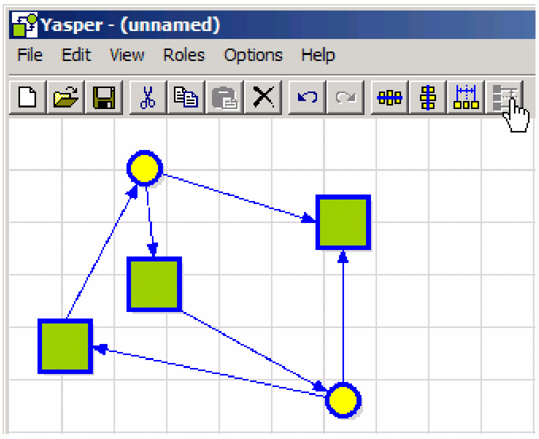
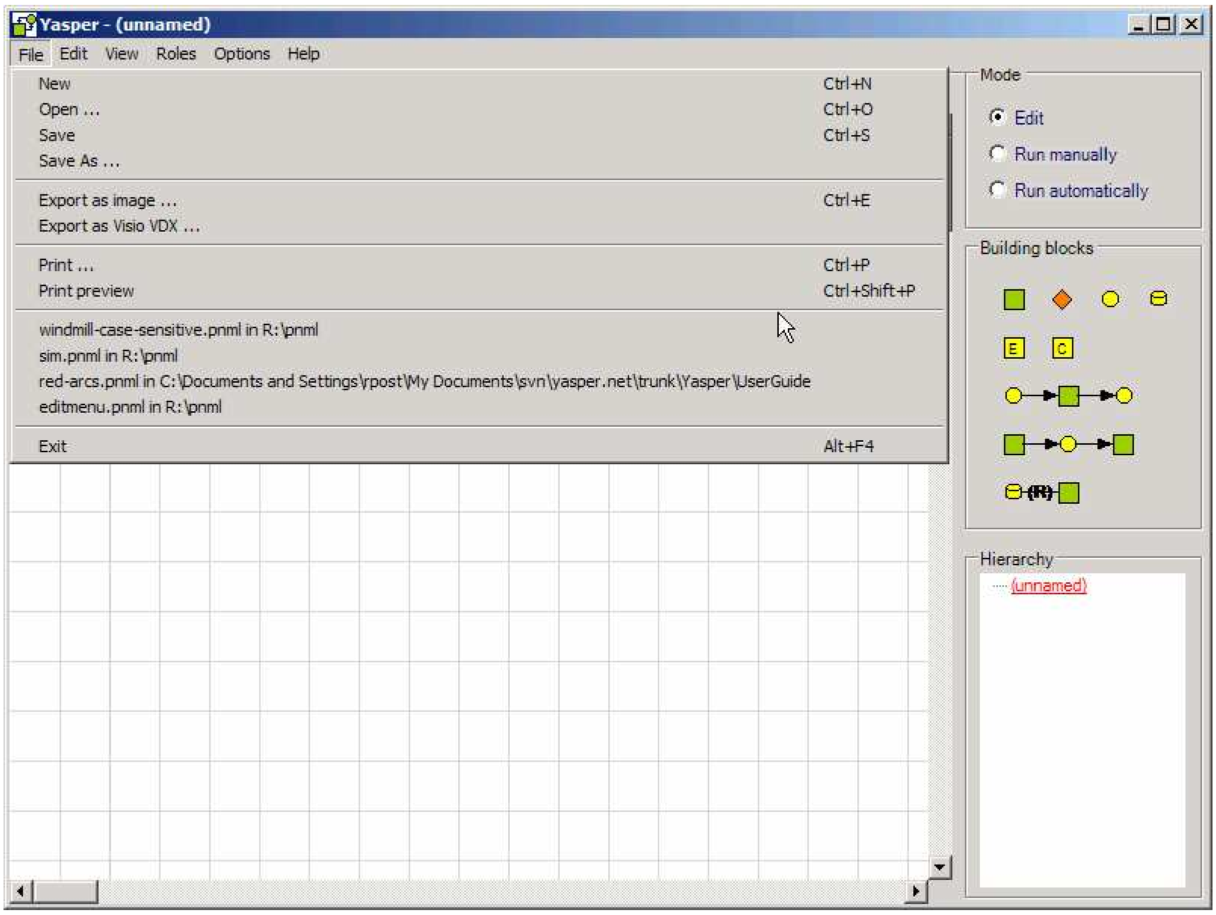


Diagram layouts can be tidied up efficiently and quickly by well-chosen combinations of horizontal and vertical alignment and distribution operations.

Another approach is to keep nodes on gridline crossings by setting the *Nodes snap to grid* option in the *Options* menu (section 2.16).

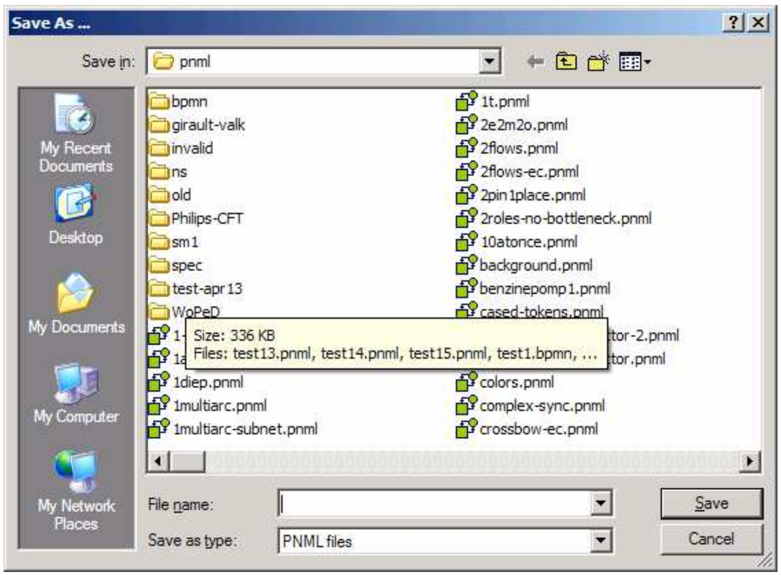
2.14 *File* menu operations

The Yasper *File* menu offers operations to load and save models, to export them to various file formats, to print images of models, and to exit Yasper.



2.14.1 Loading and saving Yasper models

To save a model to file and specify the filename, select *Save As …* on the *File* menu.
A dialog will ask for the filename.



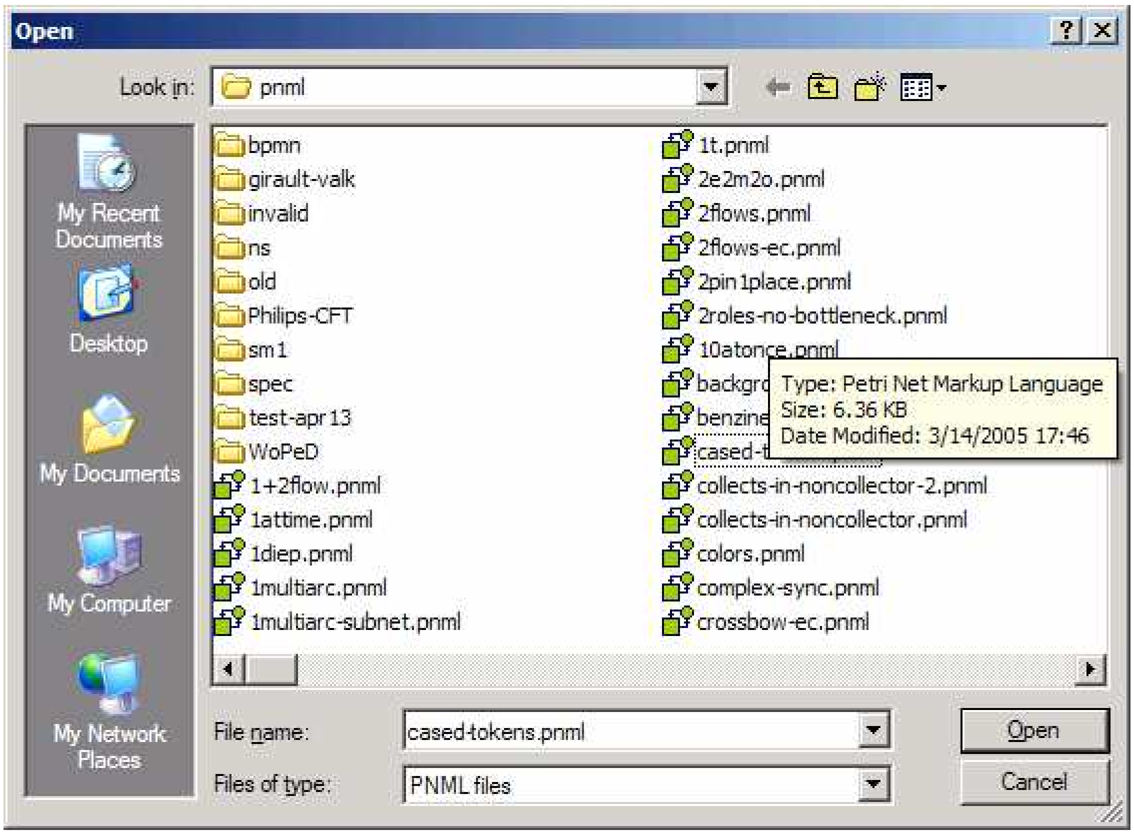
Size: 336 KB
Files: test13.pnml, test14.pnml, test15.pnml, test1.bpmn, ...

File name:
Save as type: PNML files

To save a model and use the filename with which it was opened or saved before, select *Save* on the *File* menu, click the *Save net* toolbar icon, or press *Ctrl-S*.
If Yasper doesn't know the filename to use yet, it shows the *Save As* dialog.

To start working on a new model, select *New* on the *File* menu or press *Ctrl-N*.
If the existing model has unsaved changes, Yasper will offer the opportunity to save it first.

To open an existing model, select *Open* on the *File* menu, click the *Open net* toolbar icon, or press *Ctrl-O*. If the existing model has unsaved changes, Yasper will offer the opportunity to save it first. Then, a dialog will ask for the file to open.



Type: Petri Net Markup Language
Size: 6.36 KB
Date Modified: 3/14/2005 17:46

File name: cased-tokens.pnml
Files of type: PNML files

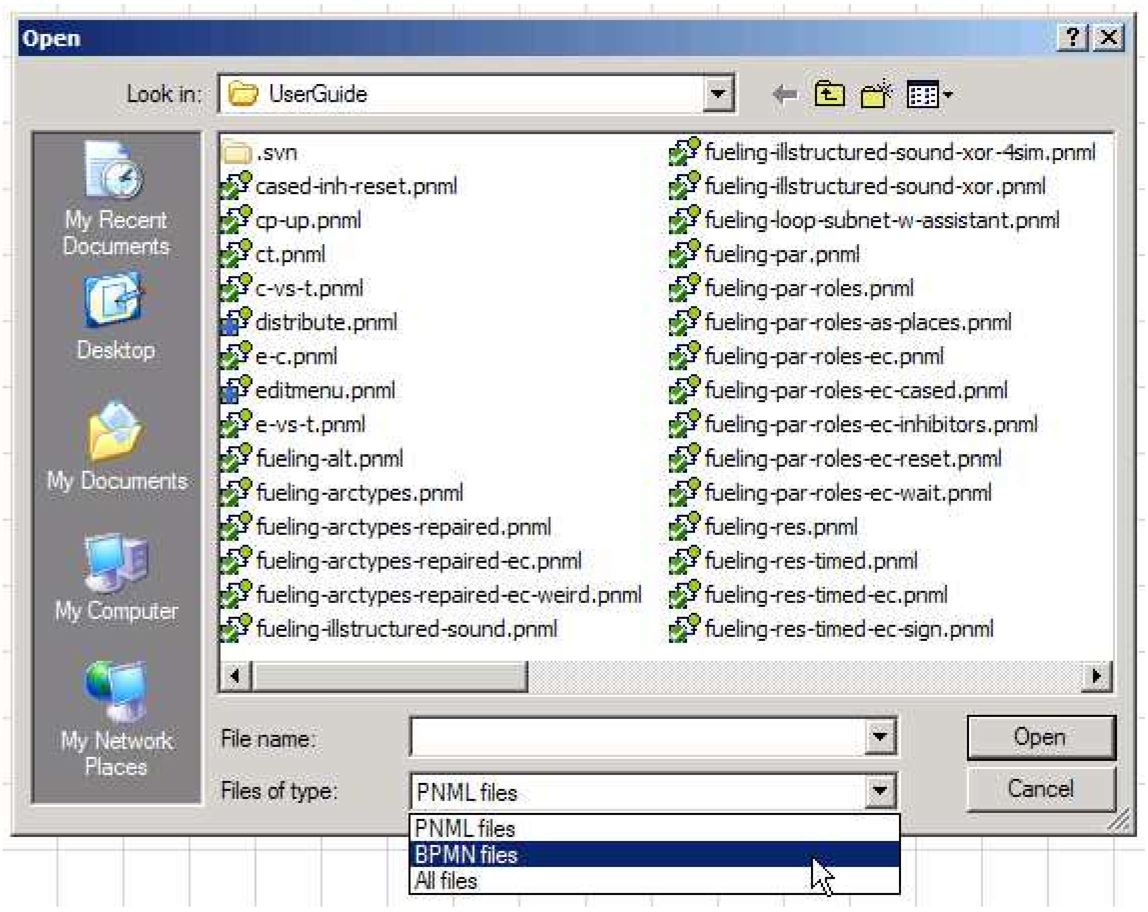
This dialog works in the standard way. Select a file and click *Open* to open it, or navigate to another directory using the various means the dialog provides. It is also possible to specify the fully qualified pathname or a URL of a PNML file in the File name selector.

Yasper saves its models in an XML-based file format conforming to the PNML standard[17], with the extension *.pnml*. It can obviously read every model produced with Yasper, but its use of PNML also provides limited interchangeability with other process modeling tools. In principle, all basic nets (section 1.1) and some additional features can be interchanged with any other tool that can read or write PNML. In practice, interchangeability is still limited, since tools disagree on how to interpret or employ the PNML standard. Therefore, Yasper may refuse to read or mis-interpret PNML produced by other tools, while other tools may do the same on PNML produced with Yasper.

2.14.2 Importing BPMN models

The *Open* dialog can also be used to import models in BPMN format. This format is used by a tool called Deloitte Industry Prints Process and Repository to create process models in the BPMN notation.[18]

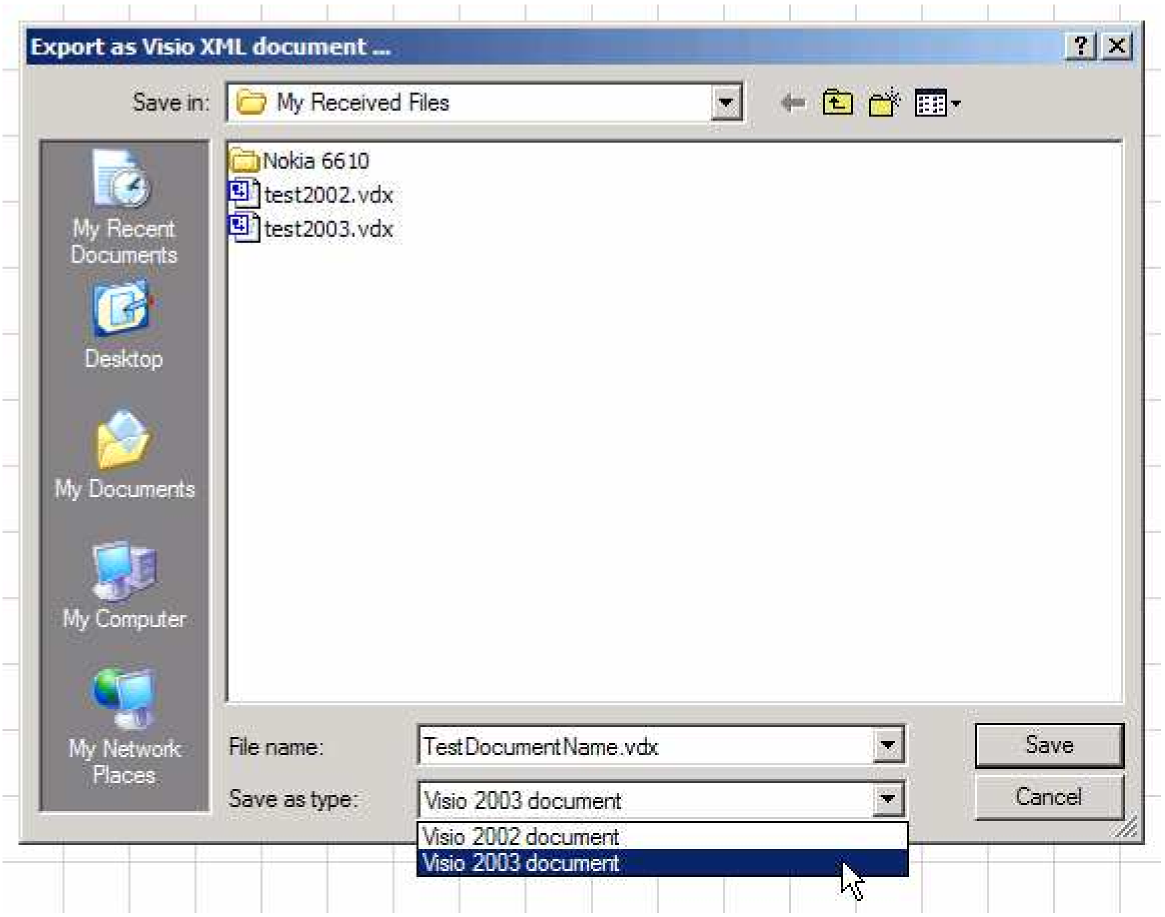
[17] For the PNML standard, see http://www.informatik.hu-berlin.de/top/pnml/;
for Yasper's use of it, see http://www.petriweb.org/specs/.
[18] For the BPMN standard, see http://www.bpmn.org/


Open
Look in: UserGuide
My Recent Documents
Desktop
My Documents
My Computer
My Network Places
.svn
cased-inh-reset.pnml
cp-up.pnml
ct.pnml
c-vs-t.pnml
distribute.pnml
e-c.pnml
editmenu.pnml
e-vs-t.pnml
fueling-alt.pnml
fueling-arctypes.pnml
fueling-arctypes-repaired.pnml
fueling-arctypes-repaired-ec.pnml
fueling-arctypes-repaired-ec-weird.pnml
fueling-illstructured-sound.pnml
fueling-illstructured-sound-xor-4sim.pnml
fueling-illstructured-sound-xor.pnml
fueling-loop-subnet-w-assistant.pnml
fueling-par.pnml
fueling-par-roles.pnml
fueling-par-roles-as-places.pnml
fueling-par-roles-ec.pnml
fueling-par-roles-ec-cased.pnml
fueling-par-roles-ec-inhibitors.pnml
fueling-par-roles-ec-reset.pnml
fueling-par-roles-ec-wait.pnml
fueling-res.pnml
fueling-res-timed.pnml
fueling-res-timed-ec.pnml
fueling-res-timed-ec-sign.pnml
File name:
Files of type: PNML files
PNML files
BPMN files
All files
Open
Cancel

To import a BPMN model in this format, use the *Open* function, but in the *Files of type:* selector, select *BPMN files*. Instead of PNML files, the list of BPMN files in the same directory will be shown. Pick one and click the *Open* button, or navigate to another directory with the methods provided by the dialog.

2.14.3 Exporting models to Microsoft Visio

To export a model to Microsoft Visio format, select *Export as Visio VDX …* on the File menu.



File name: TestDocumentName.vdx

Save as type: Visio 2003 document

Visio 2002 document

Visio 2003 document

The dialog offers two options:

- export to Visio 2002 VDX format
- export to Visio 2003 VDX format

Although the extensions are the same, the two document formats are incompatible.
Visio 2002 and 2003 can partly read each other's VDX documents, but in doing so, throw warnings and lose information. Therefore it is important to select the correct format when exporting.

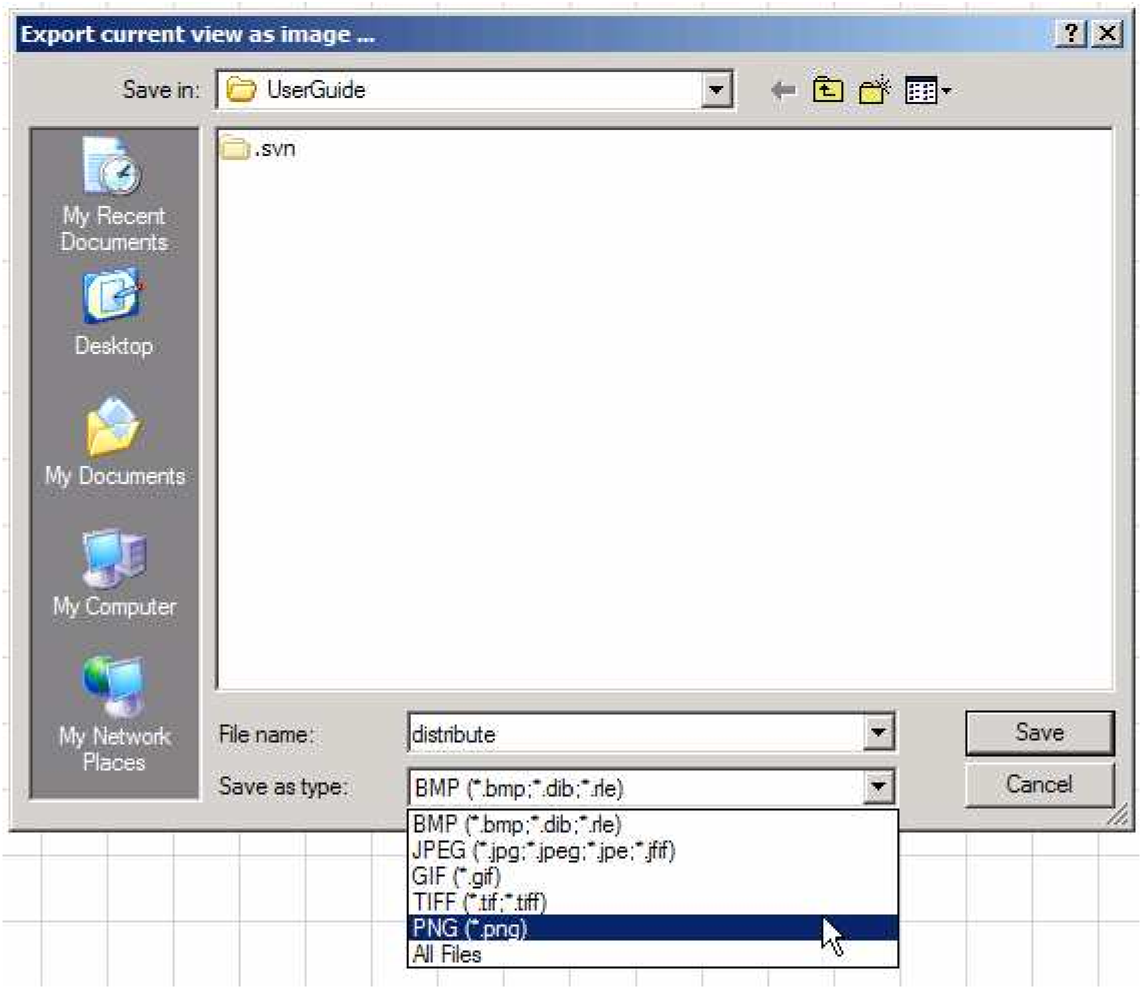
The translation provides a fairly accurate Visio representation of the Yasper model. If the document has subnets, they will be on separate Visio document pages, and the page symbol can be double-clicked to go to the page content.

The resulting Visio diagram is not intended as a replacement of the Yasper model:

- some information, e.g. on roles, is not included;
- there is no support for making sure that subsequent edits of the Visio document will leave it a valid representation of a Yasper model, and
- there is no support for importing Visio diagrams back into Yasper.

2.14.4 Exporting diagram images

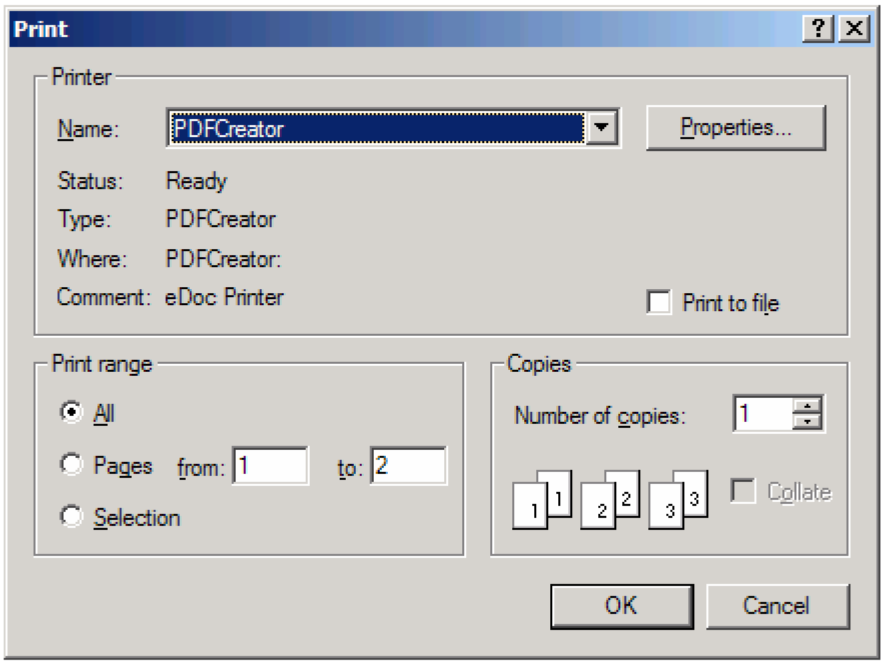
To export an image of the current (sub)net to file, *select Export as image …* in the File menu. Various bitmap image formats are supported.



The image exported is the same as that copied by *Copy Image* (see section 2.13.2), except that no elements are selected.

2.14.5 Printing diagrams

To print a Yasper model, select *Print* in the *File* menu, click the *Print* icon in the toolbar, or press Ctrl-P. The print dialog will appear.



The images are identical to those produced by *Export as image* (see previous section), except that they are automatically scaled to fill the printed page.

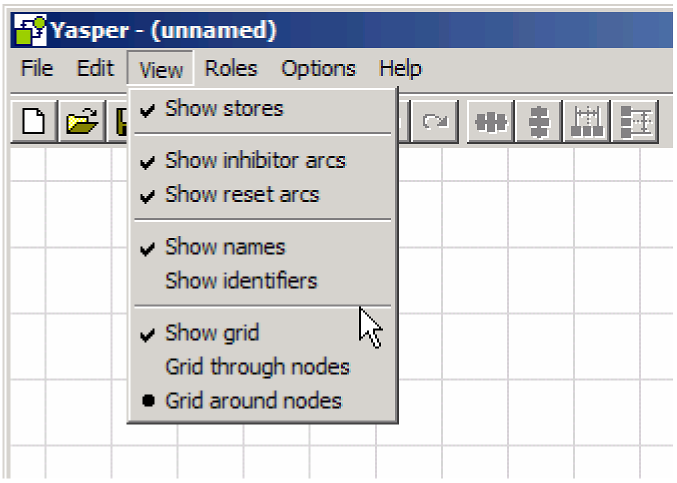
The *Print range* determines what to print:

- select *All* to print the main net and all subnets, each on a separate page;
- specify a page range to limit which of the nets are printed;
- select *Selection* to print the net that a *Copy* operation would copy (see section 2.13.2).

For a closer look at what is going to be printed, use *Print preview* on the *File* menu (*Shift-Ctrl-P*).

2.15 *View* menu options

Use the *View* menu to configure how the model is displayed.



Settings are remembered; future Yasper sessions will use them.[19]

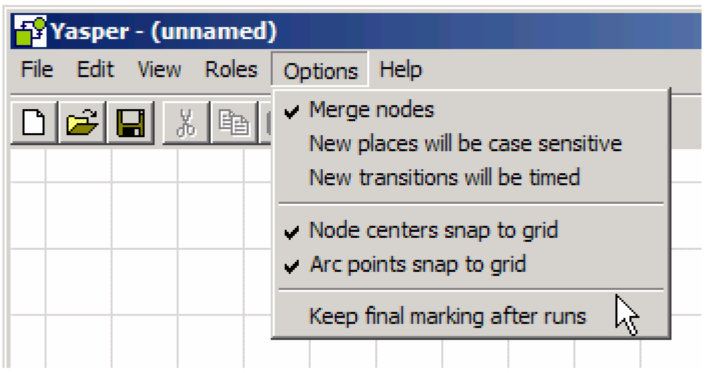
- When *Show stores* is off, data stores (cf. section 1.8) are omitted from the diagram. (They remain present in the model.)
- When *Show inhibitor arcs* is off, inhibitor arcs (cf. section 1.5.2) are omitted from the diagram. (They remain present in the model.)
- When *Show reset arcs* is off, reset arcs (cf. section 1.5.3) are omitted from the diagram. (They remain present in the model.)
- *Show names* displays the user-assigned names on elements. Note that such a name does not always exist and does not have to be unique.
- *Show identifiers* displays the system-assigned identifiers on elements. An identifier always exists and is always unique. Like *Show names*, this setting not only affects the diagram, but also the appearance of elements in dialogs and reports; it is convenient to set this when explicit names have not yet been assigned.
- *Show grid* displays a grid on the diagram surface; this is mainly useful for horizontal and vertical alignment, e.g. with the *snap to grid* feature (cf. section 2.16)
- *Grid through nodes* makes *snap to grid* snap elements to grid crossings;
- *Grid around nodes* makes *snap to grid* snap elements to grid tile centers.[20]

Most settings only affect the diagram; the *Show names* and *Show identifiers* settings also affect how elements appear in the various dialogs used during editing, and in the automatic simulation report. Names are assigned by the user, optional, and not always unique, while identifiers are machine-assigned, required and unique.

[19] However, if multiple Yasper sessions are running at the same time, changing a setting in one will not affect the other.
[20] *Grid around nodes* is the layout style used for most board games; Go players will prefer the *Grid through nodes* layout.

2.16 *Options* menu options

The user preferences in the *Options* menu affect the way editor behaves.
Like *View* settings, they are remembered for future Yasper sessions.


Yasper - (unnamed)
File Edit View Roles Options Help
✓ Merge nodes
New places will be case sensitive
New transitions will be timed
✓ Node centers snap to grid
✓ Arc points snap to grid
Keep final marking after runs

- *Merge nodes* makes a node melt away into a compatible node when moved onto it; for details see section 2.4.
- *New places will be case sensitive* determines whether places dragged from the building blocks (see section 2.1) are case sensitive.
- *New transitions will be timed* determines whether transitions dragged from the building blocks will be timed.
- *Node centers snap to grid* determines whether after moving selected elements, node centers will automatically snap to the gridline crossings (see section 2.3).
- *Arc points snap to grid* determines whether after moving selected arcs, their support points (see section 2.7) will snap to the grid lines.
- *Keep final marking after runs* determines what happens when switching from manual or automatic run mode back to editing mode. If this setting is disabled, editing will continue with the net as it was before simulation started; if it is enabled, it will continue with the token marking that was in effect in simulation mode.[21]

[21] Except that if tokens were being processed by transitions, they will be forcibly produced, since tokens cannot reside in transitions during editing.

3 Manual simulation

3.1 What is simulation?

As explained in section 1.1.2, we know exactly what it means for a process described in Yasper to execute, since it is precisely defined (in section 1.6.4.4) when a process execution step is possible, what happens when it starts, and what happens when it finishes. Therefore, at any time it is unambiguously determined what may happen.

It is not always unambiguously determined what *will* happen; many choices remain open:

- multiple transition executions may be possible at the same time;
- for timed transitions, processing time may vary;
- for transitions with roles, multiple roles may be available;
- for choice elements (XORs), multiple inputs and/or outputs may be available.

Therefore it can be very instructive to see execution happen in a process model. This is known as *simulation*, since it simulates execution of the real process being modeled.

Yasper offers two methods of simulation:

- manual run: the user picks most of the choices by hand
- automated run: the system picks all of the choices at random

This chapter describes the facilities for manual simulation;
automated simulation is described in chapter 4.

In manual simulation, the user executes the process step by step, making choices along the way. This is useful for showing that certain situations can be reached – for example, deadlocks. For quantitative analysis of a process model, use automatic simulation.

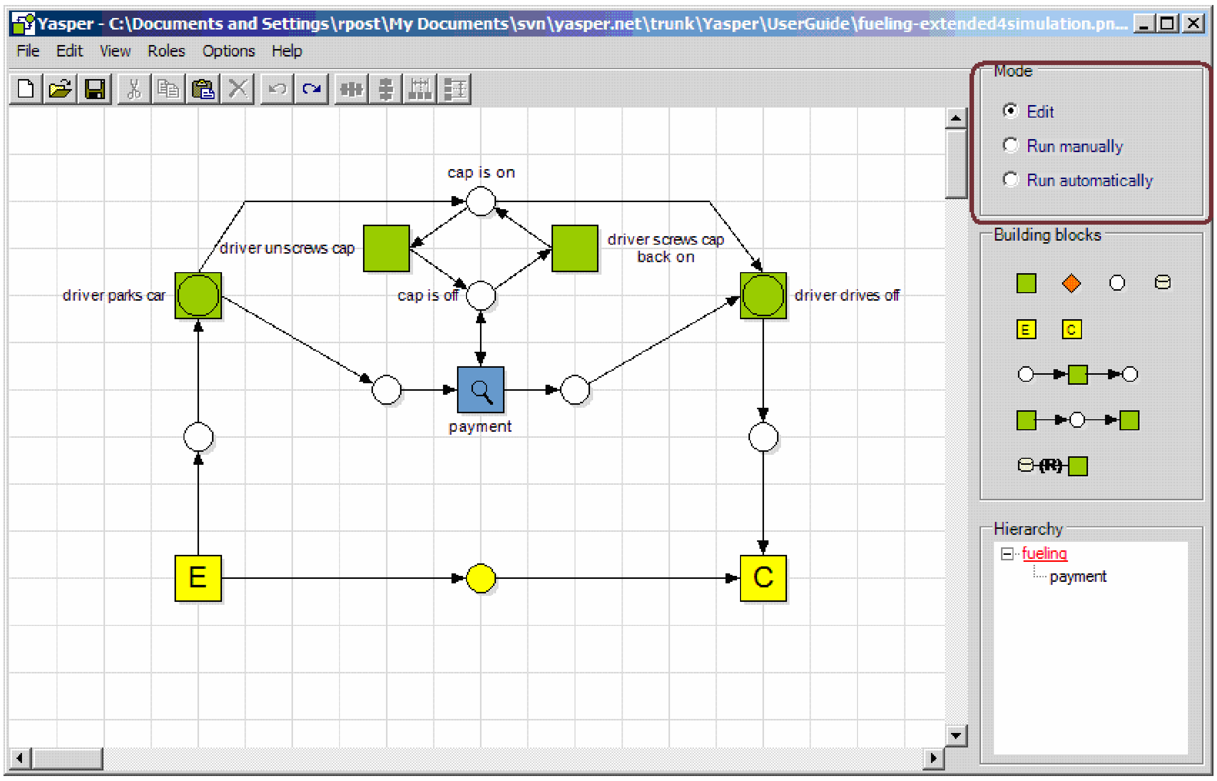
3.2 Preparing for simulation (adding tokens)

A model does not always represent the particular initial situation in which a run should start. For manual simulation, which does not employ time, such preparation mainly consists of choosing the right initial distribution of tokens. Models with initial places, such as those given in section 1.1 to 1.5, require that tokens be added to the initial places. Models with places that represent resources can be configured by setting various numbers of tokens in such places.

To perform such changes, go to *Edit* mode; in manual or automatic run simulation mode, the only way to modify the token marking is by executing transitions.

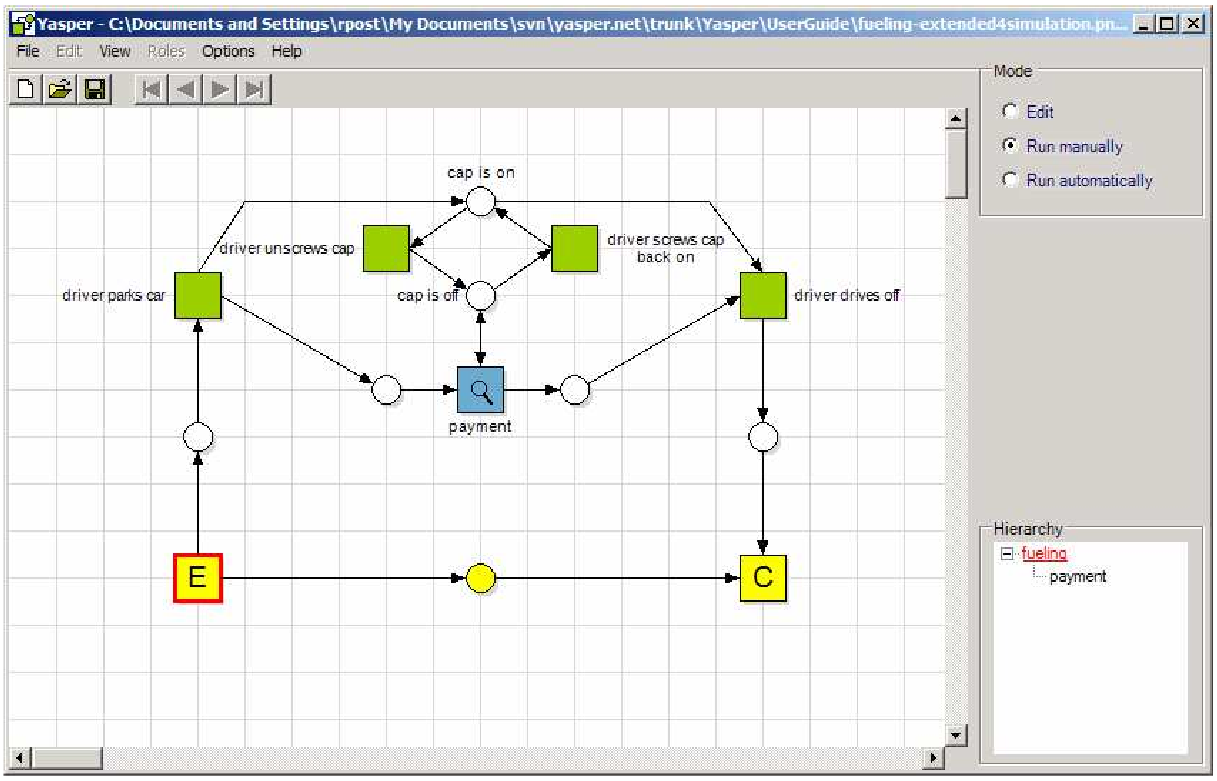
3.3 Starting manual simulation

In the right upper hand corner, three radio buttons control the mode Yasper is in.



To switch to manual simulation mode, click *Run manually*.[22]

[22] This example describes the same process as the example in section 1.2; it was derived from it by adding an emitor-collector path, making some transitions timed, moving a part into a subnet, and changing the layout.


Yasper - C:\Documents and Settings\rpost\My Documents\svn\yasper.net\trunk\Yasper\UserGuide\fueling-extended4simulation.pn...
File Edit View Roles Options Help
Mode
Edit
Run manually
Run automatically
cap is on
driver unscrews cap
driver screws cap back on
driver parks car
cap is off
driver drives off
payment
E
C
Hierarchy
fueling
payment

The display changes in various ways:

- there is no time in manual simulations, so timed transitions appear timeless;
- the elements that can execute are indicated by a red border;
- the toolbar gets four buttons, used in replaying the simulation.

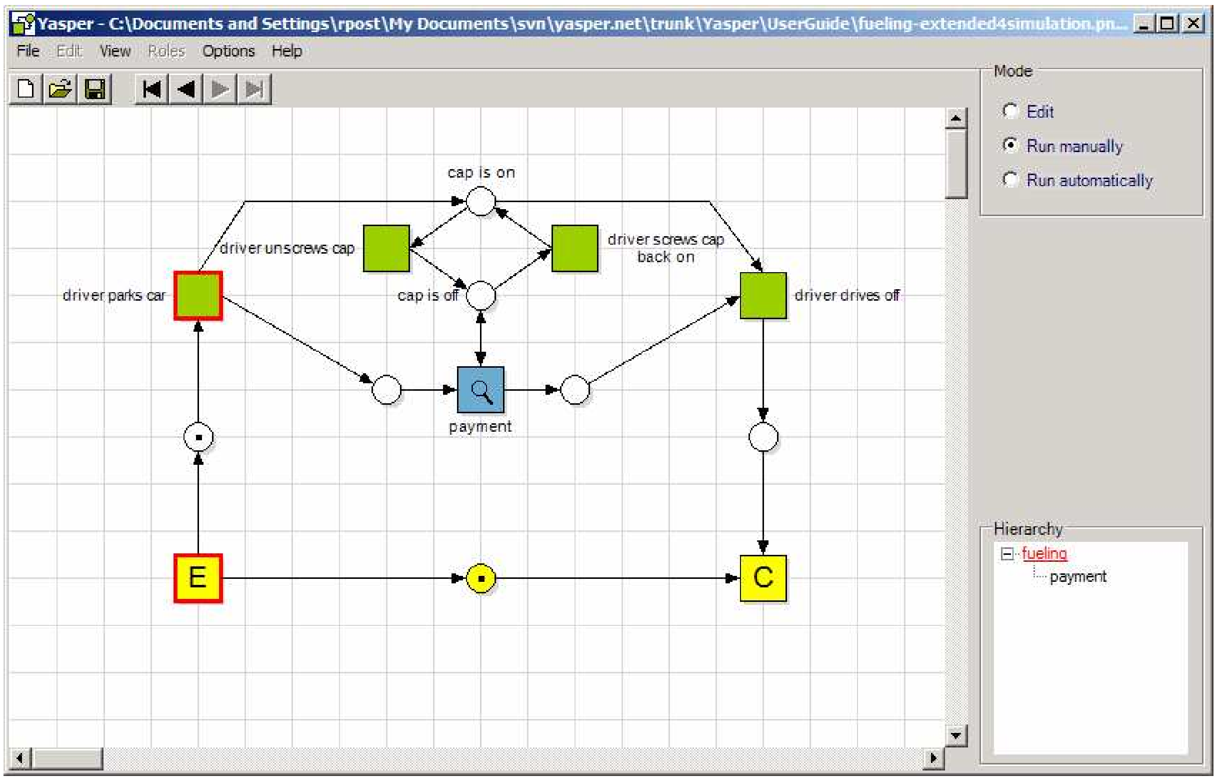
Editing operations become unavailable.

To switch back to edit mode, click *Edit mode*.

Some *File* operations can be invoked in manual simulation mode, but have the effect of switching to edit mode, since they do not make sense unless performed in edit mode.

3.4 Executing a manual simulation

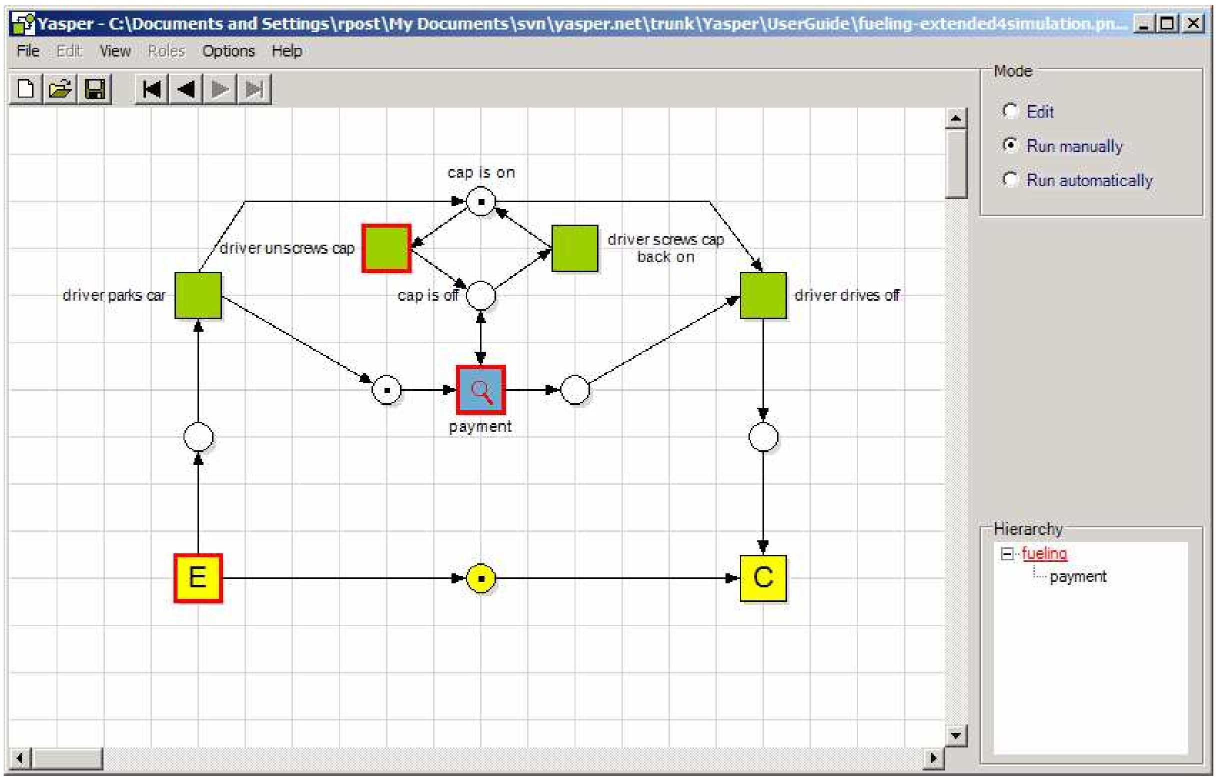
To execute a step, click on an element that is marked executable. In the above picture, that can only be the emitor; clicking it produces


Yasper - C:\Documents and Settings\rpost\My Documents\svn\yasper.net\trunk\Yasper\UserGuide\fueling-extended4simulation.pn...

Mode: ( ) Edit [✓ Run manually] ( ) Run automatically

Repeat to continue. When multiple items are executable, select any of them.

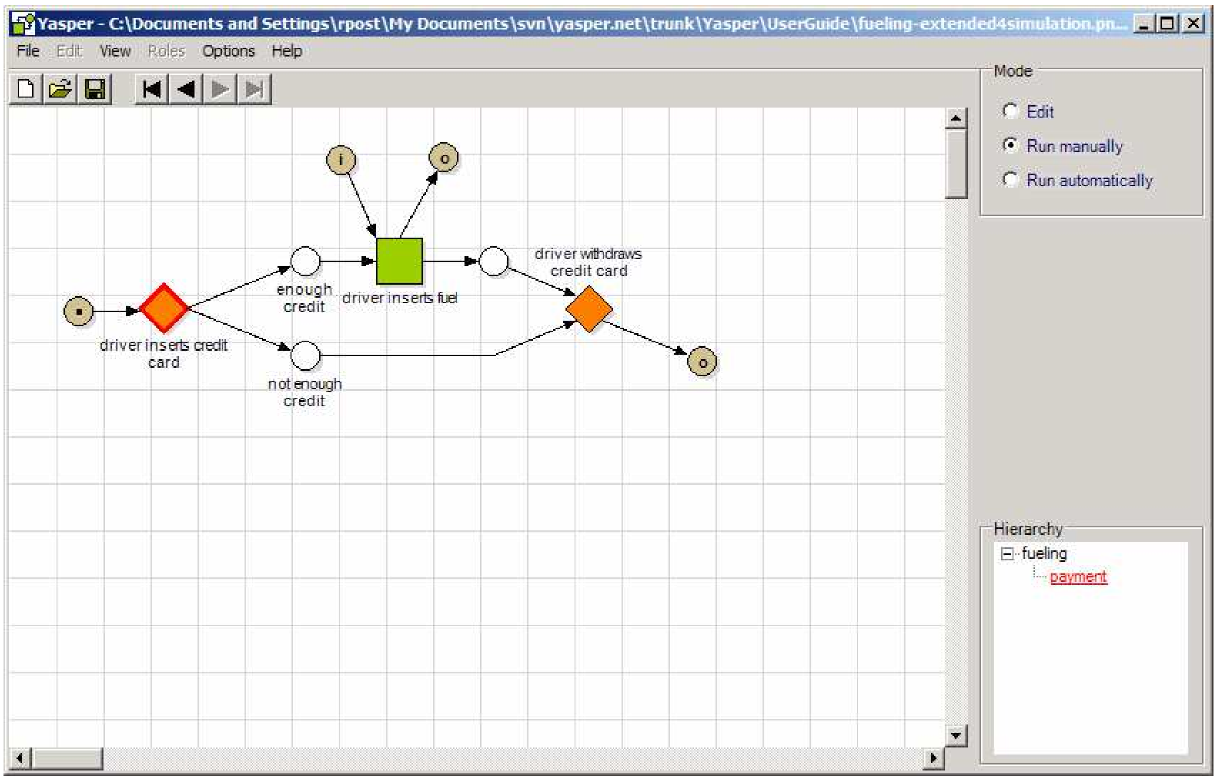
Let’s assume *driver parks car* is clicked.


Yasper - C:\Documents and Settings\rpost\My Documents\svn\yasper.net\trunk\Yasper\UserGuide\fueling-extended4simulation.pn...

Mode: Edit [✓ Run manually] Run automatically

Hierarchy: fueling, payment

A subnet is marked executable whenever any of its elements are. Click on the subnet to move inside.

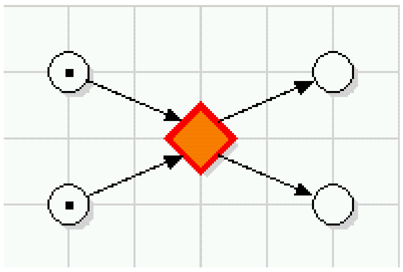

Yasper - C:\Documents and Settings\rpost\My Documents\svn\yasper.net\trunk\Yasper\UserGuide\fueling-extended4simulation.pn...

Mode: Edit [✓ Run manually] Run automatically

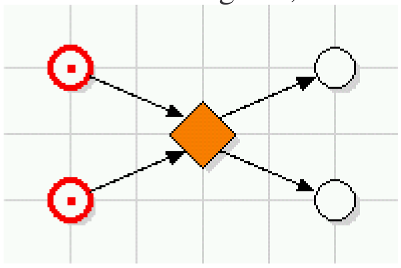
Hierarchy: fueling, payment

To move up into the surrounding net, click on a pin.

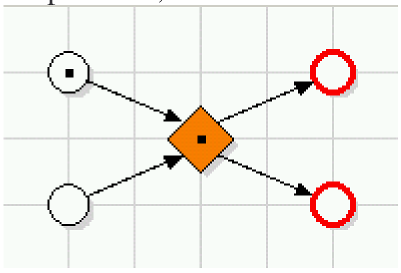
Executing a XOR is usually a multi-step process.



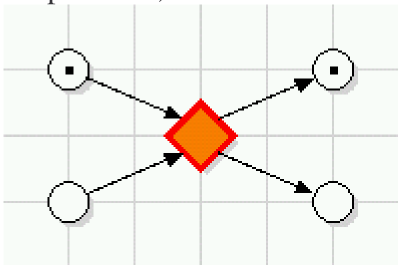
To start executing one, click it.



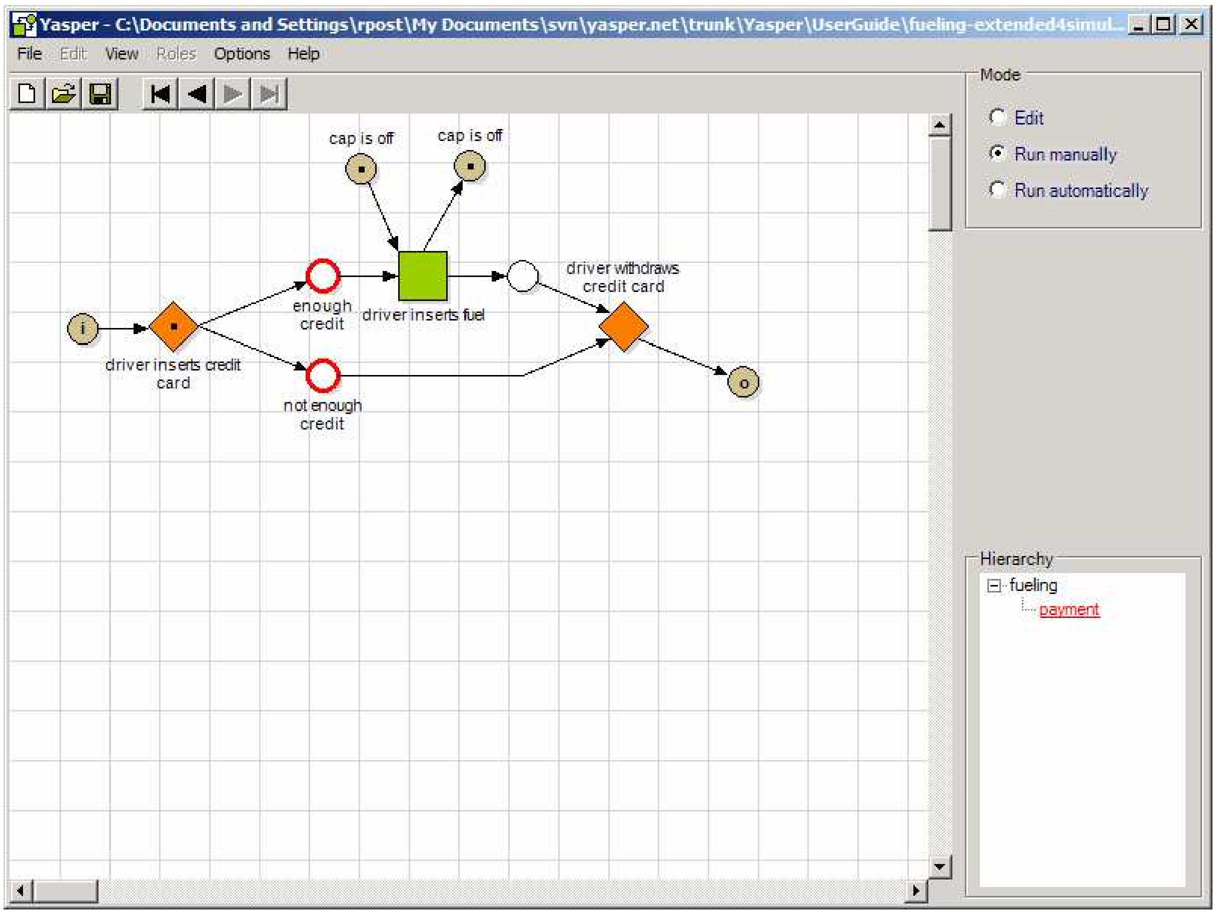
If input can be taken from multiple places, they will be marked with red borders.
To pick one, click it.



If output can be put into multiple places, they will be marked with red borders.
To pick one, click it.



If there aren't multiple inputs or outputs to choose from, the respective step is omitted. For instance, clicking the XOR in the example above, which only has one input, directly produces


Yasper - C:\Documents and Settings\rpost\My Documents\svn\yasper.net\trunk\Yasper\UserGuide\fueling-extended4simul...
File
Edit
View
Roles
Options
Help
cap is off
cap is off
i
driver inserts credit card
enough credit
not enough credit
driver inserts fuel
driver withdraws credit card
o
Mode
Edit
Run manually
Run automatically
Hierarchy
fueling
payment

3.5 Reviewing a simulation

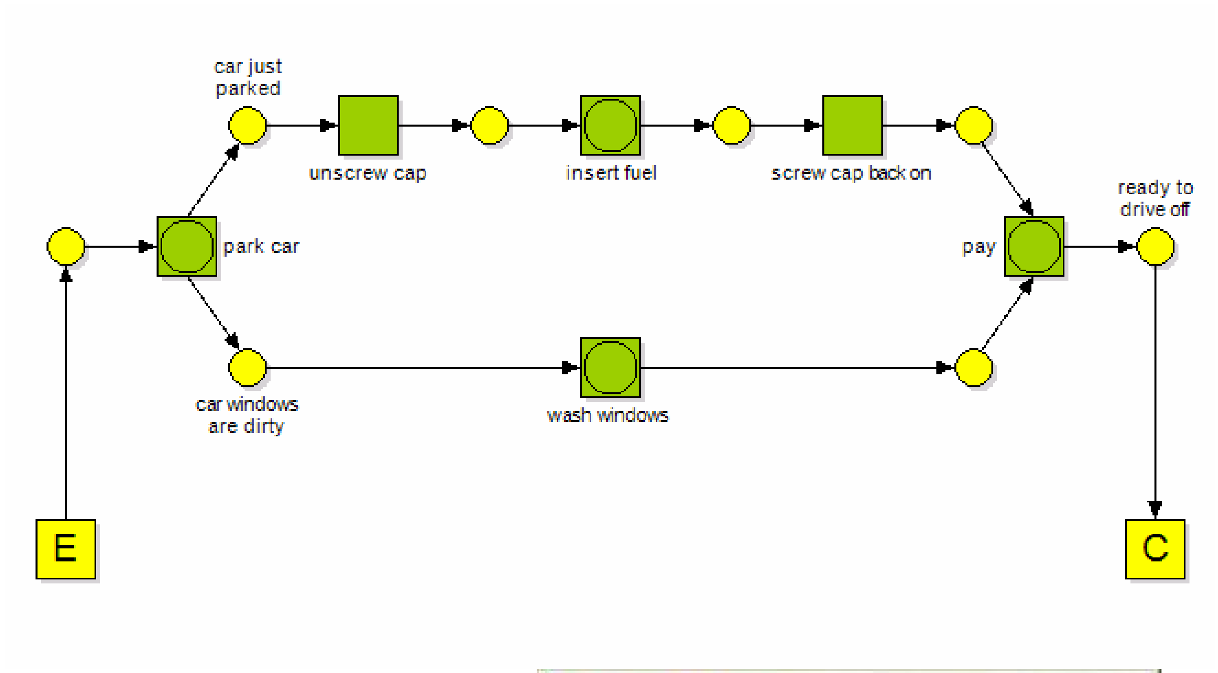
Four controls in the toolbar appear in manual simulation mode:

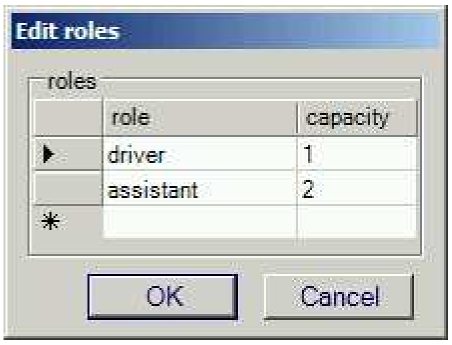
⏮	Undo all steps – back to the start of the run
◀	Undo the last step
▶	Redo the last undone step
⏭	Redo all redoable steps – forward to after the last step taken

The buttons are grayed out when the function in question is not available. For example, right after taking a step, the first two are always enabled, since the step can be undone, while the other two are disabled, since there is nothing to redo.

3.6 Manual simulation with roles

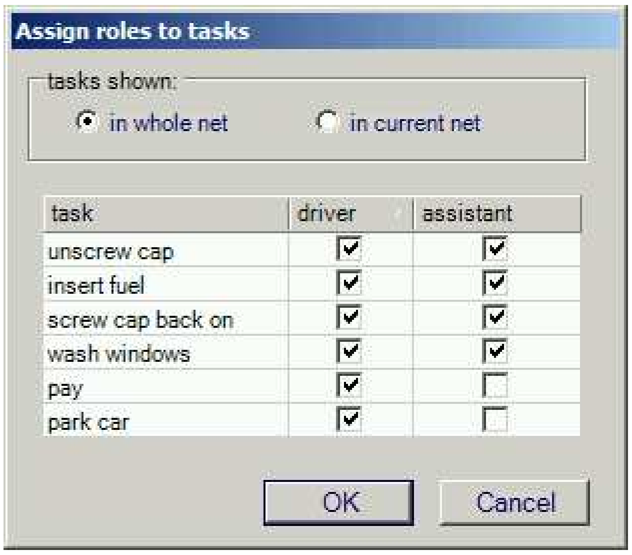
When roles have been assigned to tasks (see section 2.11.2), manual simulation offers the choice which role to use when performing a task. Reconsider the first example of section 1.6.4.4, with roles assigned as in section 1.3:




Edit roles

roles

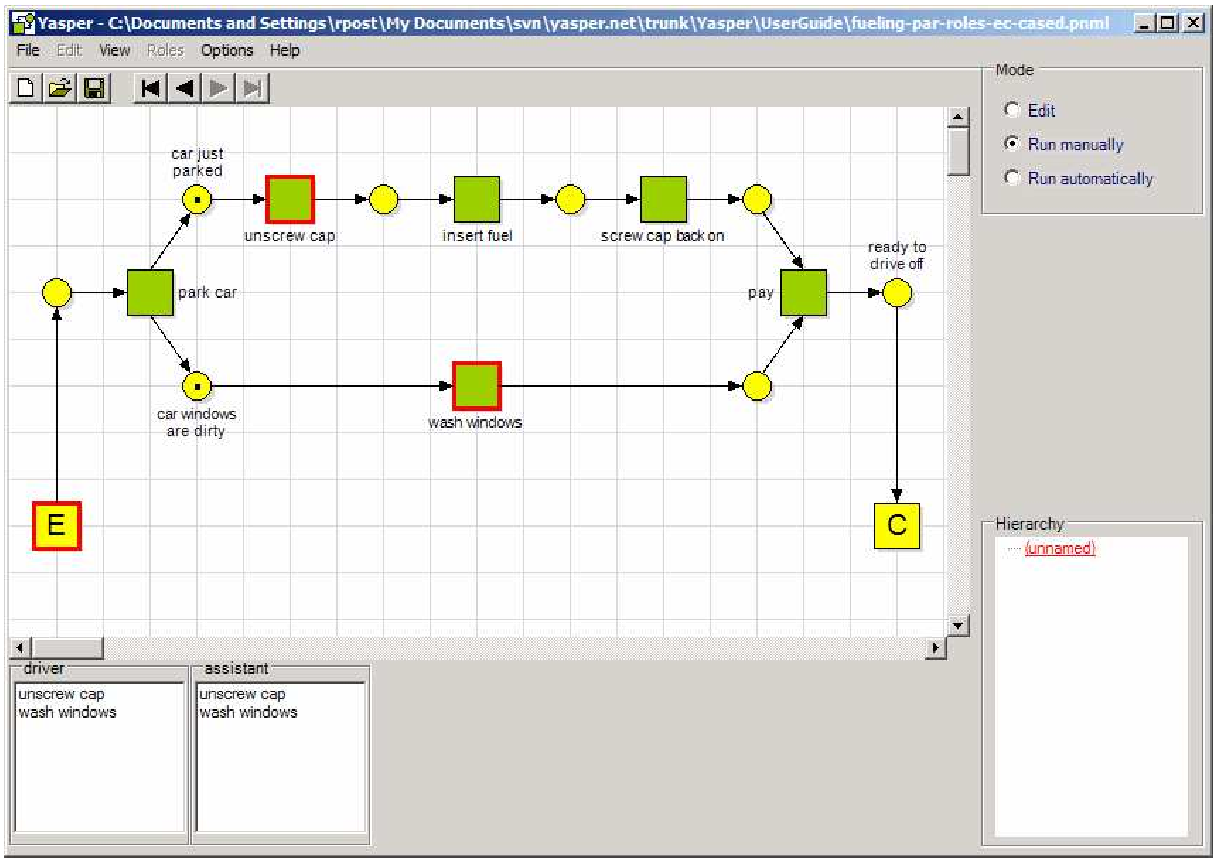
	role	capacity
▶	driver	1
	assistant	2
✳		


Assign roles to tasks

tasks shown: [✓ in whole net] in current net

task	driver	assistant
unscrew cap	✓	✓
insert fuel	✓	✓
screw cap back on	✓	✓
wash windows	✓	✓
pay	✓	
park car	✓	

In manual simulation mode, the interface displays a box for every role. At any time the box lists the transitions and XORs that can execute with that role:



Mode
Edit [✓ Run manually] Run automatically

driver
unscrew cap
wash windows

assistant
unscrew cap
wash windows

To execute a task with any available role, click on it in the diagram.
To execute a task with a specific role, click on the task in the box for that role.

4 Automatic simulation

As discussed in section 3.1, Yasper provides the user with two methods to execute processes in a process model:

- manually (see chapter 3)
- automatically (this chapter)

In an automatic simulation, cases for the process are generated and executed automatically. Statistics on execution times and resource utilization are aggregated and summarized in a report.

While in manual runs, choices on how to continue are made buy the user, during automatic runs all choices are made randomly, depending on configuration parameters set by the user beforehand.

Automatic simulation serves two main purposes. First, many logical modeling errors are caught when automatic simulation refuses to run or produces unexpected results. Second, once a correct process model has been constructed, simulation can be used to estimate performance and efficiency.

This chapter explains

- the general principles of automatic simulation (section 4.1)
- how to prepare for a run (sections 4.2 and 4.3)
- how to invoke and control a run (section 4.2)
- what the figures in the simulation report mean (section 4.5)
- switching views during simulation (section 4.6)

4.1 How automatic simulation works

Automatic simulation assumes *workflows*: process models that have the net effect of taking individual cases from a well-defined starting point to a well-defined end point. Before automatic simulation can be performed, the starting points, end points and intermediate points must be explicitly marked in the model with *emitors*, *collectors* and *case sensitive places*, respectively – section 1.6.4 provides full details and examples.

The simulator generates workflow cases in emitors, runs them through the process model, collecting statistics along the way, and summarizes them in a report. Some of the reported statistics pertain only to cases that have *completed* in collectors (see section 1.6.4.5).

Several user-configurable parameters influence the behavior of automatic simulation runs:

- *processing time* for transitions (see section 2.9.1 on how to set it)
- *output weights* on XORs (see section 2.9.2)
- *role capacities* and *role allocations* for either (see section 2.9.1)

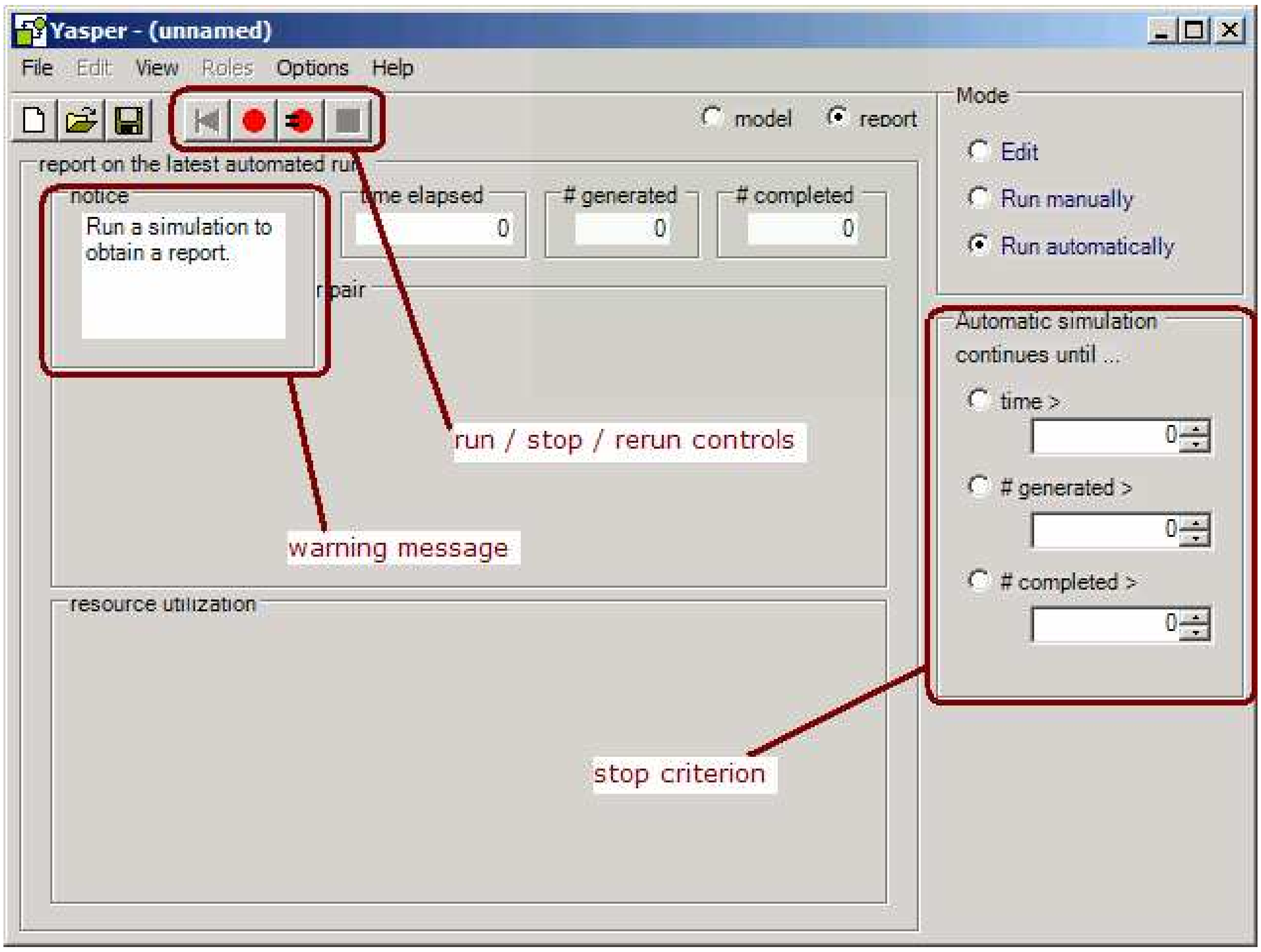
Another parameter merely helps summarize the behavior:

- *processing cost* for transitions (see section 2.9.1)

To modify these parameters, go to *Edit* mode; it is not possible to change them in run mode.

4.2 Preparing to run a simulation

Click on *Run automatically*. The controls for automatic simulation appear.



Run a simulation to obtain a report.

Three areas on the screen are of immediate interest:

- a *warning message* notifies whether simulation is possible
- *run controls* in the toolbar allow the simulation to be started
- a *stop criterion* to limit the simulation run

To get rid of the warning message, click on it.

To set a stop criterion, click the respective radio button and fill in a value:

- use *time* > to stop as soon as the given time is exceeded
- use *# generated* > to stop when the given number of cases has been generated
- use *# completed* > to stop when the given number of cases has been completed[23]

[23] *Completed* in the sense of section 1.6.4.5, i.e. its last case token was collected by a collector.

4.3 Not being able to run a simulation

If simulation is impossible, all run controls are disabled.

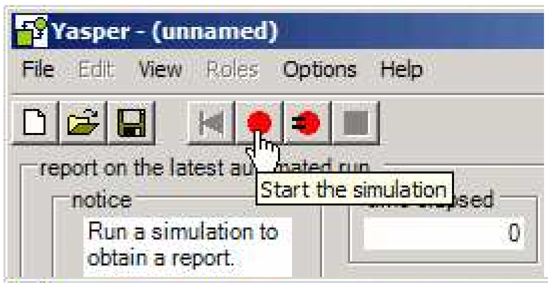
This will happen when

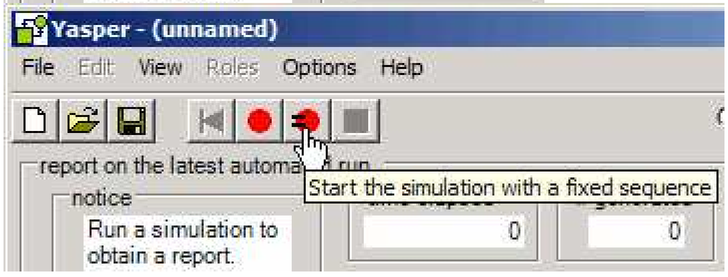
- no path of flow arcs leads from an emitor to a collector (so cases cannot complete)
- there is a timeless transition or XOR without inputs (so time cannot proceed)

To remedy this, click on *Edit* and change the model to remove these conditions.

For instance, basic Petri nets without case can often be prepared for simulation by adding an emitor, case-sensitive place and collector, and connecting them to the existing net.

4.4 Starting, halting and restarting a simulation

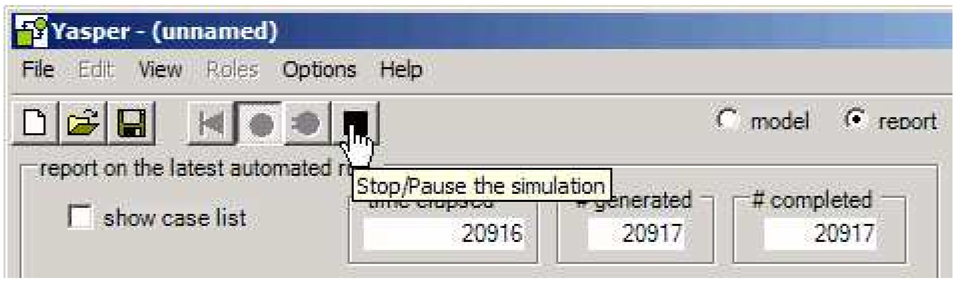

Yasper - (unnamed)
File Edit View Roles Options Help
report on the latest automated run
notice
Run a simulation to obtain a report.
Start the simulation
time elapsed
0


Yasper - (unnamed)
File Edit View Roles Options Help
report on the latest automated run
notice
Run a simulation to obtain a report.
Start the simulation with a fixed sequence
0
0

To start a run, click Start the simulation or Start the simulation with a fixed sequence.
In the latter case, runs always take the same course when repeated.

These buttons are disabled when simulation cannot start.
This is the case when

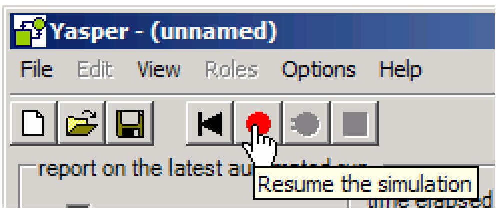
- a run is already in progress
- a halting criterion has been specified, and it is met
- no (first or next) step is possible (the simulation is in *deadlock*)
- simulation is impossible (as specified in the previous section)

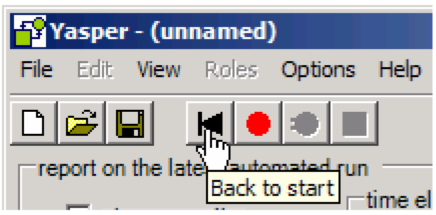

Yasper - (unnamed)
File Edit View Roles Options Help
model
report
report on the latest automated run
show case list
Stop/Pause the simulation
time elapsed
20916
# generated
20917
# completed
20917

During a run, the current time, number of generated cases, and number of completed cases are displayed in the respective fields.[24]

To halt a run, click *Stop/Pause the simulation*. A report is generated.

[24] Whether or not these numbers always increase during a run depends on the model being simulated.

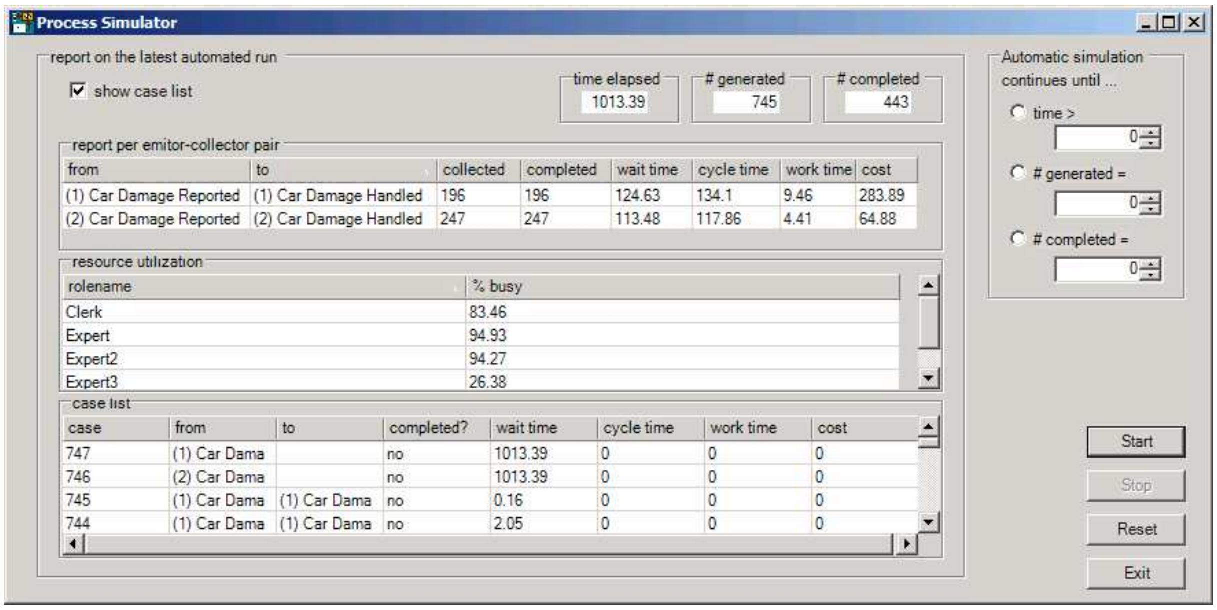



Yasper - (unnamed)
File Edit View Roles Options Help
report on the late automated run
Back to start
time el

To continue a halted run, click *Resume the simulation.*
To revert to the initial situation, click *Back to start*.

4.5 The simulation report

When a simulation run halts, a report is generated.


Process Simulator

report on the latest automated run

☑ show case list

time elapsed: 1013.39 | # generated: 745 | # completed: 443

report per emitor-collector pair

from	to	collected	completed	wait time	cycle time	work time	cost
(1) Car Damage Reported	(1) Car Damage Handled	196	196	124.63	134.1	9.46	283.89
(2) Car Damage Reported	(2) Car Damage Handled	247	247	113.48	117.86	4.41	64.88

resource utilization

rolename	% busy
Clerk	83.46
Expert	94.93
Expert2	94.27
Expert3	26.38

case list

case	from	to	completed?	wait time	cycle time	work time	cost
747	(1) Car Dama		no	1013.39	0	0	0
746	(2) Car Dama		no	1013.39	0	0	0
745	(1) Car Dama	(1) Car Dama	no	0.16	0	0	0
744	(1) Car Dama	(1) Car Dama	no	2.05	0	0	0

Automatic simulation continues until ...

○ time > 0

○ # generated = 0

○ # completed = 0

A report can have three parts:

- *report per emitor-collector pair*
  (always included, but empty when no case was completed)

- *resource utilization*
  (only displayed when the model specifies roles)

- *case list*
  (only displayed when *show case list* is checked)

The *report per emitor-collector pair* displays statistics on completed[25] cases, aggregated by their emitor and completing collector. The columns display the following:

- *from* = the emitor's name and/or id[26]
- *to* = the completing collector's name and/or id
- *collected* = the total number of case tokens of cases with this emitor that ever arrived at this collector, including tokens for cases that haven't completed, or have completed in a different collector
- *completed* = the number of cases with this emitor completed by this collector

[25] *Completed* in the sense of section 1.6.4.5, i.e. their last case token was collected by a collector.
[26] Whether name and/or id are shown depends on the *Show names* and *Show identifiers* options in the View menu, cf. section 2.15.

All further quantities are averages on the completed cases only:

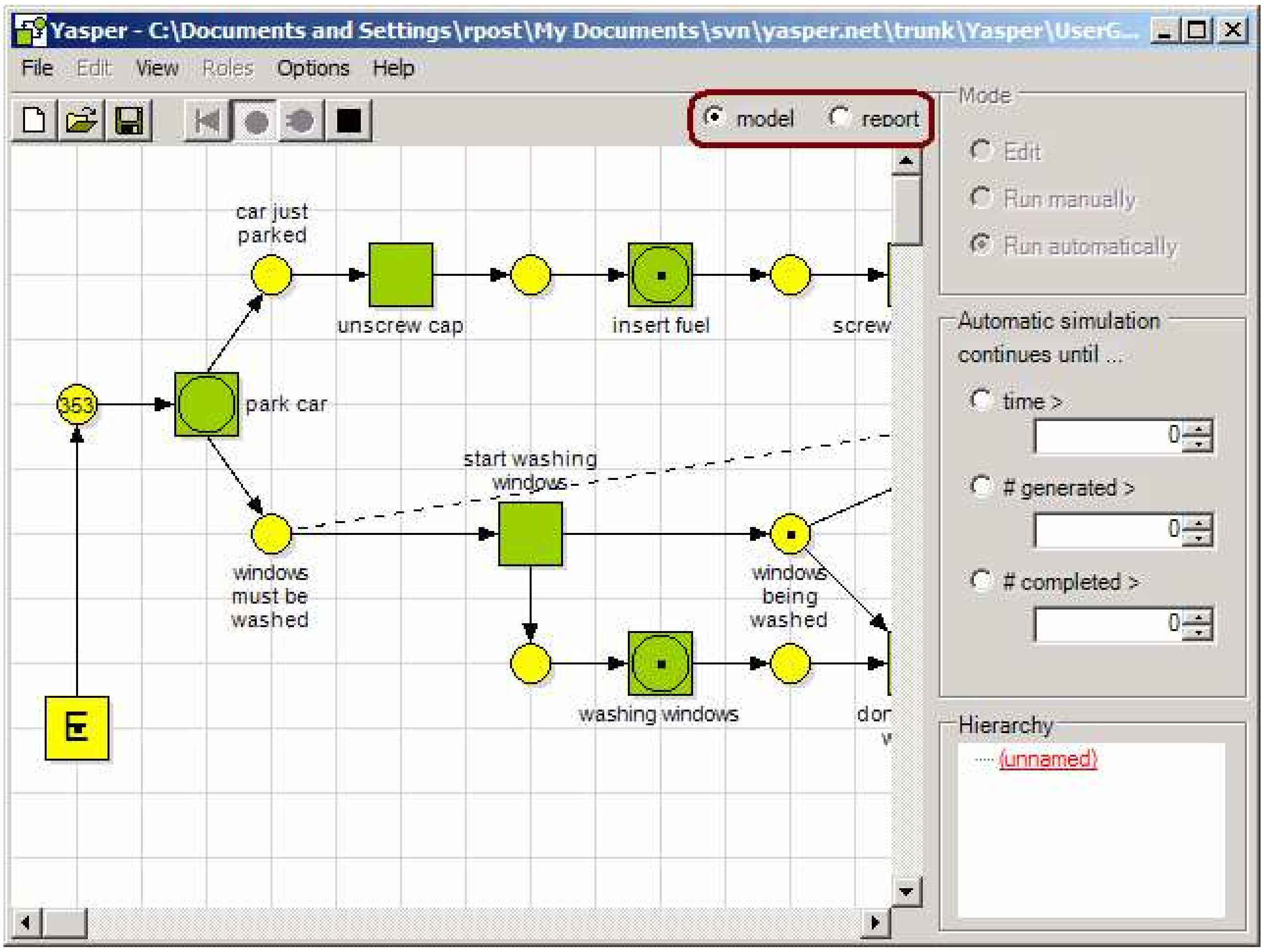
- *work time* = the average amount of processing time spent on a complete case by tasks; if parallel processing was done on the case, the processing times of all case tokens are added together
- *wait time* = the average amount of time a case was kept waiting, i.e. during which it had one or more tokens and no transition was processing any of them
- *cycle time* = the average amount of time a case was in the system, measured from leaving its emitor to being completed at its collector
- *cost* = the average processing cost of a case

The *resource utilization* report displays, for each role, what percentage of time its instances were involved in processing a task. All cases are considered, not only the ones that have completed.

The *case list* displays information on each individual case, whether completed or not. The meaning of the columns is the same as for the first table, except that it isn't aggregated.

4.6 Switching views and modes

To the right of the toolbar, two buttons appear in automatic run mode.
These can be used to switch views between the reports and the diagram.



To see the token flow while a simulation is running, click on *model*.
To switch back to viewing the reports, click on *report*.
This can be done before, during, or after a run.

Switching views in automatic run mode is not the same thing as switching to *Edit* or *Run manually* and back. The former does not affect the simulation state, and can even be performed while a run is in progress. The latter halts the simulation, and either rewinds to the initial marking, or, when *Keep final marking after runs* is set in the *Options* menu, modifies the state to be a valid marking, by forcibly terminating any processing being done in transitions, since tokens cannot reside inside transitions in either mode.

Some *File* operations can be invoked in automatic simulation mode, but have the effect of switching to edit mode, since they do not make sense unless performed in edit mode.

5 Index

(This index is still very incomplete.)