

Pragmadev Tracer

User Manual



Overview	4
MSC & PSC reference guide	5
General diagram format	5
Links	5
Message links	5
PSC-specific normal, required and failed message syntax	6
Operation call and return links	7
Semaphore take, take results and give	8
Dynamic instance creation links	8
Lifeline components	8
Timer events	9
Message save	9
Stop symbol	9
Action symbol	10
Method and suspended segments	10
Semaphore unavailability	11
Relative time constraints	11
Co-regions	12
PSC strict operator	12
PSC constraints: wanted and unwanted messages & chains	13
Main symbols	15
Lifeline	15
Semaphore lifeline	15
Collapsed lifelines	15
Condition or instance state symbols	16
MSC references	17
Inline expressions	17
Absolute times	21
Comments	21
Texts	21
Usage	22
Launching PragmaDev Tracer	22
Connection	22
Graphical user interface	23
Main window	23
Starting a trace	24
Options	26
Diagram editor	26
Creating and editing diagrams	29
Generating documentation: printing and exporting	46
Conformance checking: diagram diff & property match	51
Linking with model elements	59
Command line interface	61
CLI diagram comparison: msdiff	61
Tracer commands	63
Command reference	63

Common options and arguments	63
Task creation	65
Task deletion	65
Messages	65
Semaphore creation	68
Semaphore deletion	68
Semaphore take attempt	69
Semaphore take succeeded	69
Semaphore take timed out	70
Semaphore give	70
Timer start	70
Timer cancellation	71
Timer timed out	71
Task state changed	71
Action symbol	72
Start a new MSC trace	72
Pause MSC trace	72
Resume MSC trace	72
Close MSC trace	73
Exit PragmaDev Tracer	73
Set directory	73
Acknowledgment	73
Tracing example	74

1 - Overview

PragmaDev Tracer is a tool allowing to generate execution traces from actual applications running on targets. Traces can be created either online, from the actual behavior being executed, or offline, using a intermediate format obtained from the running application and analyzed afterwards by the tracer. The traces are displayed using the standard ITU-T MSC representation, or the SDL-RT MSC representation (see the SDL-RT website <http://www.sdl-rt.org>).

Online traces are created directly by the program running on the target which sends textual commands to the Tracer via a standard socket. Two modes are available:

- The graphical mode, allowing to display all the traces as they are created;
- The command-line mode, allowing to generate trace diagrams without visual user feedback. This is the ideal mode to generate traces in batch operations.

In addition to this main features, *PragmaDev Tracer* also allows:

- Direct creation of trace diagrams for documentation purposes;
- Visual comparison of trace diagrams;
- Creation of specification MSC diagrams, that can be compared to execution traces for conformance checking;
- Creation of property diagrams using the *Property Sequence Chart (PSC)* format, which can be matched against execution traces;
- Easy documentation of all diagrams, that can be exported fully or partially to common image formats, allowing to insert them in a document.

The general graphical form of the diagrams supported by *PragmaDev Tracer* is described in section “MSC & PSC reference guide” on page 5.

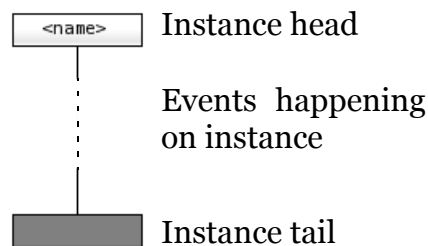
The general usage of the Tracer is detailed in “Usage” on page 22, including how to launch it, the options it accepts, a description of the main and tracer windows and the available preferences. The available commands that can be sent through the socket are described in section “Command reference” on page 63.

The tracing feature is described in “Starting a trace” on page 24. The MSC and PSC editor is described in “Diagram editor” on page 26, as well as the documentation features. The available checks that can be performed - trace against trace, specification against trace or property matches - are described in “Conformance checking: diagram diff & property match” on page 51.

2 - MSC & PSC reference guide

2.1 - General diagram format

A MSC or PSC diagram represents the interaction going on between entities called *instances* over time. Instances will typically be tasks, processes or objects. Instances are represented by symbols called *lifelines*, that look like follows:



The lifeline always starts with a head that specifies the instance name.

All events happening on the instance are then displayed on a vertical line under the lifeline head. These events are described below in “Lifeline components” on page 8.

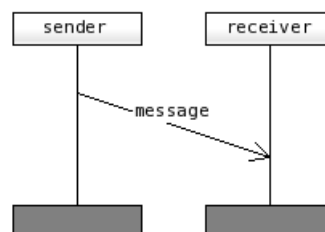
The lifeline terminates by a lifeline tail, that can take several forms depending on the status of the instance at the end of the diagram.

Lifelines are distributed along the horizontal axis, and the vertical axis represents the time, flowing from top to bottom. Events happening between lifelines are mostly represented by *links*, described in “Links” on page 5. Other symbols allow to further describe the diagram or add semantics to specification MSCs or PSCs; they are described in “Main symbols” on page 15.

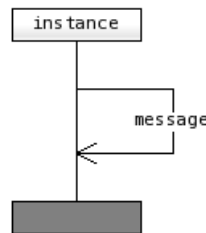
2.2 - Links

2.2.1 Message links

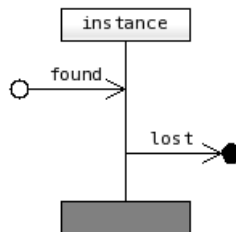
An asynchronous message sent by an instance and received by another is represented by a line with an outlined arrow at its end:



An instance can also send a message to itself:



A message can also be received from an unknown source, or sent to an unknown target. In this case, they are called a found message and a lost message respectively:



The syntax for the message link text is free, but *PragmaDev Tracer* expects a specific format for some features:

- Conformance checking by diagram comparison or property matches has an option to consider message parameters or not, as explained in “Basic MSC diff: trace vs. trace, spec. vs. spec., ...” on page 51. In this case, the message link text should have the format:
 <message name> (<message parameters>)
 The <message name> will be everything up to the first ‘(’ in the text, and the parameters everything between this ‘(’ and the final ‘)’. If anything else than a space appears after the last ‘)’, the syntax won’t be recognized and ignoring message parameters will have no effect.
- The structured message parameters display also expects a specific format for the whole message link text. See “Message parameter format” on page 67 for more details.

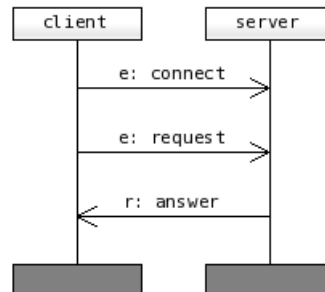
2.2.2 PSC-specific normal, required and failed message syntax

In PSC diagrams, texts for message links are supposed to be prefixed with one of the following:

- ‘e:’ indicates the message is a regular one. This means that the message is part of the precondition for the property: all messages prefixed with ‘e:’ must appear to trigger a property match. If any of these messages do not appear in the checked diagram, the preconditions for the property aren’t satisfied, and no matching is attempted. Regular messages should appear first in the PSC diagram.
- ‘r:’ indicates a required message. This means that if all the regular messages preceding this message are present in the checked diagram, this message must be present for the property to match. If it doesn’t, the property is violated.

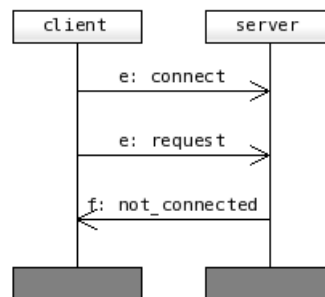
- ‘f:’ indicates a fail message. This means that if all the regular messages preceding this message are present in the checked diagram, this message must not appear for the property to match. If it does appear, the property is violated.

Here is an example of a required message in a property:



This means that if the `client` has sent a `connect` message to the server, then sends a `request` message, the server *must* send back an `answer` message, or the property is violated.

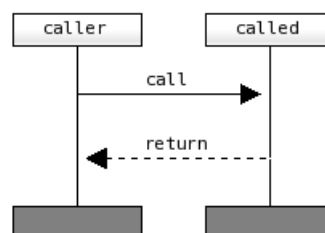
Here is an example with a fail message:



This means that if the `client` has sent a `connect` message to the server, then sends a `request` message, the server *must not* send back a `not_connected` message, or the property is violated.

2.2.3 Operation call and return links

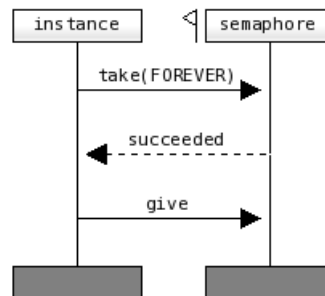
A synchronous call from an instance to another one is represented by a solid horizontal line with a block arrow at its end. The return of the call is represented by a dashed horizontal line, also with a block arrow at its end:



The syntax for these links is free, but PragmaDev Tracer will expect a specific syntax for the operation call link for some features. These features and the expected syntax are the same as for messages, as described in “Message links” on page 5.

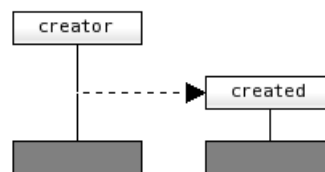
2.2.4 Semaphore take, take results and give

A specific kind of lifeline can be used to represent a semaphore in *PragmaDev Tracer* (see “Semaphore lifeline” on page 15). For these lifelines, the standard take and give operations are available, and are displayed like normal operation calls. The results of the take (success, time-out) is displayed as an operation return link:



2.2.5 Dynamic instance creation links

All lifelines are not necessarily present at the start of a MSC diagram, some of these can be dynamically created later. A dynamic creation is always done by another instance, and a dashed line with a block arrow to the head of the created instance is used to represent this creation:



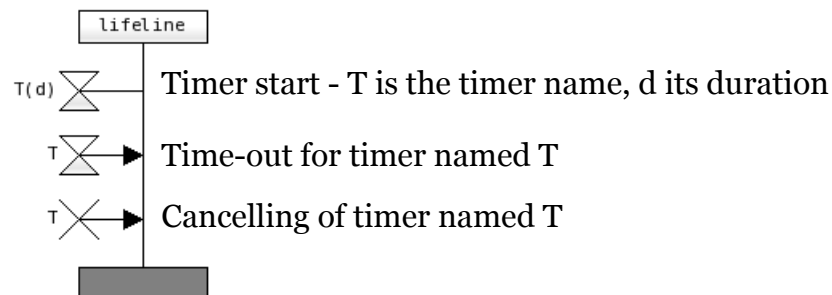
Note: in *PragmaDev Tracer*, this is the only way to have an instance starting elsewhere than at the top of the diagram.

2.3 - Lifeline components

Lifeline components are events impacting a single lifeline. They appear as symbols attached to the lifeline.

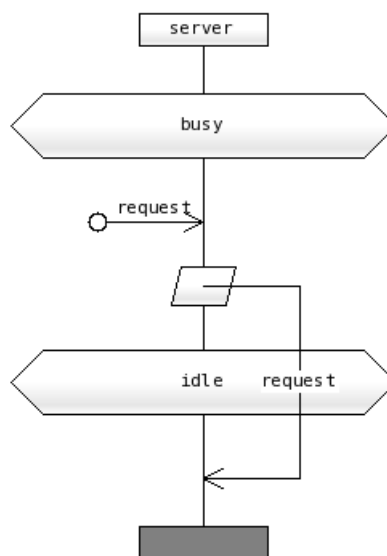
2.3.1 Timer events

An instance can start timers, that will time-out in a given amount of time. A timer can also be cancelled by the instance that created it. The symbols for timers are the following ones:



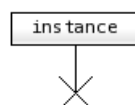
2.3.2 Message save

A message received by an instance can be saved to be treated later, for example after a state change. The message will be displayed as resent from the instance lifeline to itself when it is actually treated. For example:



2.3.3 Stop symbol

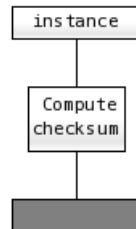
A stop symbol indicates that an instance has killed itself. Note that in standard MSCs, an instance can only kill itself, there is no notion of one instance killing another one. The stop symbol is displayed as follows:



and is always the last symbol appearing on a lifeline.

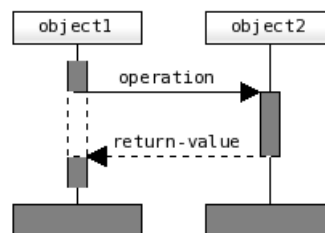
2.3.4 Action symbol

Action symbols describe actions performed by the lifeline. In the current version of *PragmaDev Tracer*, this description is informal. For example:



2.3.5 Method and suspended segments

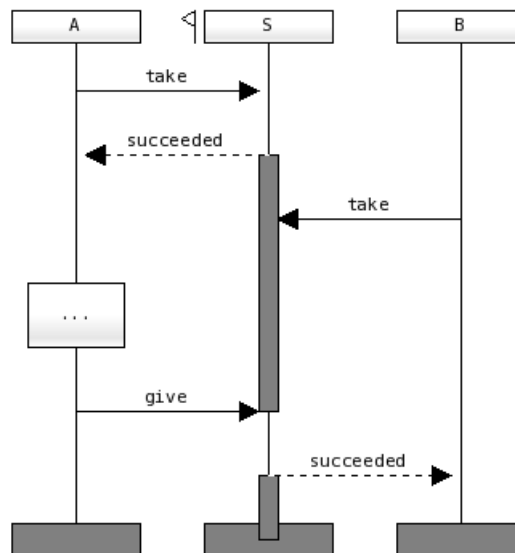
Segments are mostly used for instances describing objects that do not have parallel flows of execution. In this case, an object calling an operation on another one will be inactive (suspended) during the call while the other object actually executes the operation, or method. The control will be given back to the caller object when the operation returns. This can be shown in the MSC using method and suspended segments:



The instance `object1` is executing something (method segment) when it calls `operation` on `object2`. Then `object1` becomes inactive (suspended segment) while `object2` executes the operation (method segment), and `object1` becomes active again (method segment) when the operation returns.

2.3.6 Semaphore unavailability

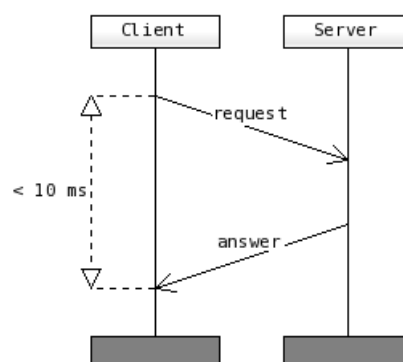
When a semaphore is taken by an instance and becomes unavailable, the same graphical representation is used on the semaphore lifeline as the method segment on an object lifeline. For example:



When the take from A to S succeeds, the semaphore becomes unavailable. When B attempts a take, it is put on hold until A gives the semaphore back. Then the take for B succeeds and the semaphore becomes unavailable again.

2.3.7 Relative time constraints

A relative time constraint appearing in a specification MSC diagram or a PSC diagram indicates that the sequence of events it encloses must happen within a given time. For example:



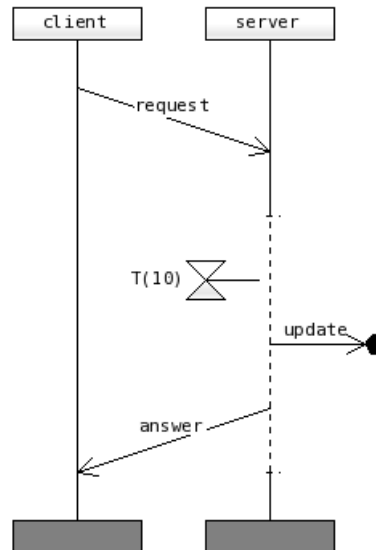
This specifies there must be less than 10 ms between the time when Client sends the request message and the time when it receives the answer message.

During conformance checking, relative time constraints are compared to absolute times in the compared diagram (see “Conformance checking: diagram diff & property match” on page 51 and “Absolute times” on page 21). Please note that units are not yet sup-

ported: relative time constraints can only contain a valid comparison operator (<, >, <=, >=, ...) followed by a real number.

2.3.8 Co-regions

A co-region on a lifeline specifies that all events happening on this lifeline can happen in any order, and not only the order specified graphically. For example:

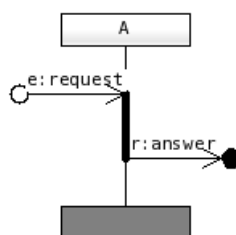


The coregion, indicated by the dotted line on the server lifeline, indicates that the timer and the outputs of messages `update` and `answer` can happen in any order.

Note that co-regions are not supported in the conformance checking feature of *PragmaDev Tracer* (“Conformance checking: diagram diff & property match” on page 51). The same semantics can usually be specified by using inline expressions; see “Inline expressions” on page 17 for details.

2.3.9 PSC strict operator

Events specified on lifeline in PSC diagrams are supposed to be loosely ordered by default. This means that if anything happens between two of these events, the property is matched anyway. It is however possible to specify a strict ordering for a set of events, meaning that these events must happen in this order without anything in between. This is done with the *strict operator*, that looks like follows:



This means that a request message received by A must be immediately followed by the output of an answer message, without anything in between (see “PSC-specific normal, required and failed message syntax” on page 6 for the PSC-specific link text syntax).

2.3.10 PSC constraints: wanted and unwanted messages & chains

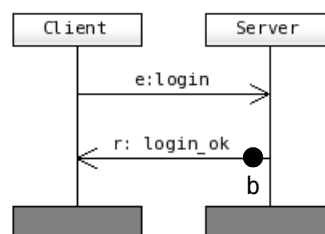
PSC diagrams allow to specify on a message a set of messages that must or must not appear before or after it for the property to match. Unlike other messages, the messages in these constraint appear in what PSC calls the *intra-message* format, i.e as a text formatted like: <sender instance name>.<message name>.<receiver instance name>.

The constraint itself is represented by a symbol appearing under one end of the message link:

- If it appears under the link start (message output on sender), it is a *past constraint*, meaning it must be satisfied before the message is sent for the property to match;
- If it appears under the link end (message input on receiver), it is a *future constraint*, meaning it must be satisfied after the message has been received for the property to match.

This kind of constraint can have several forms:

- An *unwanted message constraint* specifies a set of messages that should not appear. If any of the specified messages appear, the constraint is not satisfied and the property does not match. This kind of constraint is represented as follows:

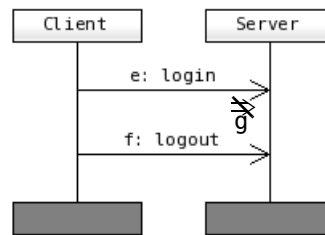


b = {Client.cancel_login.Server, Client.logout.Server}

The circle represents an unwanted message constraint, which is a past one since it appears under the link start. So this specifies that if the Client sends a login message to the Server, the Server must answer by sending back a login_ok message, unless either the message cancel_login has been sent from the Client to the Server before, or the message logout has been sent by the Client to the Server before.

- An *unwanted chain constraint* specifies a sequence of events that should not appear. If all messages in the constraint appear in the order specified in the con-

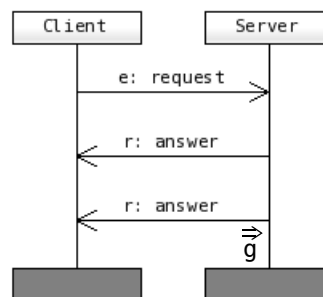
straint, then the property does not match. This kind of constraint is represented as follows:



$g = (\text{Client.request.Server}, \text{Server.answer.Client})$

The arrow with a slanted line represents an unwanted chain constraint, which is a future one since it appears under the link end. So this specifies that if the Client sends a login message to the Server, it is a property failure if it sends a logout message after it, unless it has sent the request message and the Server has sent back the answer message in-between.

- A *wanted chain constraint* specifies a sequence of events that must appear. If any of the messages in the constraint does not appear, or the messages appear in a different order than the one specified in the constraint, then the property does not match. This kind of constraint is represented as follows:



$g = (\text{Client.repeat.Server})$

The constraint appears under the link start, so it's a past message constraints. So this specifies that if the Client sends a request message to the Server, the Server must send back an answer message, and then another one if the Client sends the repeat message after the first answer.

Note: the symbols shown above are actually not yet available in *PragmaDev Tracer*. It actually supports more general types of constraints called wanted and unwanted alternative chain constraint. These merge the message and chain constraints described above. The general syntax for these constraints is:

$[<\text{constraint type prefix}> I_1.m_1.J_1, I_2.m_2.J_2, \dots \mid I_n.m_n.J_n, \dots \mid I_m.m_m.J_m, \dots]$

where $<\text{constraint type prefix}>$ can be either \Rightarrow for a wanted constraint, or $\neq \Rightarrow$ for an unwanted constraint.

- If the constraint is unwanted, this specifies that neither the sequence $I_1.m_1.J_1, I_2.m_2.J_2, \dots$, nor the sequence $I_n.m_n.J_n, \dots$, nor the sequence $I_m.m_m.J_m, \dots$ should appear for the property to match.

- If the constraint is wanted, this specifies that either the sequence $I_1.m_1.J_1$, $I_2.m_2.J_2$, ..., or the sequence $I_n.m_n.J_n$, ..., or the sequence $I_m.m_m.J_m$, ... must appear for the property to match.

This allows to represent all the PSC constraint kinds:

- An unwanted message constraint $\{I_1.m_1.J_1, I_2.m_2.J_2\}$ will be represented as:
 $[=\backslash=> I_1.m_1.J_1 \mid I_2.m_2.J_2]$
- An unwanted chain constraint $(I_1.m_1.J_1, I_2.m_2.J_2)$ will be represented as:
 $[=\backslash=> I_1.m_1.J_1, I_2.m_2.J_2]$
- A wanted chain constraint $(I_1.m_1.J_1, I_2.m_2.J_2)$ will be represented as:
 $[==> I_1.m_1.J_1, I_2.m_2.J_2]$

2.4 - Main symbols

2.4.1 Lifeline

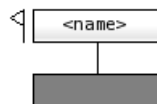
A lifeline represents an interacting entity in a MSC or PSC diagram, as explained in “General diagram format” on page 5. PragmaDev Tracer allows the instance name appearing in the lifeline head to have the following format:

```
<instance name>[:<class name>][(<instance identifier>)]
```

Lifelines can appear in all kinds of diagrams: SDL and SDL-RT MSC trace or specification diagrams, as well as PSC diagrams.

2.4.2 Semaphore lifeline

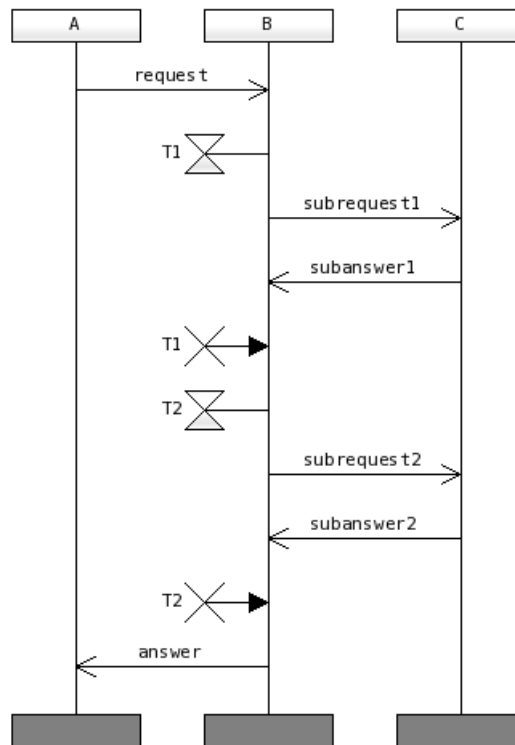
Semaphore lifelines are a SDL-RT extension to the standardized MSC format allowing to represent semaphores in the running system. They are displayed like a regular lifeline with an added flag near the lifeline head:



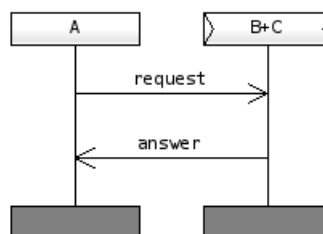
2.4.3 Collapsed lifelines

Collapsed lifelines are a *PragmaDev Tracer* extension and result from a ‘collapse’ operation. This allows to represent a set of lifelines as a single lifeline, events happening

between the lifelines in the set being hidden. For example, after collapsing the instance B and C in the following diagram:



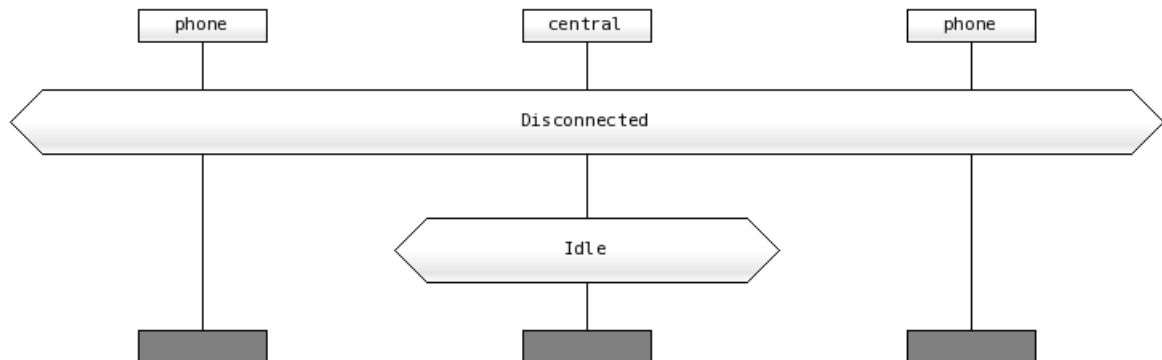
the diagram appears as follows:



2.4.4 Condition or instance state symbols

A condition symbol represents a condition for all the lifeline it spans. It is typically used to represent a state for a set of lifelines, the whole system, or for one specific lifeline. *PragmaDev Tracer* uses this symbol to represent state changes for an instance received via the 'taskChangedState' or 'ps' commands.

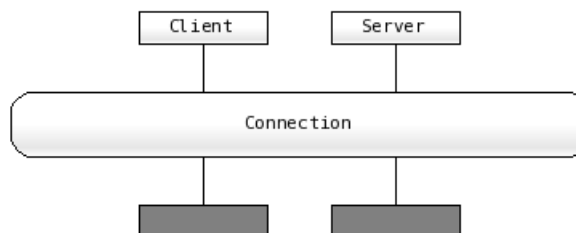
Here is an example of 2 condition symbols, the first one for the whole set of lifelines, and the second one for only one lifeline:



2.4.5 MSC references

A MSC diagram can reference another one by using a MSC reference symbol. This can be used to split a big MSC into smaller parts, or to reference the same sequence of events several times in a MSC diagram. This kind of symbol is normally only found in specification or PSC diagrams.

Here is an example of a MSC diagram referencing another one, called 'Connection':

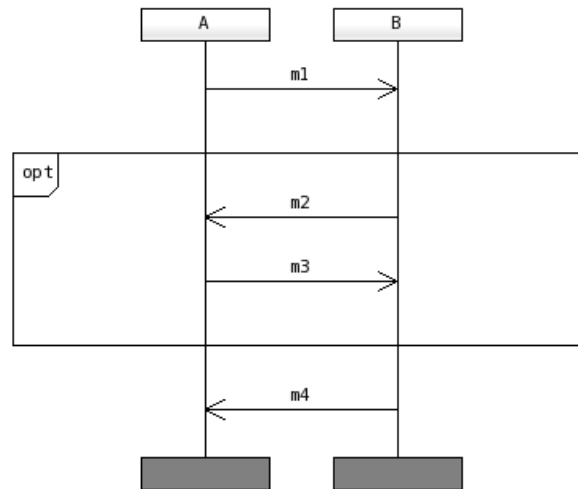


Note that in the current version of *PragmaDev Tracer*, there is no way to actually attach a MSC diagram to a MSC reference symbol. So this kind of symbol is mainly supported for documentation purposes.

2.4.6 Inline expressions

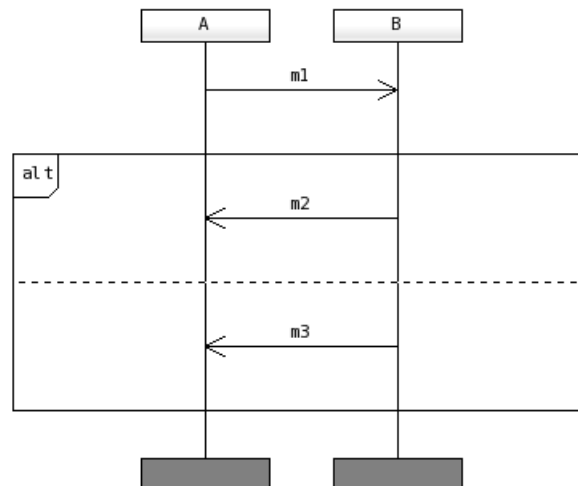
An inline expression in a specification or PSC diagram is a way to specify specific semantics for a group of events. The semantics depend on the kind of inline expression:

- An 'opt' inline expression specifies an optional set of events. For example:



specifies that the message m1 is sent from A to B, then B may send m2 to A, which answers m3, then B sends m4 to A. So the sequences m1-m2-m3-m4, and m1-m4 are both valid.

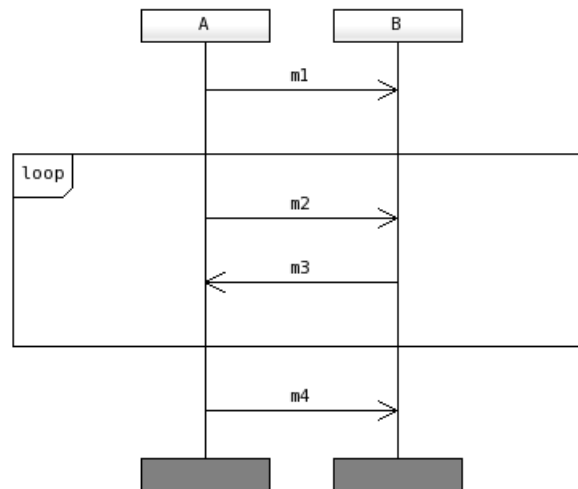
- An 'alt' inline expression specifies a set of alternative behaviors. For example:



specifies that when A sends m1 to B, B may answer by sending back m2, or m3. So the sequences m1-m2 and m1-m3 are both valid.

An 'alt' inline expression must have at least two compartments in it, and can have as many as needed.

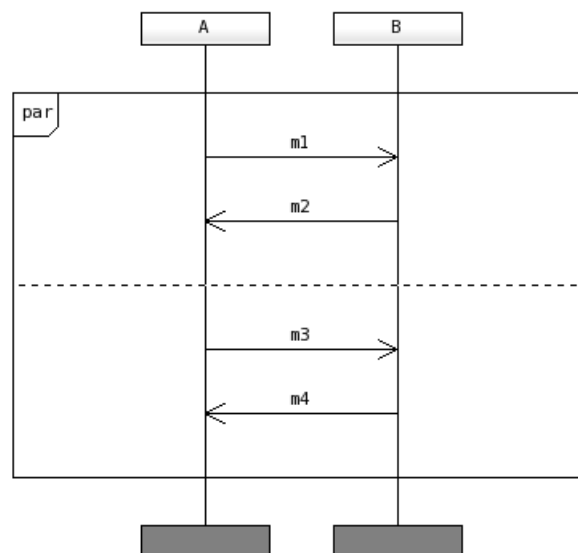
- A 'loop' inline expression specifies a set of events that might be repeated several times. For example:



specifies that after A has sent the message m1 to B, it may send any number of messages m2, to which B will answer by the message m3, until A sends the message m4 to B. So the sequences m1-m4, m1-m2-m3-m4, m1-m2-m3-m2-m3-m4, and so on, are all valid.

Note that the MSC standard allows to indicate minimum and maximum number of repeats in loop inline expressions. This feature is not yet available in *PragmaDev Tracer*.

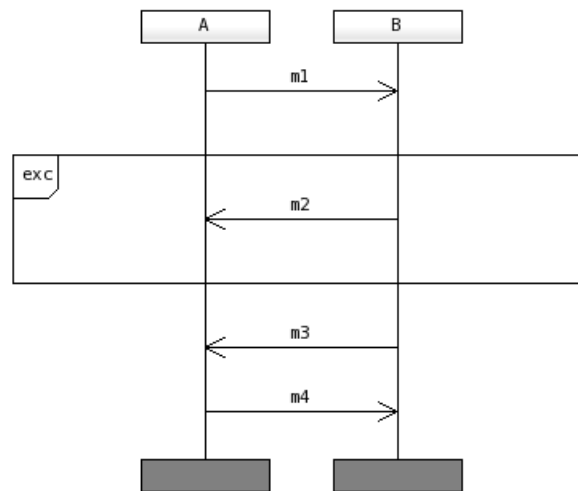
- A 'par' inline expression specifies a set of event sequences that must all happen, but in any order. For example:



specifies that the two sequences A sending m1 to B and B answering m2, and A sending m3 to B and B answering m4 must both happen, but that the order is not significant between the sequences. So the global sequences m1-m2-m3-m4 and m3-m4-m1-m2 are both valid.

A 'par' inline expression must have at least 2 compartments, and can have as many as needed.

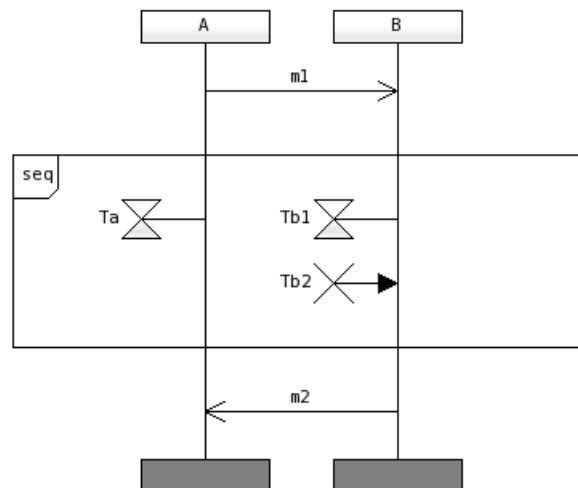
- An 'exc' inline expression represents an exception. This means the sequence of events in the inline expression is an error case and terminates the scenario. For example:



specifies that when A sends m1 to B and B answers m2, there is an error and the scenario should stop. So the sequence m1-m2 is valid, but is an error case, and the sequence m1-m3-m4 is valid and is a normal execution.

Note that the MSC standard represents an 'exc' inline expression with a dotted bottom line. *PragmaDev Tracer* uses a solid line in the current version.

- A 'seq' inline expression represents a weak sequence. This means that within such an inline expression, the events on a specific lifeline must happen in the given order, but the general ordering can be anything. For example:



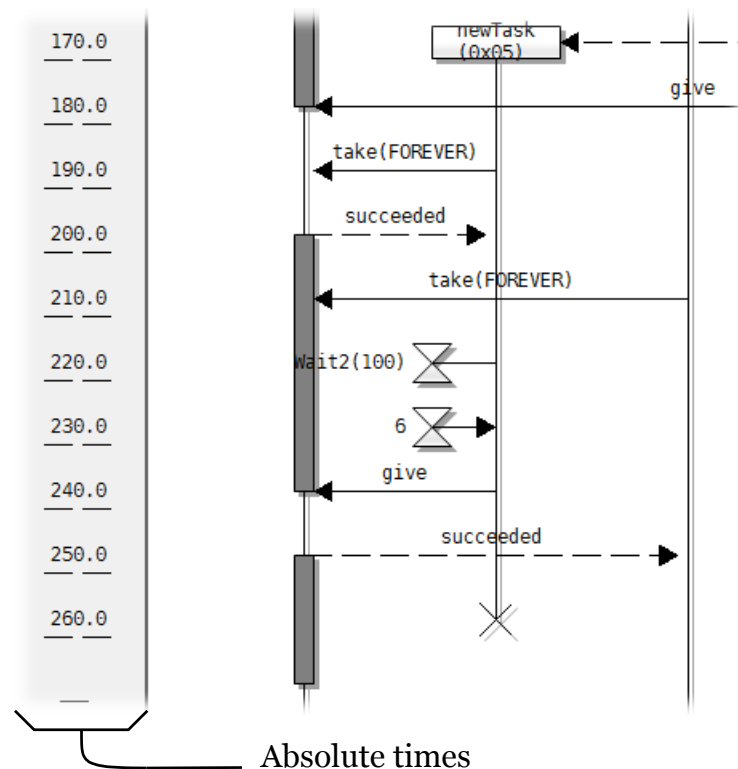
This means that on lifeline B, the starting of Tb1 has to happen before the cancelling of Tb2, but that the starting of Ta by A can happen at anytime: before the starting of Tb1, after the cancelling of Tb2 or between the two.

Note that this kind of inline expression is not supported in conformance checking ("Conformance checking: diagram diff & property match" on page 51).

2.4.7 Absolute times

In the MSC standard, absolute times can be associated to any event in the diagram by using a symbol consisting only in a dashed underline under the text for the time.

PragmaDev Tracer supports absolute times, but only associated to complete ‘event rows’: the times are displayed in the left margin of the diagram and are associated to all events with the same y coordinate, instead of any event. To keep the same representation as in the MSC standard, each absolute time is displayed with a dashed underline:



These absolute times are the reference when verifying relative time constraints during conformance checking (see “Relative time constraints” on page 11 and “Conformance checking: diagram diff & property match” on page 51). Please note that units are not yet supported: absolute time constraints must be written as a real number only.

2.4.8 Comments

A comment symbol just contains a documentation text for the item it is attached to. Comment symbols are not yet supported in *PragmaDev Tracer*.

2.4.9 Texts

A text symbol contains informal text usually describing global items in the diagram. Text symbols are not supported yet in *PragmaDev Tracer*.

3 - Usage

3.1 - Launching PragmaDev Tracer

The executable name is `pragmatracer.exe` on Windows and `pragmatracer` on Unix.

Usage is:

```
pragmatracer [-p <portNumber>] [-f <fileName>] [-d <directory>] [--nw]
```

- `-p <portNumber>`: sets the port number to use when starting socket connection.
- `-f <fileName>`: the filename where to save the trace. If provided the trace will be saved with this name, otherwise a name will be generated or asked to the user.
- `-d <directory>`: the default directory in which the trace will be saved. Used only if no file name is provided or if the file name does not indicate the entire path of the file.
- `--nw`: to launch the *PragmaDev Tracer* in command line mode.

Some features of *PragmaDev Tracer* are also available via a separate command line interface called via the executable `pragmatracercommand.exe` on Windows, and `pragmatracercommand` on Unix. This command is described in “Command line interface” on page 61.

3.2 - Connection

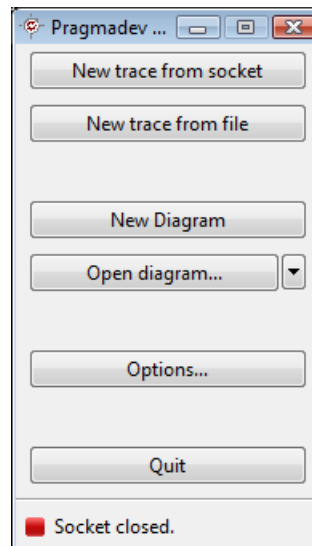
The connection is made by socket where *PragmaDev Tracer* runs as a server: the socket is initialized on a port number and then waits for a connection from a client. The port number can be set when starting the application (see above) or in the preferences (see “Options” on page 26). If no port number has been provided when starting the connection, *PragmaDev Tracer* uses the default value 50000. This port number must be the same for the client application which will connect to the tracer. If *PragmaDev Tracer* is in command line mode, the socket is opened as soon as the tracer is started.

Please note several clients can connect to the same *PragmaDev Tracer* and contribute to the same trace. As the socket can accept several connections, it might get tricky to synchronize several clients. For example, if 2 client applications are executed in a row in a shell script, the beginning of the trace of the second client application might get mixed with the end of the trace of the first one. To avoid this behavior, it is possible to ask for an acknowledgment to the tracer (see “Acknowledgment” on page 73). Waiting for an acknowledgment after the last command in the client program guarantees that the next client program will start when all commands of the previous program have been treated.

4 - Graphical user interface

4.1 - Main window

The main window is the main interface between the user and the tracer. It is the first that opens when starting the *PragmaDev Tracer* application and closing this window closes the application.



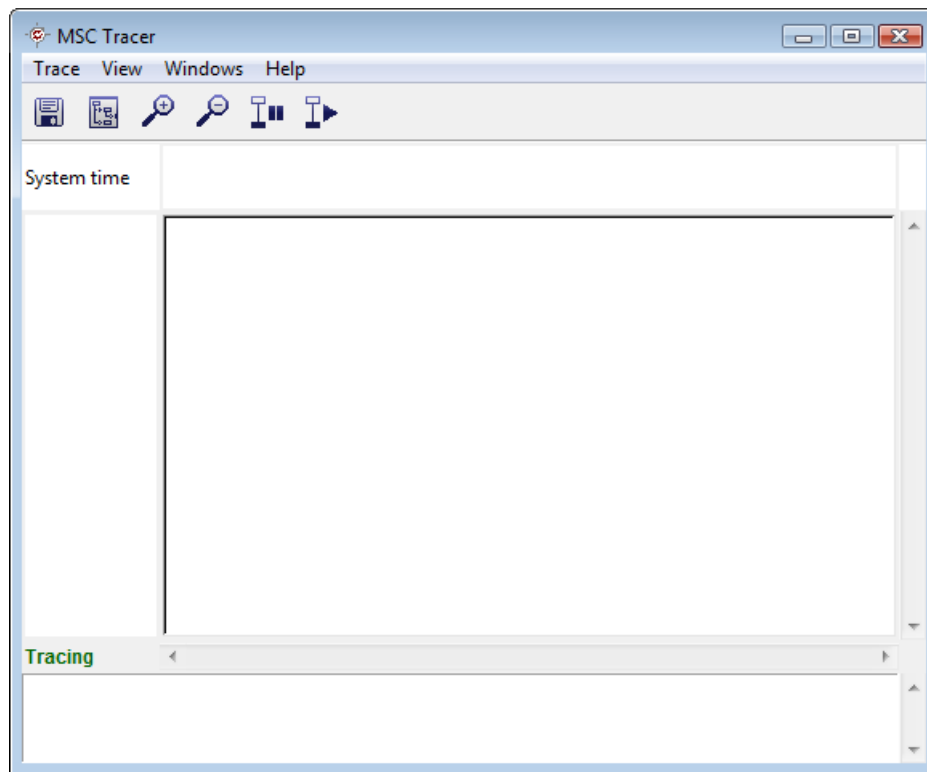
PragmaDev Tracer main window

The elements in this window are the following ones:

- “*New trace from socket*” starts a trace by reading commands from a socket, usually connected to an actual running program on a target. This kind of tracing is described in paragraph “Starting a trace” on page 24.
- “*New trace from file*” reads a sequence of commands as those sent through the socket from a file. This can typically be used for offline tracing.
- “*New diagram*” creates a new diagram and opens it in the diagram editor. The diagram can be a trace, a specification diagram or a PSC. The editor is described in paragraph “Diagram editor” on page 26.
- “*Open diagram*” opens an existing trace, specification or PSC diagram in the diagram editor. The arrow near the button pops a history menu, allowing to quickly open the last 10 opened diagrams. The editor is described in section “Diagram editor” on page 26.
- “*Options...*” opens the preferences dialog for the tracer. This dialog is described in “Options” on page 26.
- “*Quit*” quits the tracer, closing any opened trace or diagram.
- The indicator at the bottom of the window gives the socket status. If red, the socket is closed; if green, the socket is opened and a trace is running.

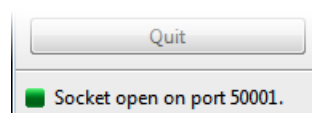
4.2 - Starting a trace

A click on "New trace from socket" initiates the socket connection and a trace window pops up:



Empty tracer window

The main window will display the socket status:




Socket opened

The trace can be paused or resumed by using the buttons  and  respectively.

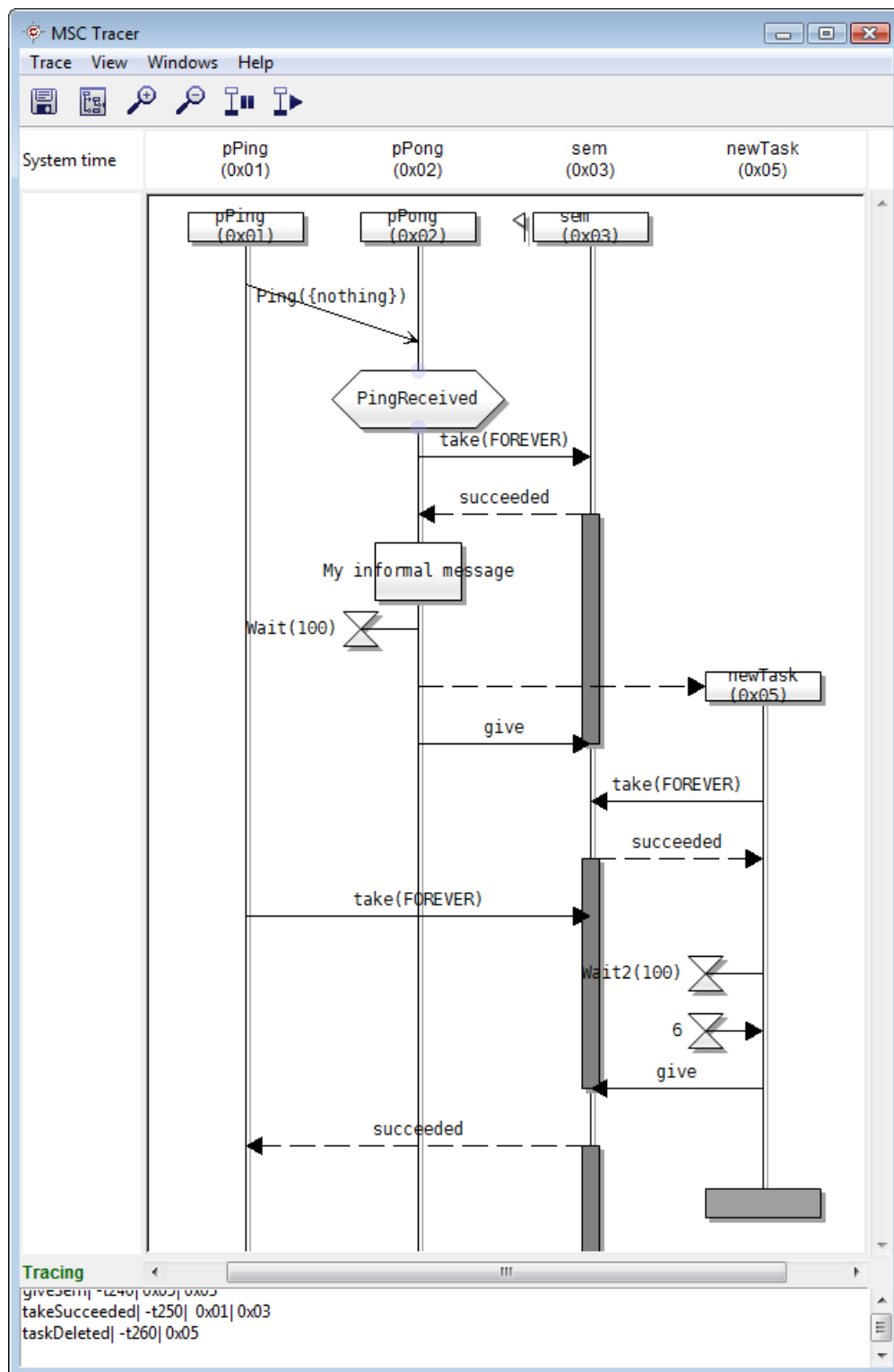
By default the trace is running. When the trace is paused, all received commands are ignored. Note that, by pausing the trace, the socket is not released so the client application can continue to send commands without generating an error.

The MSC trace can be zoomed in or zoomed out with  and .

The trace can be saved at any moment with the button .

If no file name was provided when starting the application (with -f option as described in “Launching PragmaDev Tracer” on page 22), a new name will be asked. When closing this window, the trace stops and the socket connection is closed.

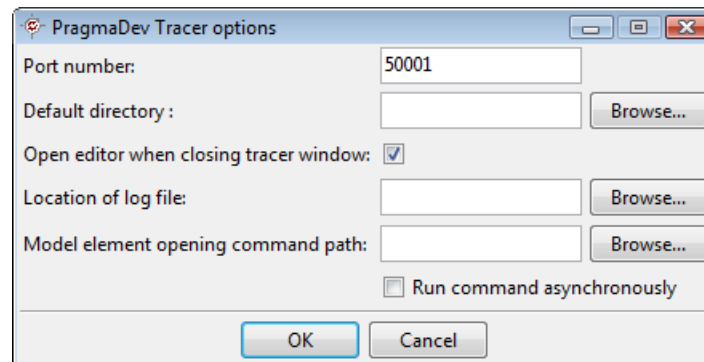
Here is an example of a trace displayed in the tracer window:



Trace in tracer window

4.3 - Options

Global options for the tracer can be set up by clicking on the “Options...” button in the main tracer window. Options are configured via the following dialog:



The available options are:

- “*Port number*”, which is the default port number to use for next socket connections;
- “*Default directory*”, which is the directory where the traces will be saved if a simple file name is provided without directory;
- “*Open editor when closing tracer window*” allows to directly open the traces in the editor as soon as the tracer window is closed;
- “*Location of log file*” is the full path for the log file to use when the tracer is in non-graphical mode;
- “*Model element opening command path*” and “*Run command asynchronously*” are explained in detail in section “Linking with model elements” on page 59.

Note that on Unix and Linux, another option is also displayed, called “*AcrobatReader path*”, allowing to specify the path for the application allowing to open the PDF files for the tracer documentation.

4.4 - Diagram editor

The diagram editor allows to view and modify all kinds of diagrams supported by PragmaDev Tracer:

- Trace diagrams generated by the tracer;
- Trace diagrams created directly via the editor itself;
- Specification MSC diagrams;
- Property diagrams in the PSC format.

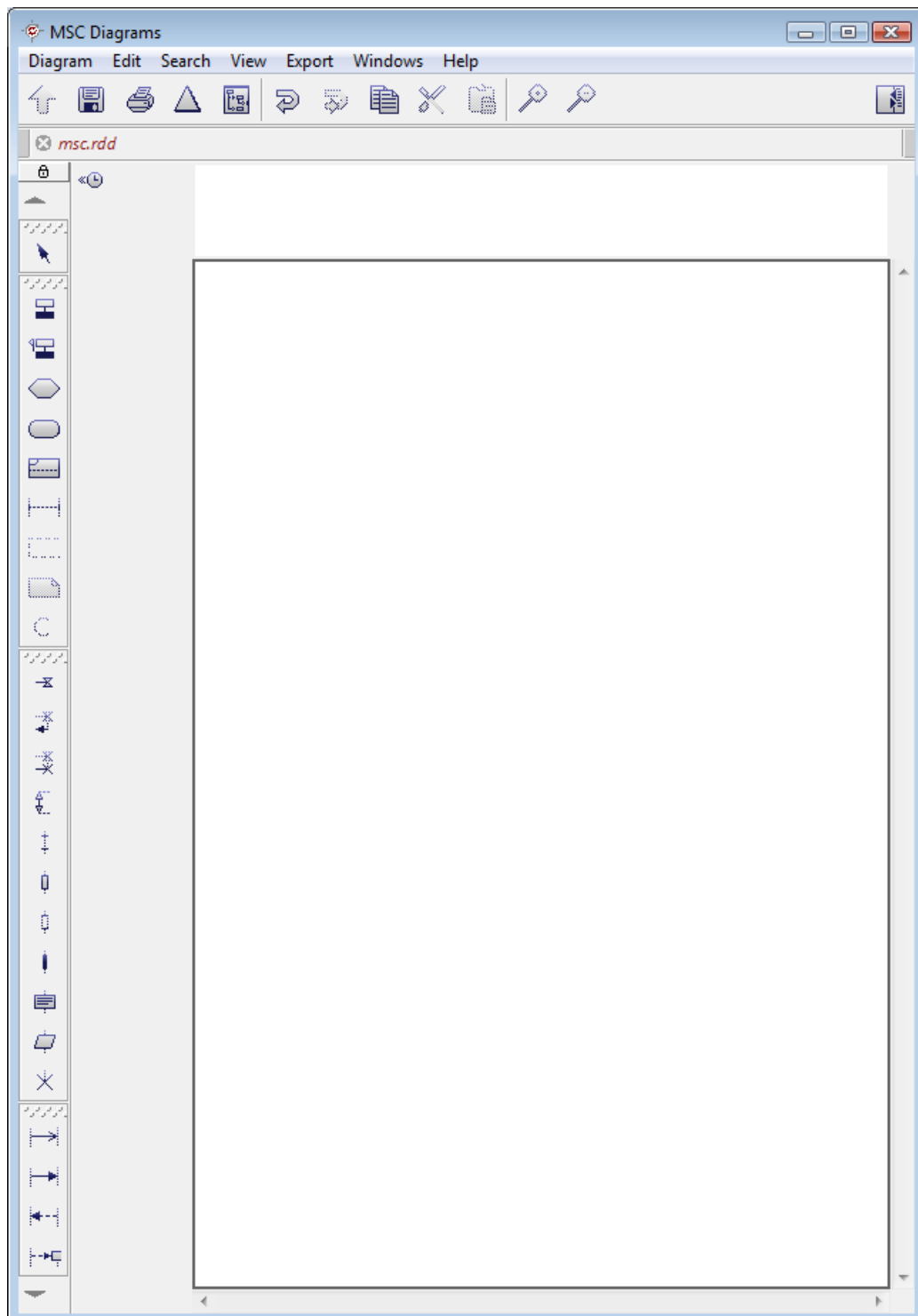
Various file formats are supported:

- PragmaDev Tracer’s own XML diagram file format.
- RTDS v4.x and earlier MSC diagram file format, in SDL or SDL-RT languages.
- Standardized ITU-T MSC-PR format. Note this format cannot be used for PSC diagrams, and cannot store some specific *PragmaDev Tracer* features, such as filters and collapsed lifelines.

These diagrams can be created by using the “*New diagram*” button in the main window. Note that creating a diagram will immediately ask for a file name to store it, as *PragmaDev Tracer* always needs a file name for diagrams, even empty ones. Existing dia-

grams can be opened via the “*Open diagram...*” button in the main window, or via the history button beside it.

The diagram editor window looks like follows:



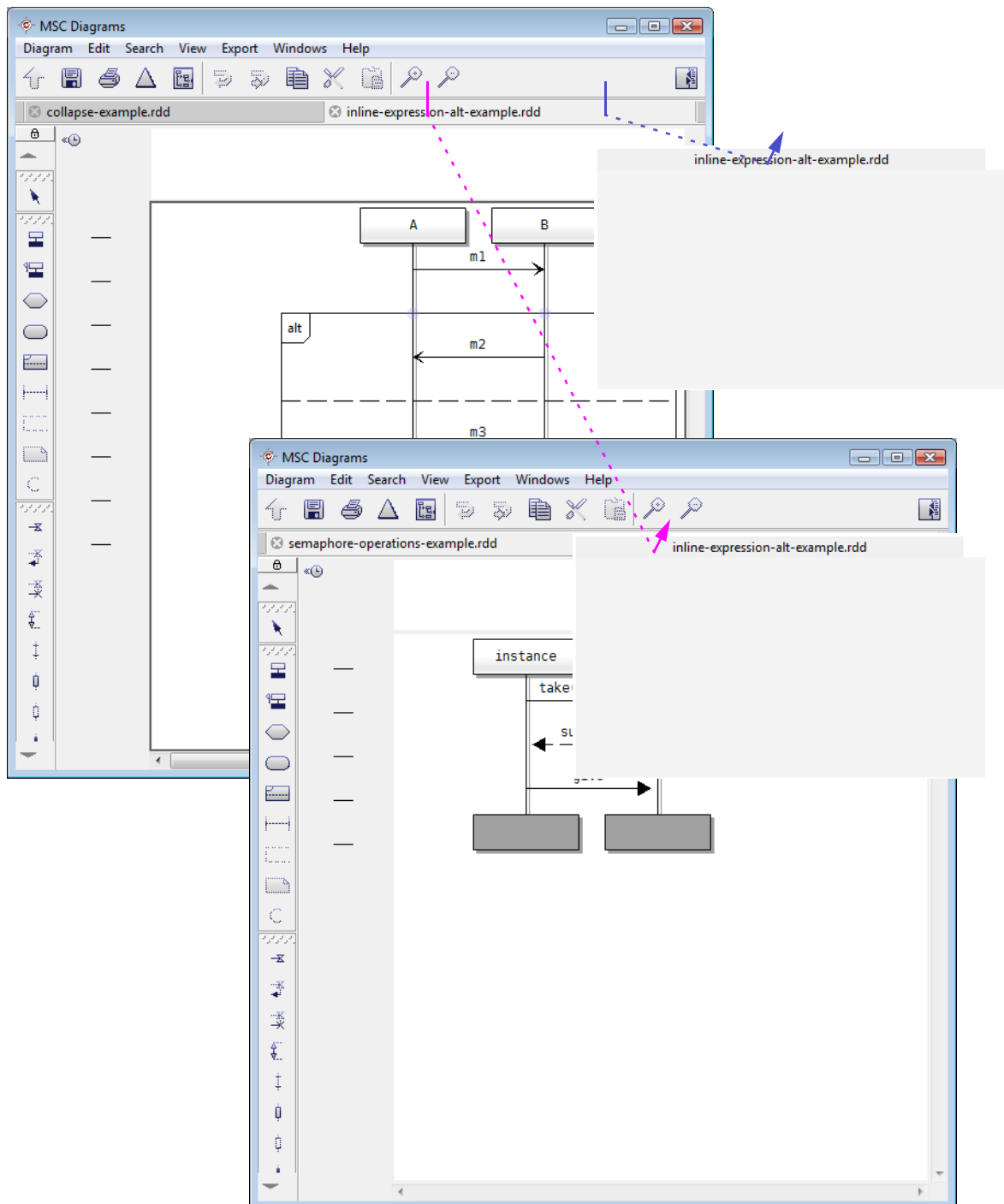
Empty diagram editor window

This window allows to:

- Create and edit diagrams;

- Print and export whole or parts of diagrams, typically for documentation purposes;
- Check diagrams by comparing them or look up property matches.

By default, all diagrams created or opened from the main window will appear in the same editor window in several tabs. If having a separate window for a given tab is needed, any tab can be detached to its own window, or put in another already existing window. To do this, just grab the tab with the mouse and drag it outside the window, or in the tab bar of another one:

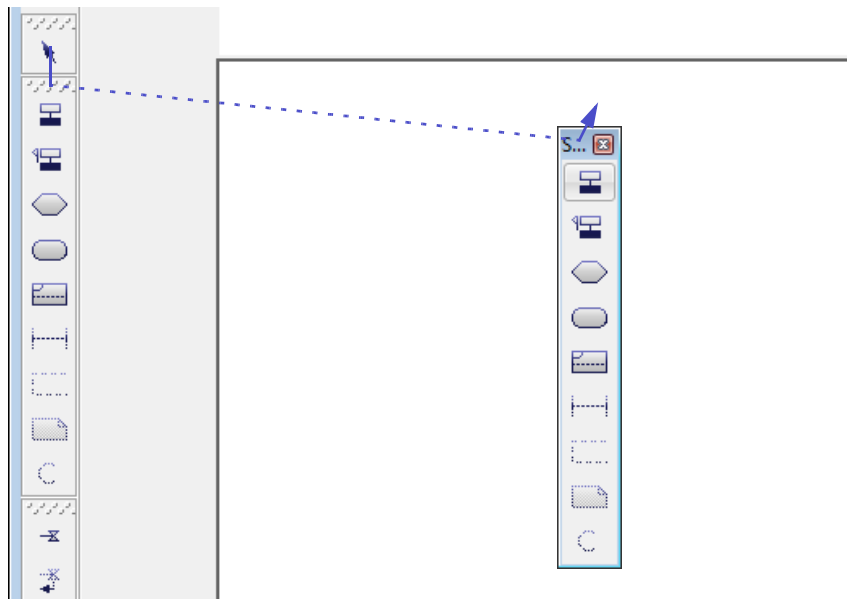


4.4.1 Creating and editing diagrams

Diagram creation and edition is mostly done via the toolbars placed on the left part of the window. The menus also allow to edit symbol and link properties, to perform searches and to customize the view of the diagrams by applying filters or collapsing and expanding lifelines. Once the editing is done, the diagram can be saved in its current file format or another one.

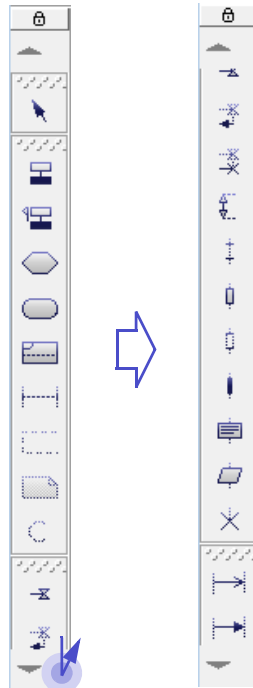
4.4.1.1 Toolbars for symbols, lifeline components and links

Toolbars on the left side of the diagram edition zone allow to select items in the diagrams and to insert symbols, lifeline components and to create links between instances. All these toolbars can be detached from the editor window by pressing the mouse button in their header and dragging the tool bar away:

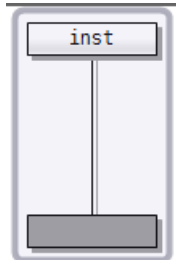


On most platforms, the detached toolbar will always be on top of the other windows. A detached toolbar can be put back in the diagram editor simply by closing its window.

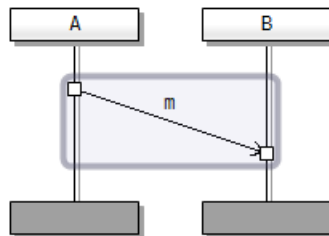
If the window is too small to display all toolbars, the arrows above and below them can be used to scroll the whole toolbar set:



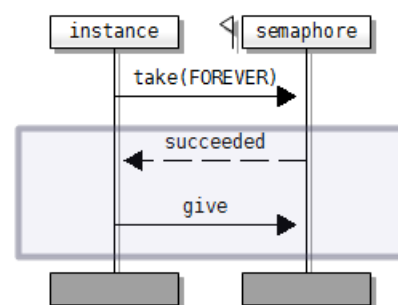
The top toolbar contains a single tool which is the selection tool. It allows to select symbols and links in the diagram, as well as to perform rectangular selections for exporting:



*Selected
lifeline*

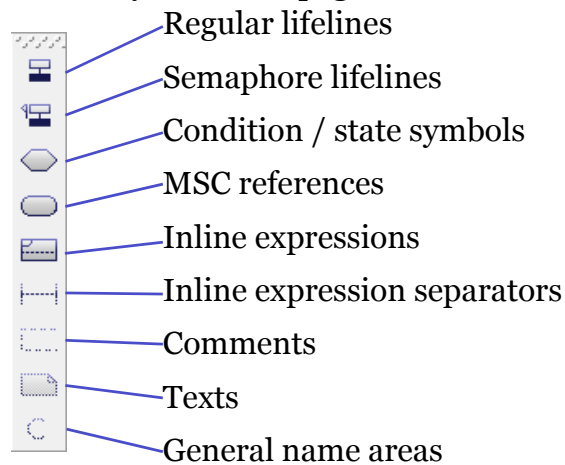


*Selected
message link*

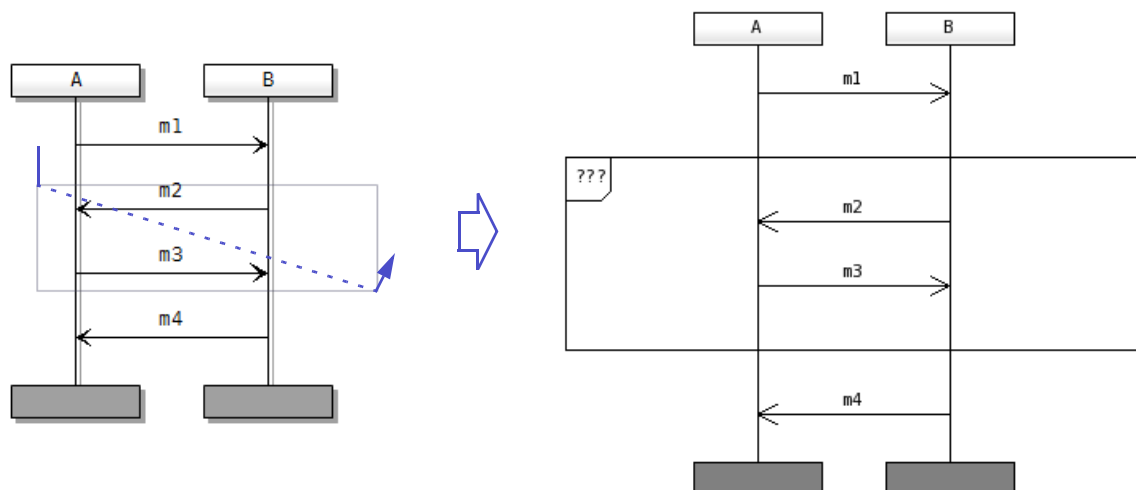


Rectangular selection

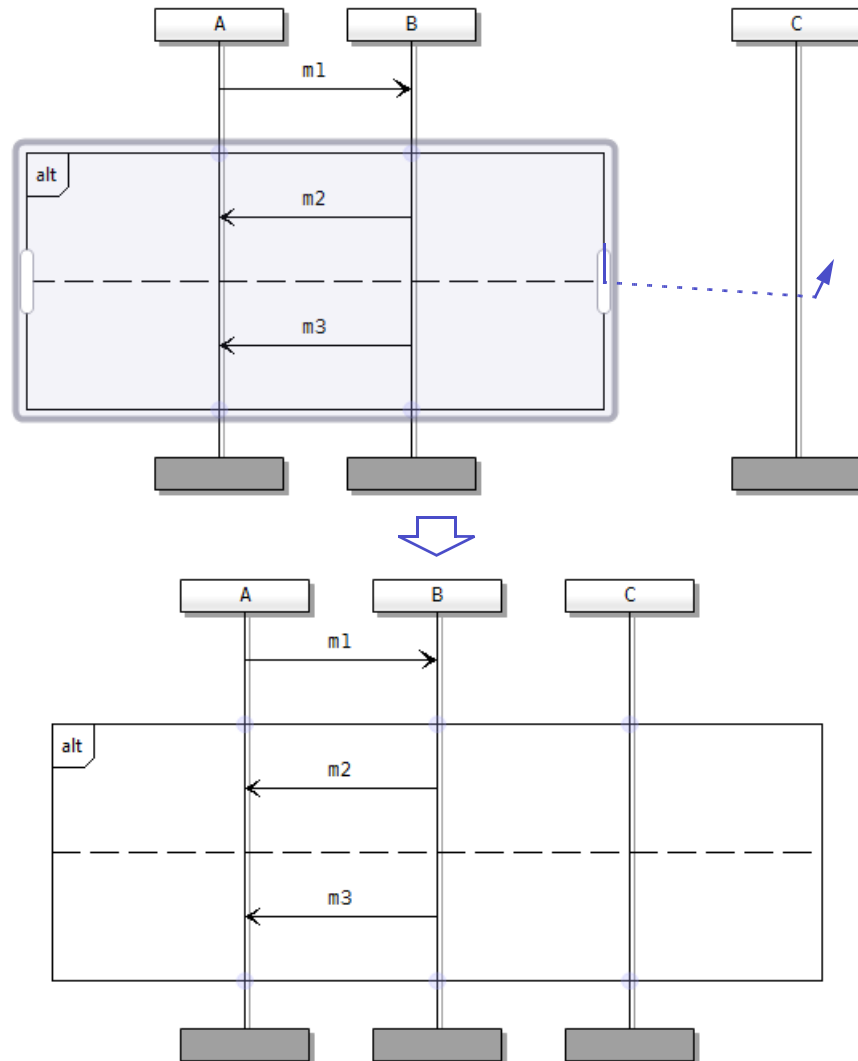
The next toolbar is the symbols insertion toolbar. It allows to create in the diagrams all the symbols described in “Main symbols” on page 15:



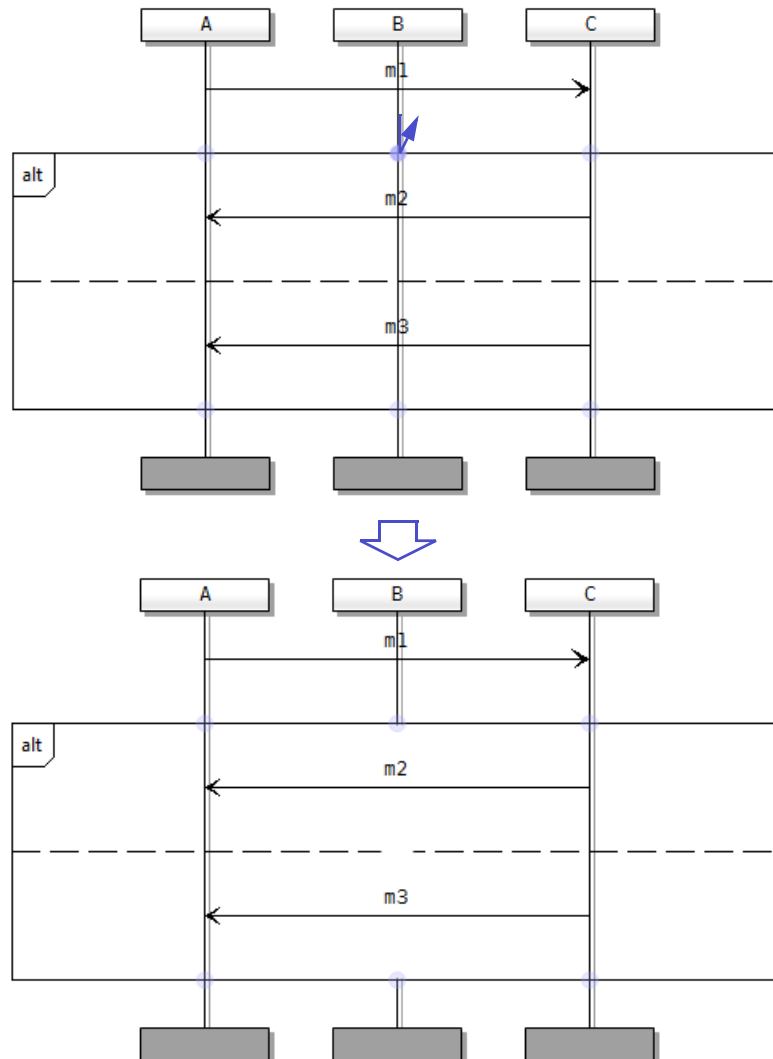
After selecting the creation tool, lifelines are created just by clicking at their positions in the diagram. Only the horizontal position will be considered, the lifeline will always be created going from the top of the diagram to its bottom. Creating condition, MSC references and inline expression symbols is done by pressing the mouse button on one of its corners, then dragging to the opposite corner and releasing the button:



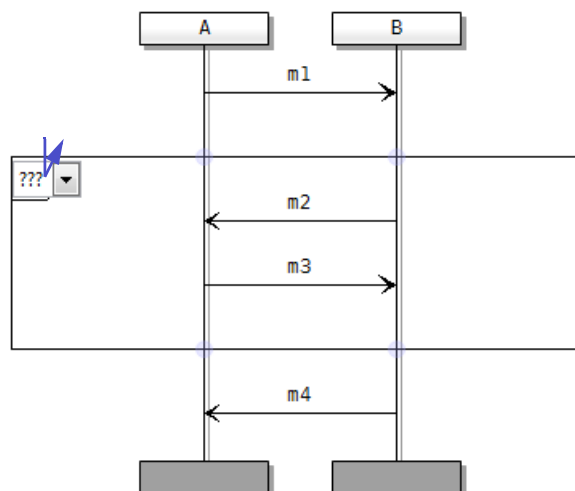
Once created, this symbols can be moved up or down by dragging them, and resized horizontally via the handles appearing on their sides when they are selected:



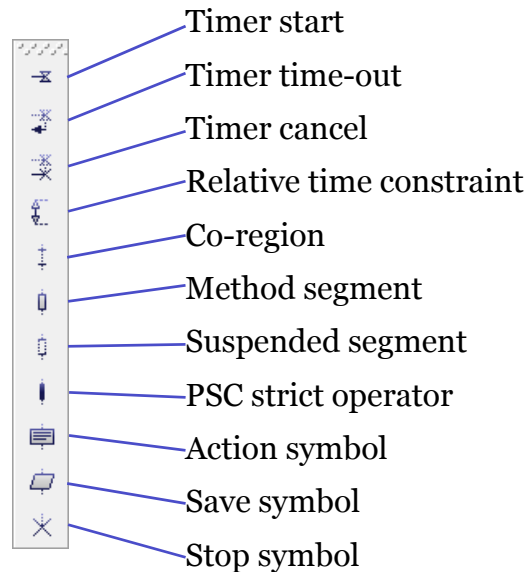
Excluding from the symbol a lifeline included in it can be done via the circular handles appearing at the connection points between the symbols and the lifelines:



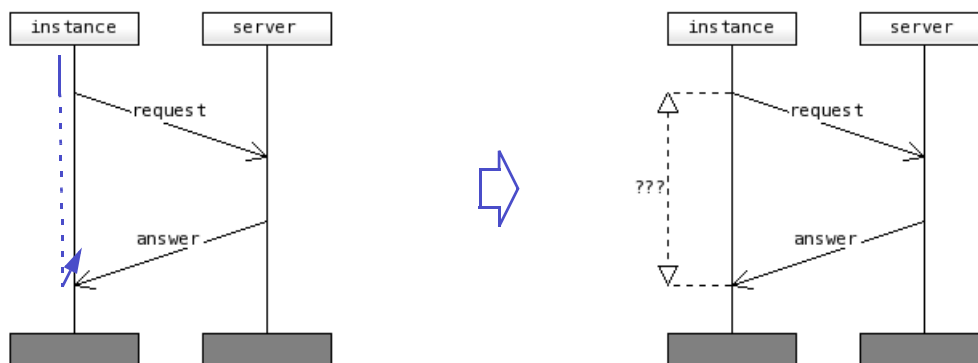
The kind for an inline expression can then be changed by selecting it in the box appearing when the mouse cursor is over its text:



The next toolbar is the lifeline component toolbar. It allows to attach to an existing lifeline all the components described in “Lifeline components” on page 8:

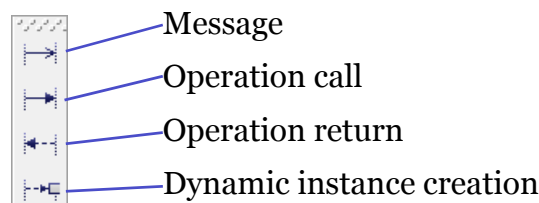


Insertion of timers, action symbols, save symbols and stop symbols are just done by clicking the corresponding tool, then clicking at the position of the new symbol on the lifeline. For relative time constraints, co-regions, segments and the strict operator, the insertion is done by pressing the mouse button at one end of the inserted component, then dragging to the other end and releasing the mouse button. For example, with a relative time constraint:



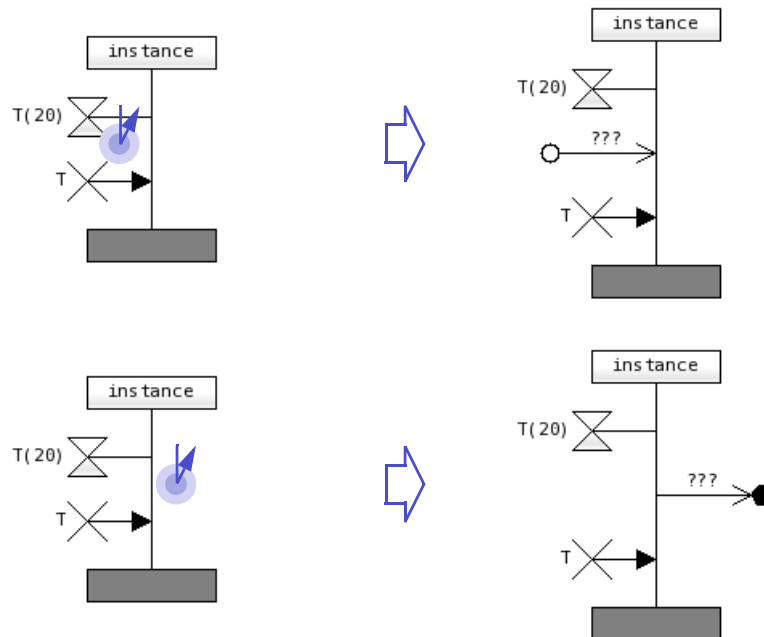
Note that the creation of a stop symbol at a position on a lifeline after which there are other events will be refused. You have to remove these events before creating the symbol.

The last toolbar is the link creation toolbar. It allows to create between two existing lifelines all the links described in “Links” on page 5:



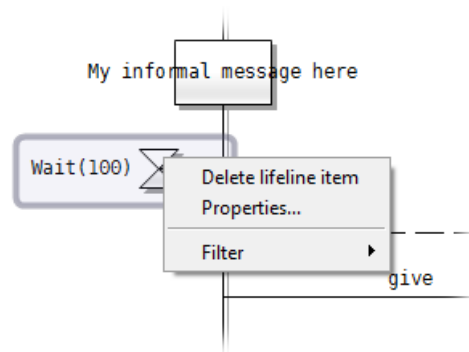
Creating a link is done by pressing the mouse button at its start position, then dragging to its end position and releasing the mouse button. Note all links except messages will be constrained to be horizontal.

The creation of a lost (resp. found) message is done by selecting the message creation tool and clicking on the right side (resp. left side) of the lifeline sending it (resp. receiving it). For example:



4.4.1.2 Operations on symbols and links

The text for symbols and links can be modified simply by clicking anywhere in it and typing. Clicking on any symbol or link will select it, displaying handles allowing to move and/or resize it. Right clicking on any displayed item will bring up a contextual menu showing the operations that can be done on it:



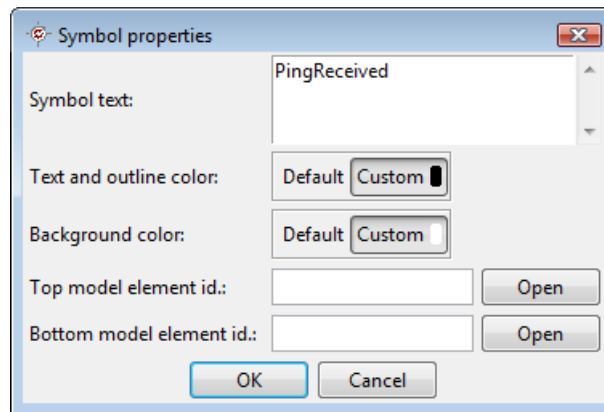
4.4.1.3 Symbol and link properties

Some attributes of a symbol or a link can only be changed via its properties dialog. This includes mostly its colors: text and outline color for both, and background color for symbols. Model element identifiers associated to symbols and links can also be viewed,

changed and opened through this dialog; see section “Linking with model elements” on page 59 for more details.

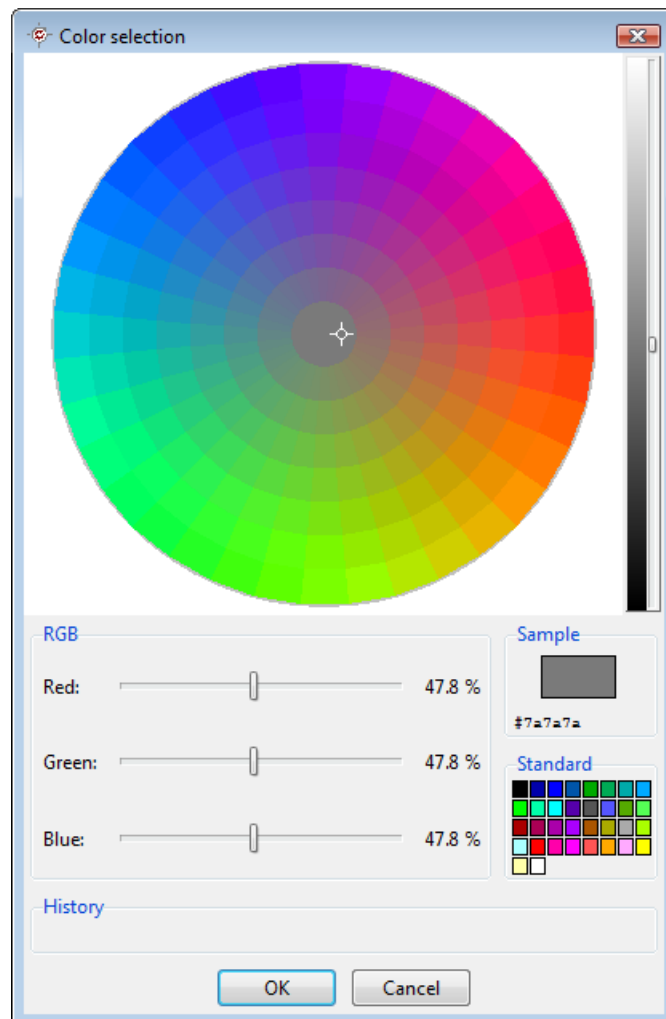
The properties dialog can be displayed by right-clicking on the item and selecting ‘Properties...’, or by selecting the item, then select ‘Properties...’ in the ‘Edit’ menu.

The dialog for the properties of a symbol looks like follows:



The top field allows to change its text. The colors are changed by using the ‘Text and outline color’ and ‘Background color’ fields: ‘Default’ will select the default colors (black for

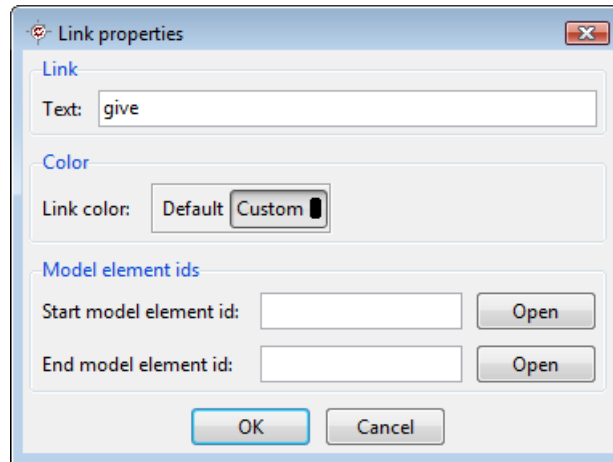
text and outline, white for background), and 'Custom' will bring up the color selection dialog:



Note you might have to change the lightness via the slider at the right side of the dialog to see all the colors. Selecting a color can be done by clicking on it either in the wheel, or in the 'Standard' box at the bottom right side, or by manually adjusting the sliders. The 'History' zone at the bottom will show all the colors you have used in the current session.

The last fields in the symbol properties are the model element identifiers associated to the symbol. There might be one of these if the symbol represents a single event (for example a timer start), or two if the symbol has a beginning and an end (for example, a lifeline or inline expression).

The dialog for link properties is similar, except it only includes a field for the text and outline color:

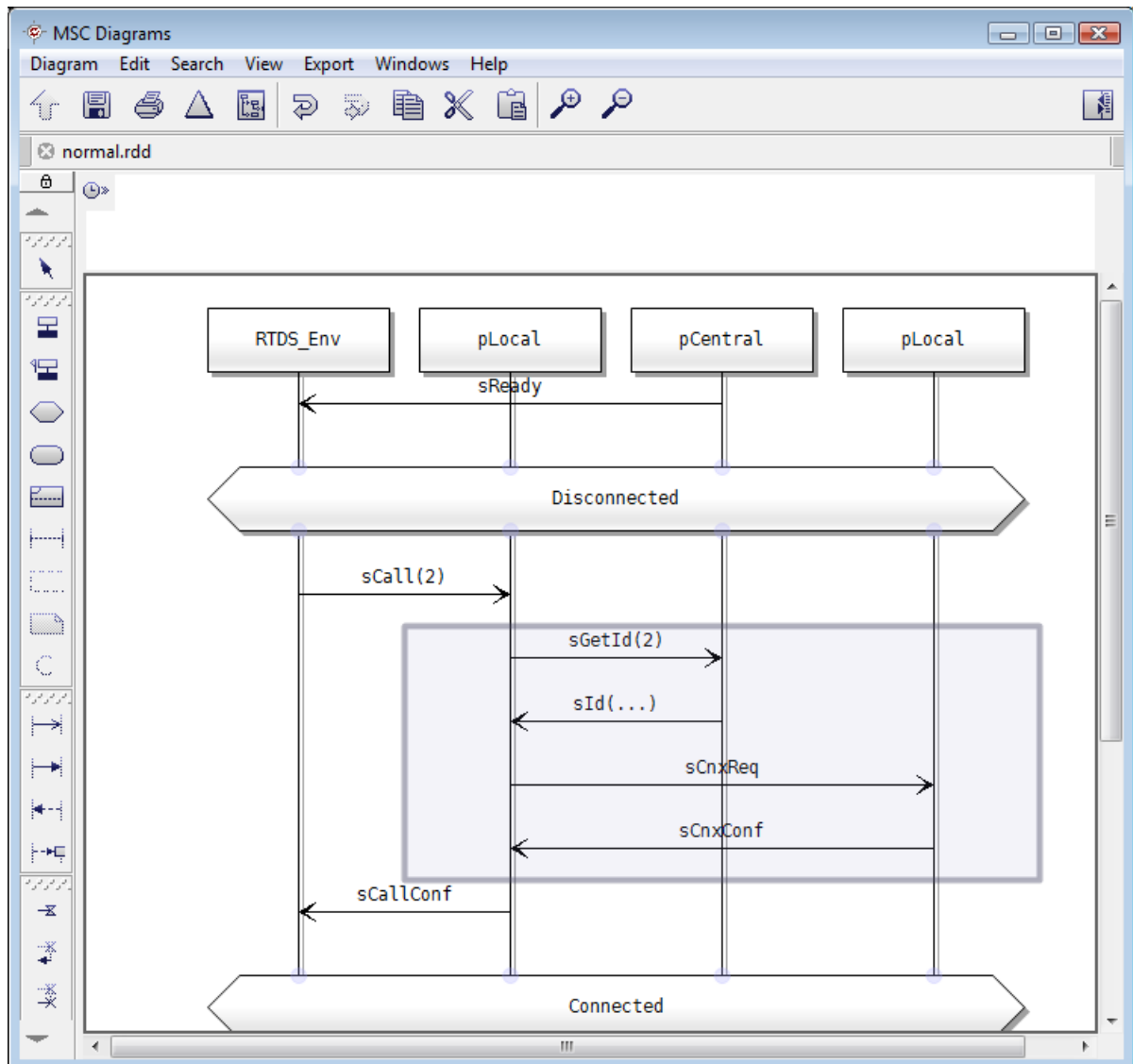


Again, two model element identifiers can be displayed, modified or opened: one for the start of the link if any, and one for its end if any.

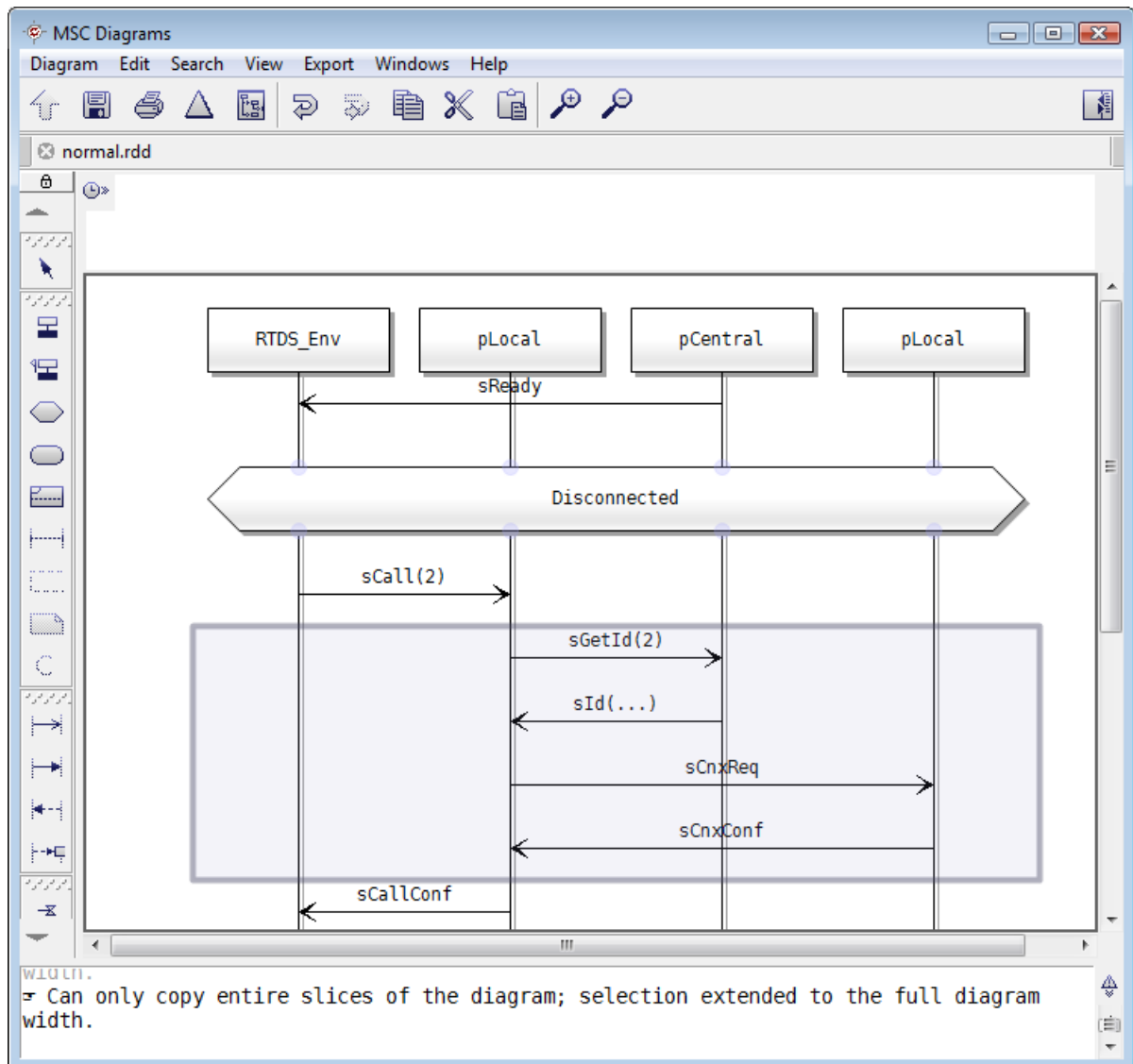
4.4.1.4 Copying or moving parts of the diagram

Copying and moving parts of a MSC diagram are done with the usual copy, cut and paste operations, except these operate on horizontal slices of the diagram. To perform such an operation, select the vertical range for the slice in a rectangular selection, then copy or cut the slice the usual way. The selection will be extended to the full width of the diagram

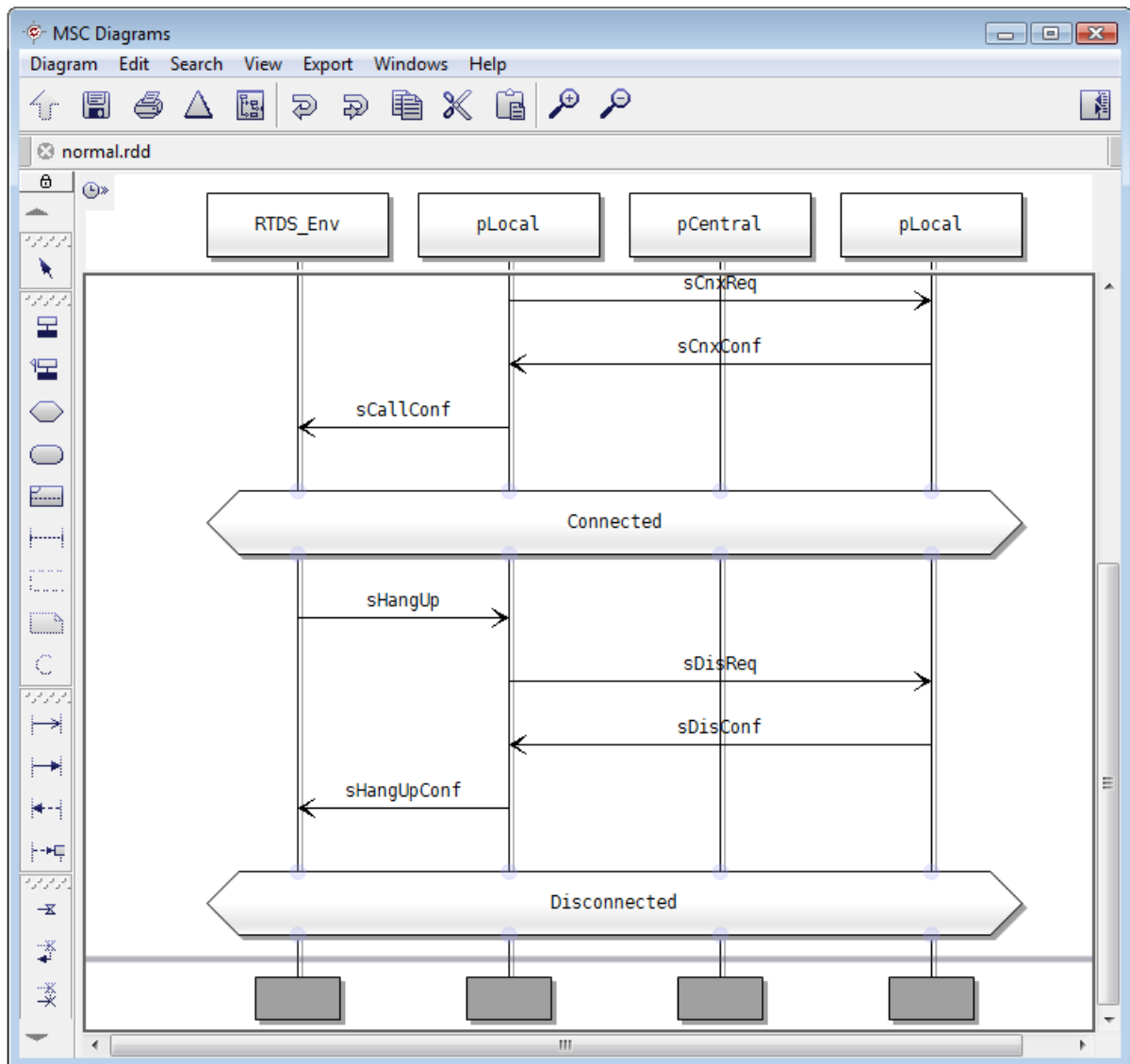
automatically if it isn't already. You can then paste the copied or cut zone at another position in the diagram. For example, if this rectangle is selected:



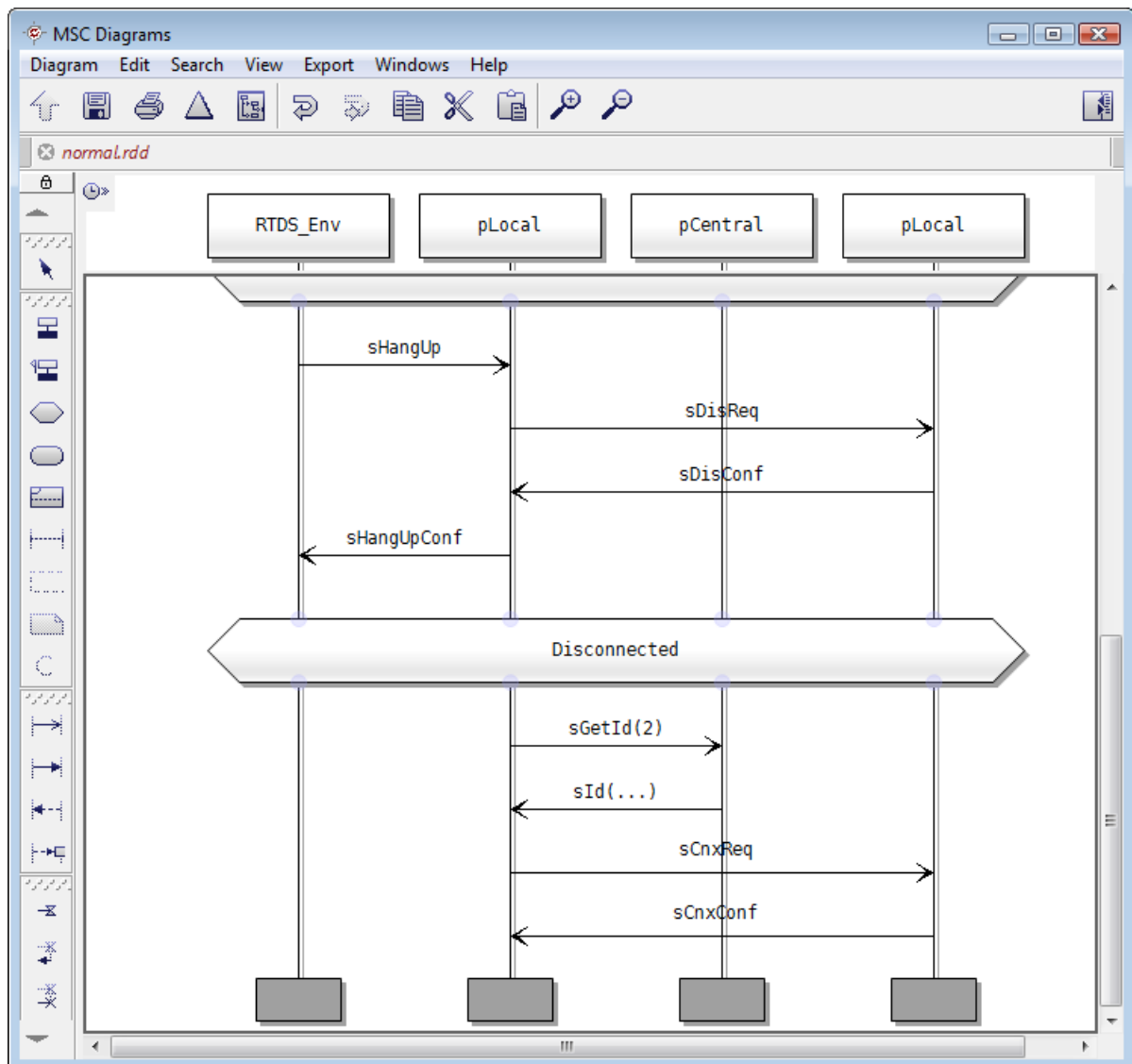
Copying it will extend the selection to the full width of the diagram and display a warning:



Then pasting the copied range at another location in the diagram will display an insertion line:



and clicking in the diagram will paste the copied slice at this position:

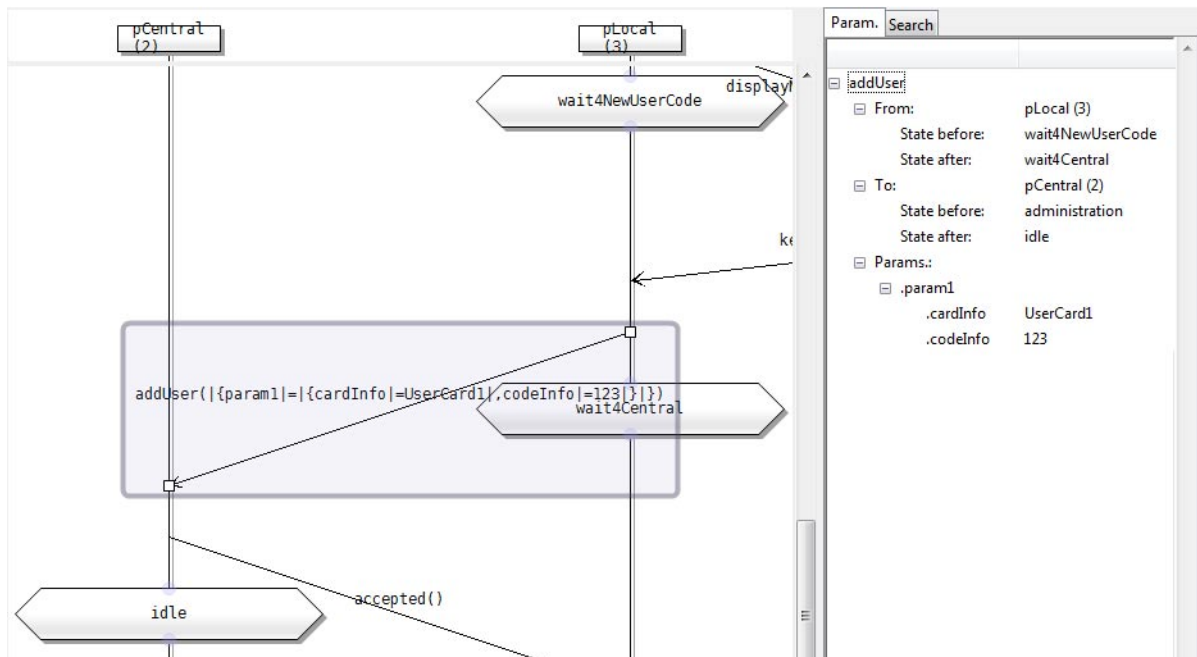


Note that the copy will fail if any object has an end within the slice but the other end outside it, such as a lifeline starting before the slice and ending in it. The paste will fail if one of the copied lifelines does not exist at the paste position.

4.4.1.5 Structured message parameters view

PragmaDev Tracer supports a special syntax allowing to specify structured parameters in message links. This syntax is described in “Message parameter format” on page 67. If the parameters for a message are written in this syntax, the editor window will display them

in a structured tree in the zone at the right of the diagram. The tree will pop out whenever a message link is selected in the editor:

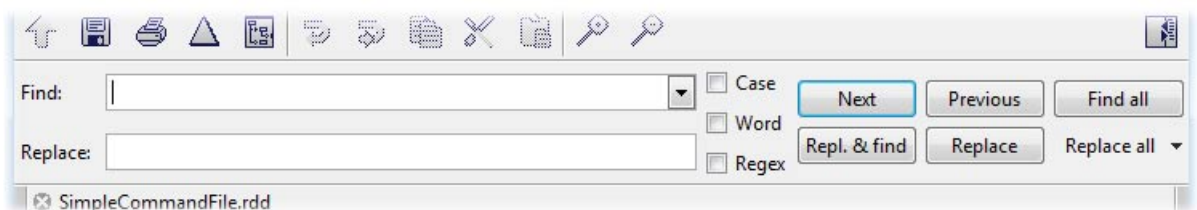


In addition to the message name and parameters themselves, the tree also displays the full text for the sender and receiver lifelines, as well as their states before and after the message was sent or received.

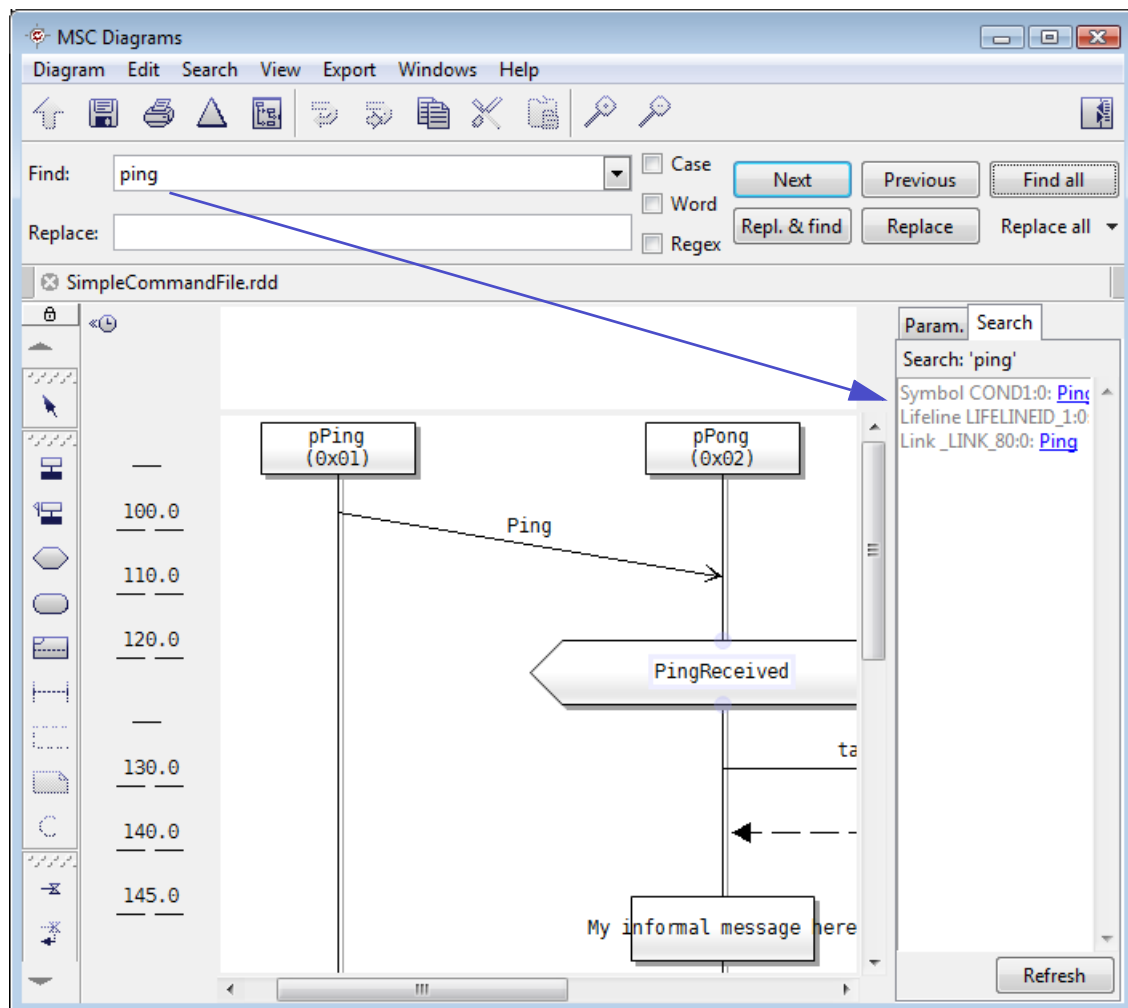
Note that it is not yet possible to directly modify the parameters in the structured view. They have to be modified in the text for the link.

4.4.1.6 Finding and replacing text

Finding and replacing text in the current diagram is done via the 'Find / replace...' item in the 'Search' menu, that displays the find and replace bar at the top of the diagram window:



The menu attached to the 'Find' field recalls previous searches. The buttons allow to perform the usual operations: find next, find previous, replace, replace and find. The 'Find all' button will display all occurrences of the searched text in the side bar of the editor:

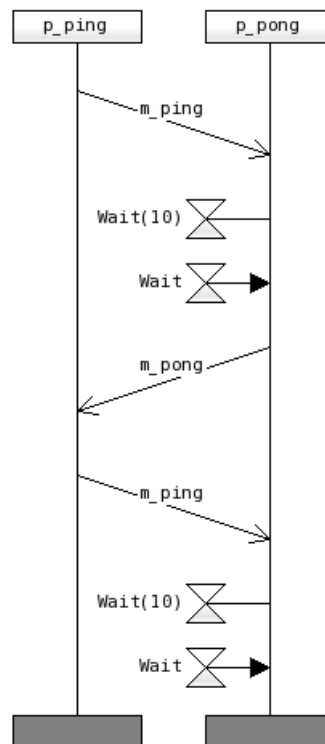


The menu attached to the 'Replace all' button allows to replace all occurrences of the searched text in the whole diagram. Please note replacing text in the current selection is not supported today in *PragmaDev Tracer*.

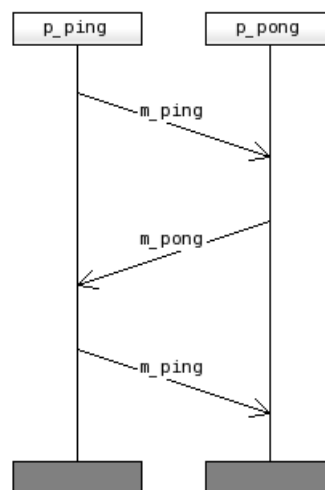
4.4.1.7 Filtering diagrams

When diagrams get big, it can be sometimes interesting to hide some kind of information to be able to focus on the important part. For example, at a given moment, it can be interesting to only see the messages exchanged by the lifelines without having other information such as timers or operation calls. For this, *PragmaDev Tracer* allows to hide one or several kinds of information in a diagram via an operation called *filtering*.

Turning some filters on in a diagram is done by selecting the information you want to hide in the 'Filter' submenu of the 'Diagram' menu. Today, only timers, messages and operation calls and returns can be hidden. For example, for a diagram like this one:



filtering out the timers results in a diagram looking like this:




Note that filtering is strictly a visual operation. The filtered out items are hidden, but they are still there, and unchecking them in the 'Filter' submenu will display them back. The applied filters are recorded in the diagram file though, but only if the *PragmaDev Tracer* XML file format is used.

4.4.1.8 Lifeline collapsing and expanding

As filtering allows to hide specific information, it is sometimes needed to hide what's happening between specific lifelines. For example, several instances can be tasks in the same entity, and it can be practical to see what's happening to this entity rather than to each individual instance.

PragmaDev Tracer allows this kind of operation via *lifeline collapsing*: several lifelines can be collapsed in a single one, all events happening on these lifelines disappearing, as well as the events happening between the collapsed lifelines. An example of collapsing is given in paragraph "Collapsed lifelines" on page 15. To get the second diagram, the lifelines B and C must be selected, then collapsed via the 'Collapse lifelines' operation in the contextual menu opened by a right click.

4.4.1.9 Saving diagrams

Saving a diagram can be done by selecting the 'Save' item in the 'Diagram' menu or by clicking the  button in the top toolbar. The file format will be chosen depending on the original file format for the opened diagram:

- If the opened diagram is in the *PragmaDev Tracer* XML format or in the *RTDS* or *MSC Tracer* XML format, it will be saved in the *PragmaDev Tracer* XML format;
- If the opened diagram is in the MSC-PR format, it will be saved in the MSC-PR format. Note that this may cause loss of information, as some features of *PragmaDev Tracer* are not supported in this format (e.g PSC-specific symbols, diagram filtering or lifeline collapsing).
- If the opened diagram is in Open Trace format, saving it is not supported yet.

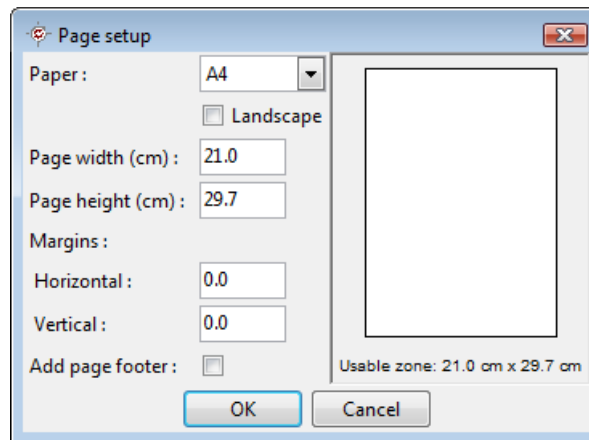
A diagram can be explicitly saved in MSC-PR file format by using the item 'Export as PR...' in the 'Diagram' menu.

4.4.2 Generating documentation: printing and exporting

Generating documentation can be done via printing a whole diagram in a set of pages, or export diagrams fully or partially in image files that can be included in word processing software.


4.4.2.1 Page setup and printing

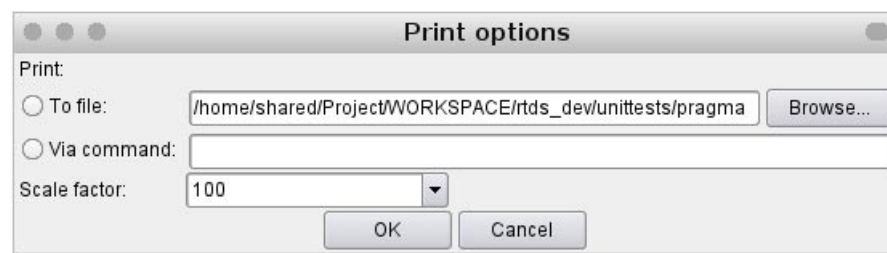
Configuring the page size for printing diagrams is done via the 'Page setup...' item in the 'Diagram' menu. The following dialog then appears:



The page size can be configured either by selecting a standard page size in the 'Paper' field, then optionally rotating the page via the 'Landscape' checkbox, or by directly entering the page size in the 'Page width' and 'Page height' fields.

If some margins are required around the page, they can be configured in the corresponding fields. A page footer giving the name of the diagram and a page number can also be added by checking the 'Add page footer' option. A preview of the actual drawing zone on the printed page is visible on the right side of the dialog, as well as the dimensions for the usable zone.

Once the page setup is done, the diagram can be printed via the 'Print diagram...' item in the 'Diagram' menu, or via the  button. If on Windows, a standard print dialog will appear. If not, the following dialog appears:



If the option 'To file' is selected, the diagram will be printed in the given file in Postscript format. If the option 'Via command' is checked, it will be exported to a temporary Postscript file, and the given command will be run passing that file as parameter. The option 'Scale factor' allows to reduce or expand the printed diagram with the given percentage.

4.4.2.2 Exporting diagrams as image files

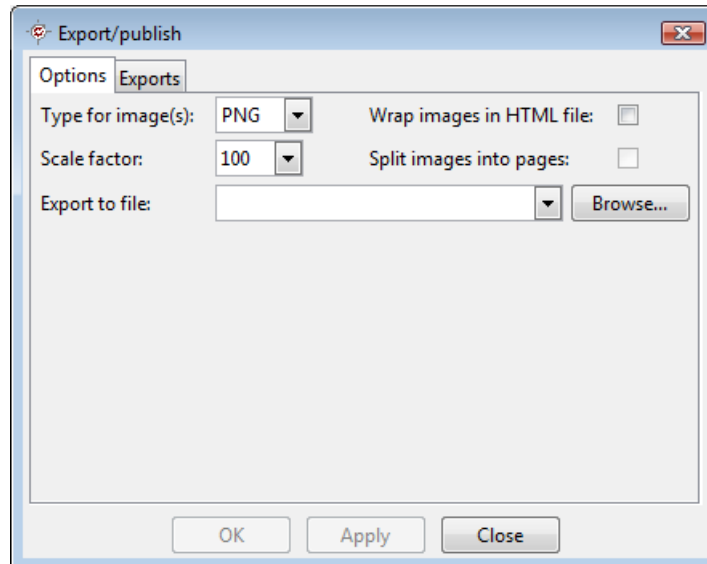
All diagrams can be fully or partially exported as image files, allowing to insert the generated images in a word processor for documentation purposes. The supported image file formats are:

- PNG;
- JPEG;

- EPS (Encapsulated Postscript);
- CGM, which is an ISO-standardized vector-based format.

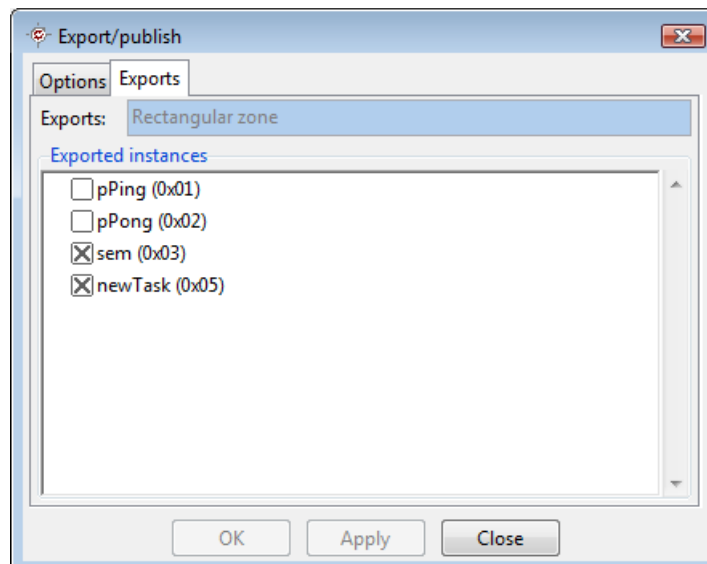
Exporting is done via the 'Export' menu, the 'Export/publish selection...' item exporting the current selection, and the 'Export/publish diagram...' exporting the whole diagram. Only selections consisting in a set of lifelines or a rectangular selection can be exported.

Both menu items open the following dialog:



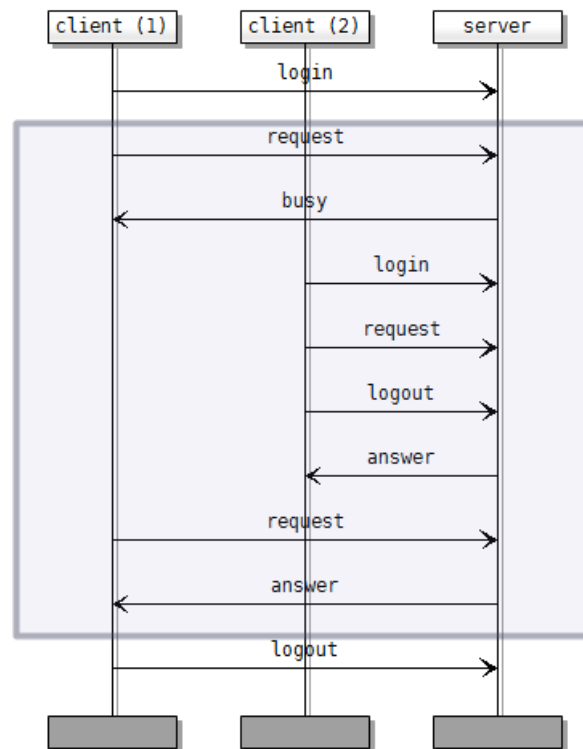
The options for the export include the image file format, the scale factor, whether the exported zone should be split into pages and if the generated image(s) should be wrapped in a HTML file. The name of the file to export to is set in the 'Export to file' field.

The 'Exports' tab allows to show and/or edit the items being exported:

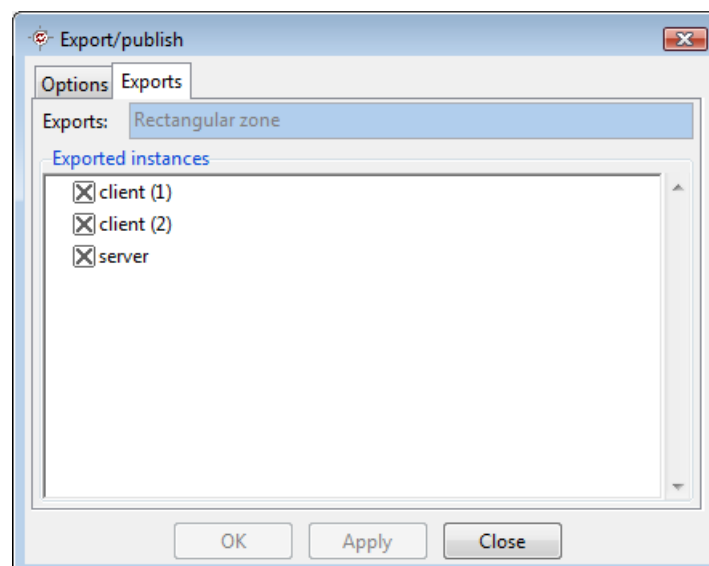


The 'Exports' field indicates the type of export: 'Whole partition' for the whole diagram, 'Selected symbols' for a set of lifelines or 'Rectangular zone' for a rectangular selection.

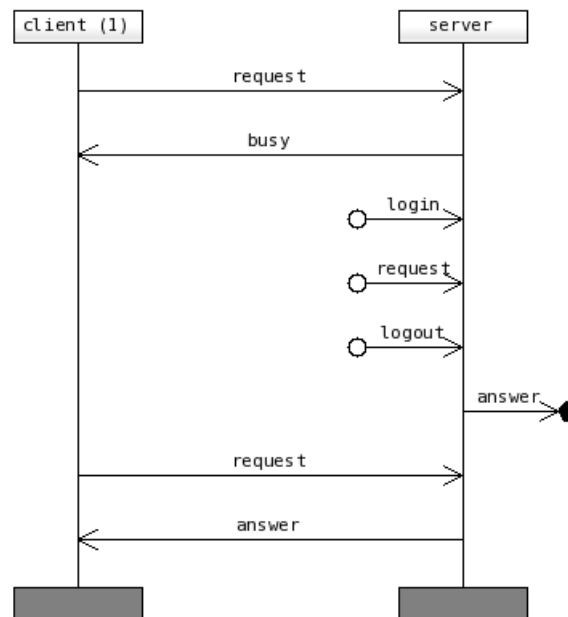
The checkboxes in 'Exported instances' allow to show and optionally modify the exported lifelines. This typically allows to export only part of a rectangular zone. For example, in this diagram:



If only the interaction between client (1) and server in the rectangular zone should be exported, it is possible to ask for the export on the rectangle shown, then go to the 'Exports' tab, which will show:



If client (2) is unchecked here, the exported image file will be:



4.4.3 Conformance checking: diagram diff & property match

PragmaDev Tracer offers 3 levels of conformance checking:

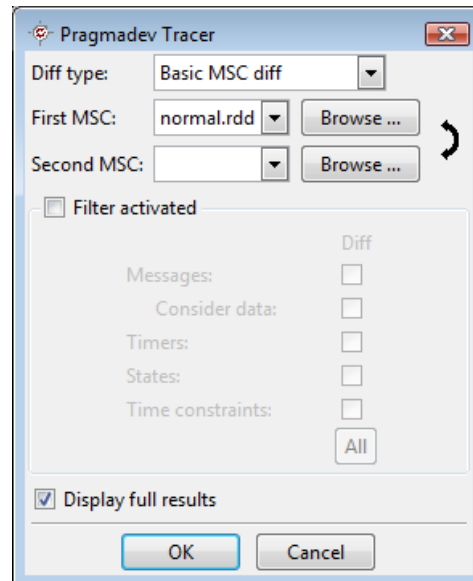
- A MSC trace can be compared to another MSC trace, used as a reference. This can typically be used for regression testing, the reference trace giving the wanted behavior, and being compared to a newly obtained trace.
In this kind of comparison, all events in both diagrams are compared one by one without any interpretation of any kind. This is mainly intended for trace comparisons, but it also works on other diagram kinds, as items normally only present in specification or PSC diagrams are taken into account too, e.g inline expressions or relative time constraints.
- A MSC trace can be compared to a specification diagram. For this comparison, the semantics in the specification is taken into account. For example, if there is an 'opt' inline expression in the specification containing a sequence of message exchanges, the comparison will interpret it, and consider that the diagrams are matching if the sequence is there, or if it is not there at all.
This allows to describe expected scenarios in a powerful way via specification MSC diagrams and match the execution traces against them later.
- Occurrences of a property described in a PSC diagram can be found in a MSC trace. In this case, the semantics are considered in the PSC diagram, as well as the PSC specific elements. Note that this is different from a specification vs. trace comparison, as properties describe a small part of a scenario that can actually match several times in a trace. MSC specification diagrams describe a whole scenario, and will be matched entirely on the trace.
Properties are a good and powerful way to specify wanted and unwanted behavior in the designed system.

Important note: the current implementation of the algorithm used for specification vs. trace comparisons and property matches is currently limited in the number of events it can handle after a matched one and before the next one. The current limitation is 200 events in the trace, so if an event matches and the next event that should match is more than 200 events away from it, it won't be found. This limitation will be removed in a future version.

4.4.3.1 Basic MSC diff: trace vs. trace, spec. vs. spec., ...

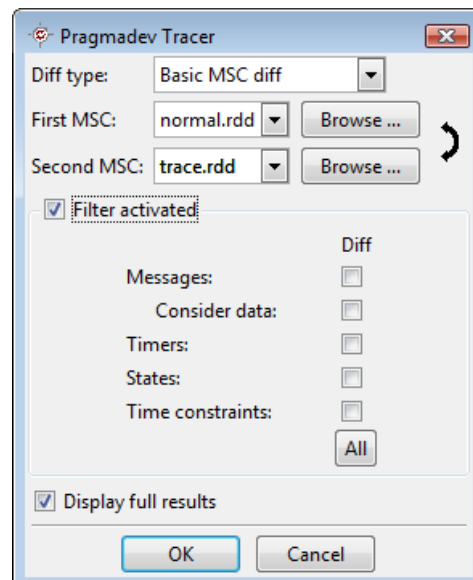
The basic MSC diff just compare two diagrams events by events and reports the found differences. This kind of comparison is launched by selecting 'Compare diagrams...' in

the 'Diagram' menu, or by clicking the  button in the toolbar. The following dialog then appears:



Selecting the basic MSC diff is done by selecting the corresponding value in the 'Diff type' field. The name for first MSC will be automatically set to the name of the currently displayed diagram. For the MSC to compare, it can be either selected in the list attached to the 'Second MSC' field, or loaded from a file via its 'Browse...' button. Once selected, the arrow in the right part of the dialog allows to exchange the two MSCs if the comparison must be done the opposite way.

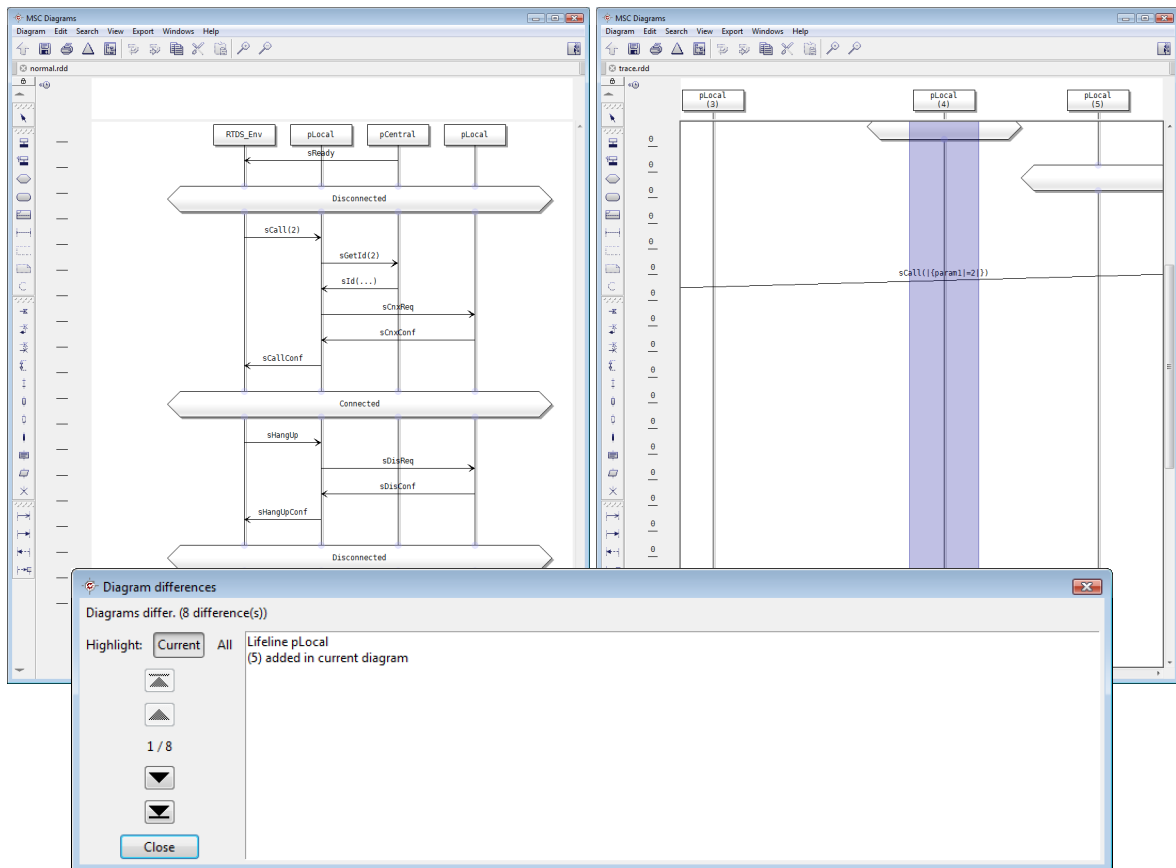
PragmaDev Tracer allows to exclude some elements from the comparison based on their type. This is done by checking the 'Filter activated' option:



All the shown element types can be included or excluded from the comparison. The 'All' button will check all the boxes if any of them is unchecked, and uncheck them if all are checked. The 'Consider data' option allows to compare messages without looking at their actual parameters. If the option is unchecked, only the names of the message are compared and nothing else, even if it is present.

The option 'Display full results' at the bottom of the dialog allows to display only a summary of the comparison results instead of the full set of differences. To display the summary, just uncheck the box.

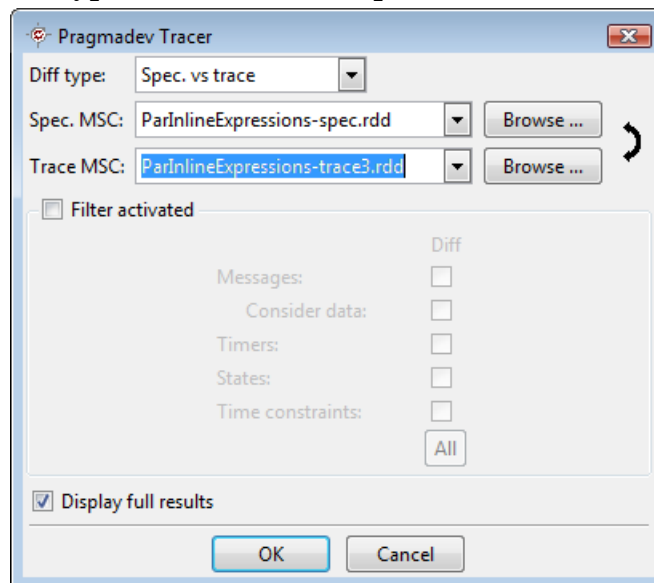
If this option is checked and after validating the dialog, PragmaDev Tracer puts each diagram in its own window and displays them side by side. A dialog also appears at the bottom of the screen, allowing to browse through the found differences:



A summary of the differences is displayed at the top. Each difference will be highlighted in red in the diagram displayed on the left, and in blue in the diagram displayed on the right. The text in the dialog gives a short description of the identified difference. The arrows allow to browse through the differences. The option 'Highlight' allows to highlight all differences in both diagrams to get a quick view of what differs without having to browse through all the differences.

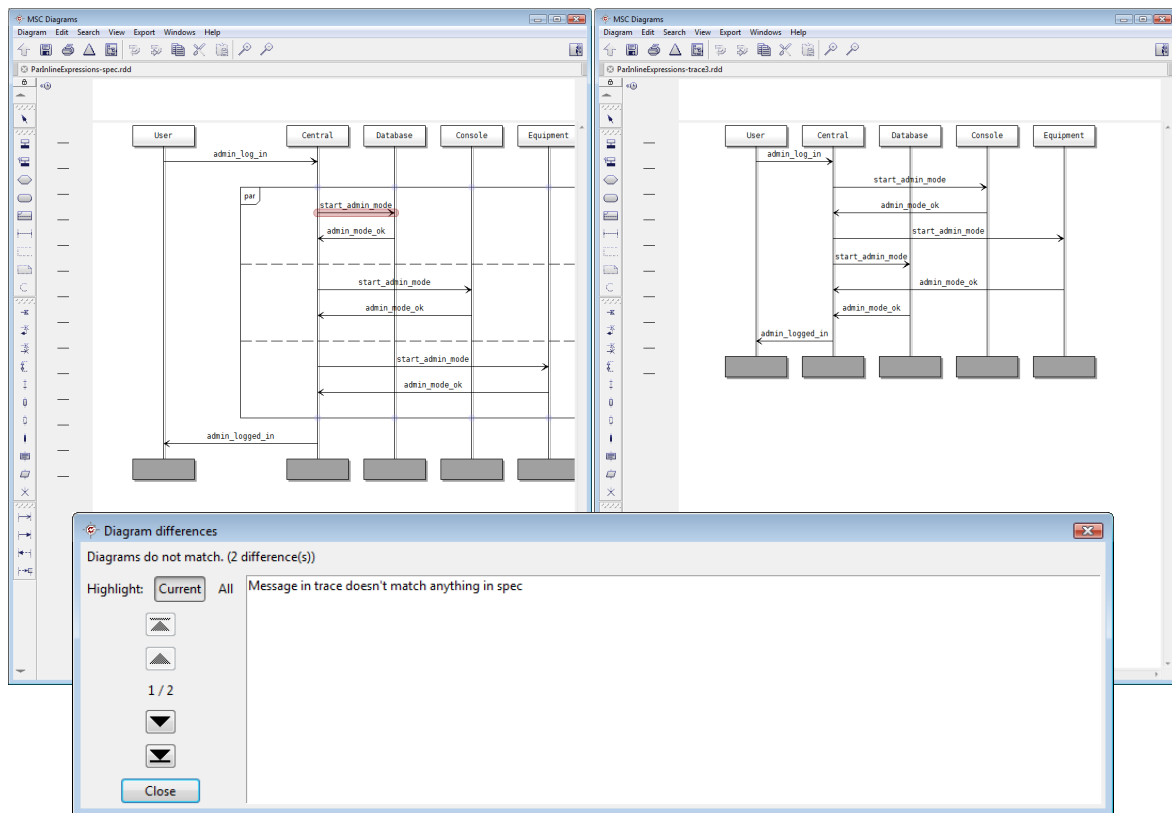
4.4.3.2 Spec vs. trace comparison

Comparing a specification diagram to an actual trace is done the same way as for a basic MSC diff, except the diff type has to be set to 'Spec. vs. trace' in the dialog:



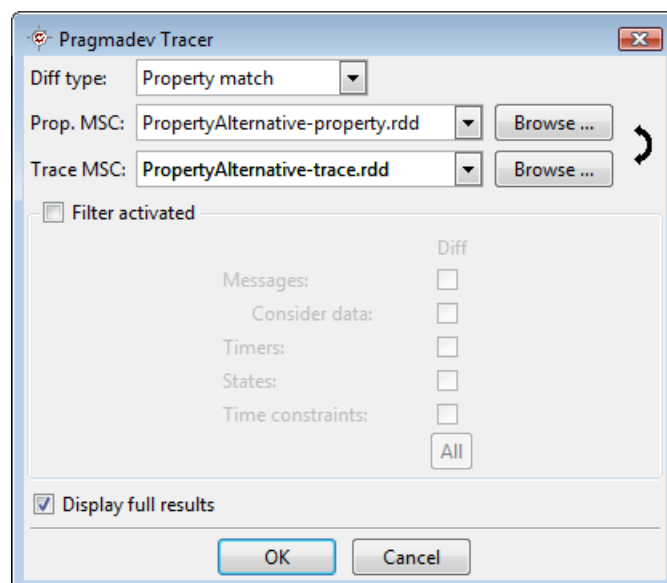
Note also that the specification diagram must be the one specified in the field 'Spec MSC' in the dialog, which is always the first one. If needed, the diagrams can be swapped by using the arrow button on the dialog's right side. The same filters are provided as for a basic MSC diff.

Once validated, the found differences are displayed in the same way as for a basic MSC diff; only the way to perform the comparison changes, as semantics in the specification is taken into account where it wouldn't be in a basic MSC diff:



4.4.3.3 Property match

Matching a PSC diagram against a MSC trace is done the same way as for the other kinds of comparisons, except the diff type has to be set to 'Property match':

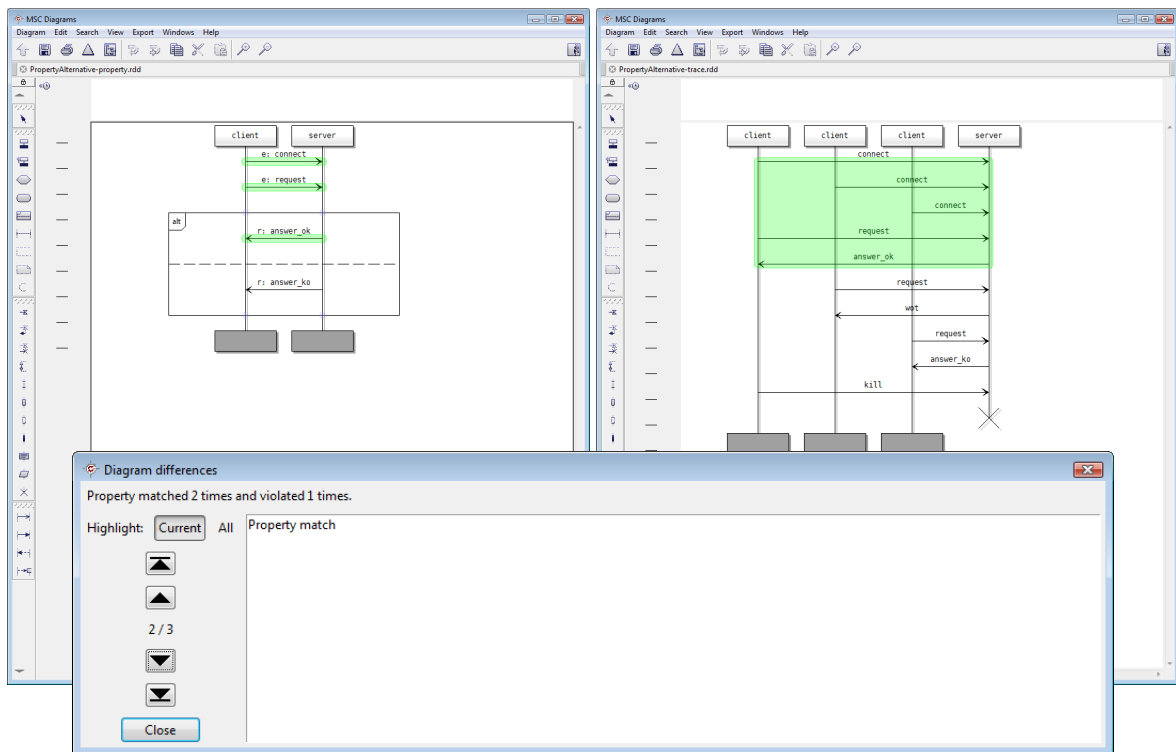


Note that the PSC diagram has to be the one specified in the 'Prop. MSC' field, which is always the first one. If needed, the diagrams can be swapped with the arrow button on

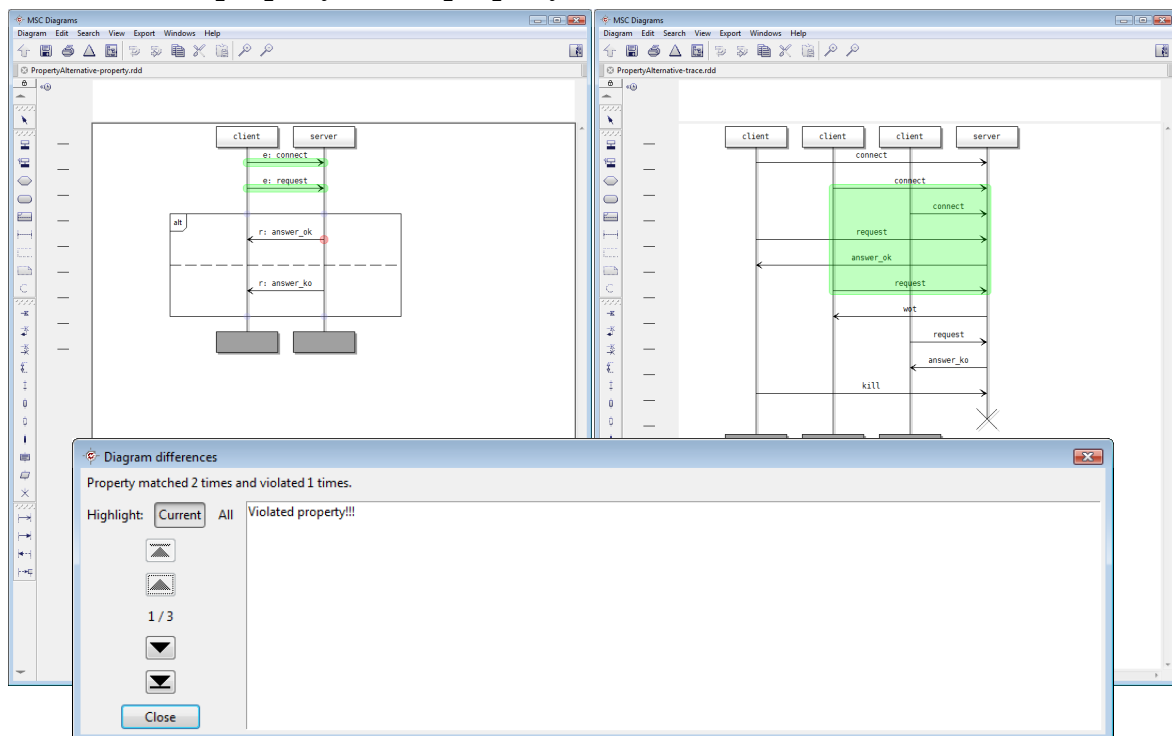
the right side of the dialog. The same comparison options are provided as for basic MSC and specification vs. trace comparisons, but they are less significant here, as a property diagram is always partial.

Once validated, the property matches and violations are displayed in a similar way to the display of differences in the other kinds of comparisons. Mostly the colors and the difference descriptions differ: each matched element in the property or the MSC diagram will be displayed as green, and each unmatched one as red. The difference description will be:

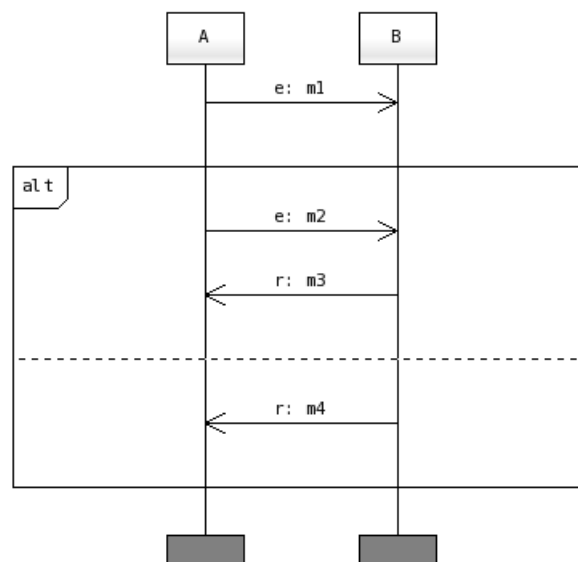
- 'Property match' if the property matches:



- ‘Violated property!’ if the property does not match:



- ‘Possibly violated property’ in some very specific cases where it is impossible to tell if the property is matched or not. A typical example where this case happens is the following:



If the trace contains a message *m1* from A to B, followed neither by a message *m2* from A to B, nor by a message *m4* from B to A, there's no way to know which part of the alternative should have matched. But if it was the first part, the message *m2* is not there, so the property doesn't apply, and if it was the second one, the required message *m4* is not there either, so the property is violated. In this case, a possible property violation will be reported.

Note that a property is not necessarily violated if something doesn't match in it. Typically, an unmatched fail message means the property is matched.

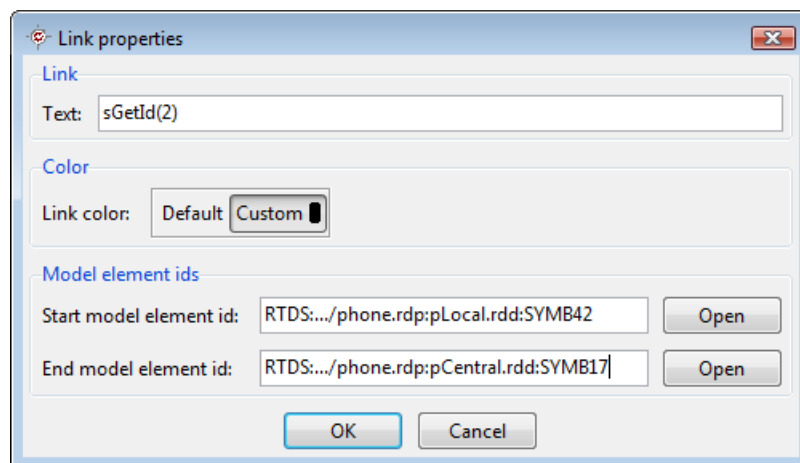
4.5 - Linking with model elements

PragmaDev Tracer allows to link all the events happening in a diagram to the elements in the model that caused the event to happen. For example, a message send can be linked to the corresponding SDL message output symbol, or a condition to the corresponding state symbol in a UML state chart.

For this, PragmaDev Tracer offers several mechanisms:

- During a trace, an event can specify a model element identifier via the option `-M`. This identifier is stored in the produced diagram file. See the description for this option in section “Common options and arguments” on page 63.
- When displaying the trace diagram in the editor, or when creating a new diagram, model element identifiers can be displayed or modified in the properties dialog associated to each object. These dialogs are described in section “Symbol and link properties” on page 35.

Here is an example of a link properties dialog displaying a model element identifier for both ends of a message:



- The properties dialog also allows to open the model element with via the “Open” button. When pressed, PragmaDev Tracer will run the model element opening command specified in the preferences, passing the model element identifier as a parameter: if the command contains the string “{model_id}”, it will be replaced with the identifier; if it doesn’t, the identifier is just added to it after a space. Note that this command may have to be a wrapper script, especially if several modelling tools are used. The wrapper script should then figure out what tool to launch depending on the model element identifier, then launch it with the proper options to display the element.

There are two modes to run the model element opening command, depending on the option “Run command asynchronously” in the tracer preferences:

- If the command is run synchronously (option unchecked), the tracer will wait for the command to exit and display its standard and error output in a dialog if needed. This works only if the actual modelling tool can be run in the background, or the tracer will remain stuck until the tool is exited.
- If the command is run asynchronously (option checked), the tracer will run itself the command in the background. In this case, it is not possible to get the output of the command and display it in a dialog, so it is automatically redirected to a

file, and the name of the file is displayed in an informational message in the editor window.

5 - Command line interface

Specific features of *PragmaDev Tracer* are available via a command line interface via a specific executable, named `pragmatracercommand.exe` on Windows and `pragmatracercommand` on Unix. The general syntax for the command is:

```
pragmatracercommand <subcommand> <options> <arguments>
```

There are today two available subcommands:

- `mscdiff` performs the comparison between two diagrams and displays a summary of the differences between them. This subcommand is described below in “CLI diagram comparison: `mscdiff`” on page 61.
- `help` displays the general help for the tool, or the help for a particular subcommand when followed by the subcommand name; for example:
`pragmatracercommand help mscdiff`
will display the help for the `mscdiff` subcommand.

5.1 - CLI diagram comparison: `mscdiff`

The syntax for the `mscdiff` subcommand is:

```
pragmatracercommand mscdiff <options> <diagram file 1> <diagram file 2>
```

where `<options>` can be:

- `-q` or `--quiet`, which will not print anything but only indicate whether the diagrams are different via the command’s exit status, 0 meaning diagrams match or are identical;
- `-s`, indicating the first diagram should be interpreted as a specification MSC and that its semantics must be taken into account;
- `-p`, indicating that the first diagram is a PSC diagram and that the comparison must actually be a property matching;
- `-x` followed by a set of letters indicate the active filters for the comparison:
 - A ‘m’ in the set will completely exclude messages;
 - A ‘d’ in the set will ignore message data (parameters);
 - A ‘t’ in the set will exclude timers;
 - A ‘s’ in the set will exclude states and condition symbols;
 - A ‘c’ in the set will ignore relative time constraints.

For example, specifying the option `-xtsc` will exclude timers, states/conditions and relative time constraints from the comparison.

If neither `-s` nor `-p` are specified in the options, the comparison is a normal one and will not consider the semantics of either diagram.

The printed summary depends on the type of comparison:

- For a normal comparison, the summary will either indicate that the diagrams are identical or give the number of found differences;
- For a specification vs. trace comparison, the summary will either indicate that the trace matches the specification, or give the number of found differences;
- For a property match, the summary will give the number of positive matches and the number of property violations.

In all cases, the command exit status will indicate if differences were found:

- For a normal comparison, the status is 0 if diagrams are identical, and non-0 if differences were found;
- For a specification vs. trace comparison, the status is 0 if the trace matches the specification, and non-0 if differences were found;
- For a property match, the status is 0 if no property violation was found, and non-0 if any violation was found. Note that possible property violations are considered as actual violations (see “Property match” on page 55 for an example of a possible property violation).

6 - Tracer commands

6.1 - Command reference

PragmaDev Tracer accepts several commands to generate the MSC trace. They represent a specific real-time event illustrated by a symbol in the MSC diagram.

The field separator in a command is:

'| ' (pipe followed by a space)

The command is ended by:

'|\n' (pipe followed by the line feed character i.e. 0x13)

To insert a printable pipe in the command, another pipe must be put up front:

'||' => printable '|'

For example, the command for a task creation named pPing with pid 1 will be:
 taskCreated| -npPing| 0x01|\n

The generic syntax of a command is:

<command_name or alias>[| <options>]| <parameters>| <optional_arguments>|\n

If a command name or alias is not recognized, the whole command is ignored.

If one of the parameters is not provided, the whole command is ignored.

If a provided option does not exist, the whole command is ignored.

If the number of optional arguments provided is greater than the expected number, the command interpreter will ignore the additional arguments.

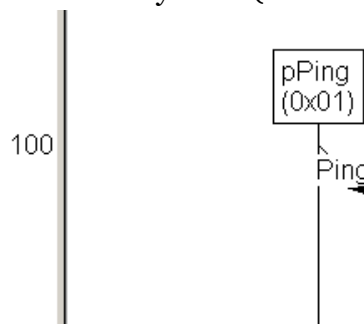
Each command has an alias; e.g.: ms has the same meaning as messageSent.

6.1.1 Common options and arguments

The following options and arguments may appear in several commands:

- -t<time>

Represents the time when an event occur; if provided, a symbol containing the value of time of the associated event will appear at the left of the MSC trace at the same height that the event symbol (task created, message sent...) involved.

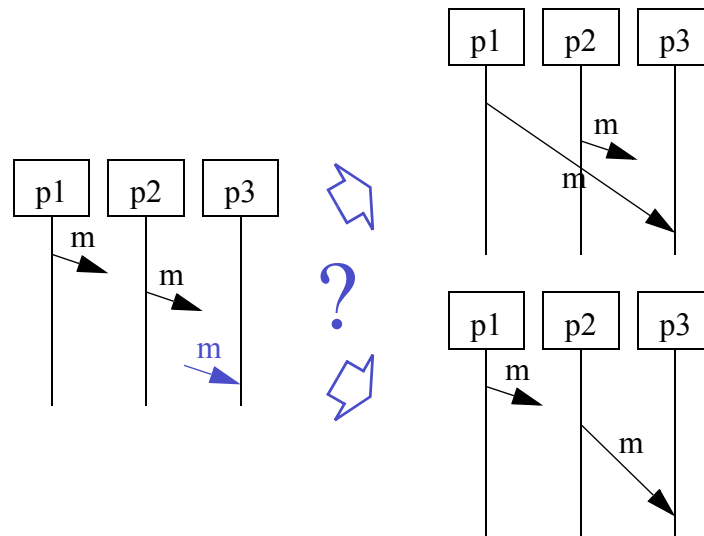


Here the sending of message ping from process pPing occurs at time 100 after the beginning of system execution.

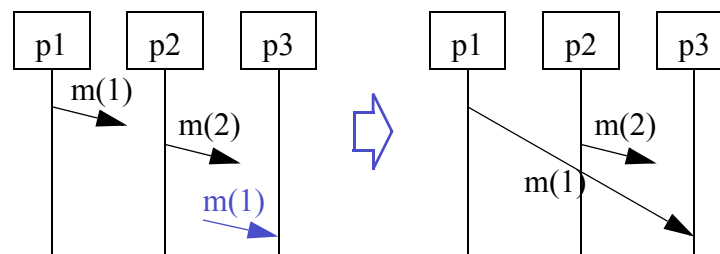
- `-i<mId>`

Represents a unique identifier for a given message. It comes with the commands `messageSend`, `messageReceive`, and `messageSave`. This option provides a unique identifier for the sending of a message, allowing to match a message receive to the corresponding message send. If this option is not used, a message receive will always suppose that the received message is the last message sent with the same name. This may result in errors in the trace if several messages with the same name exist at the same moment.

For example:



Without message unique ids



With message unique ids (after message name)

- `-M<elt_id>`
Specifies the identifier for the model element for this event. The identifier is a string that is only meaningful for the modelling tool used to create the element. The identifier is not displayed, but is simply stored in the trace diagram once this one is saved.
- `<sigNum>`
Numerical identifier for the name of a message. Note both the name and this identifier may have to be provided.
- `<pId>` or `<semId>`
Identifier of a process or semaphore; it is an unique identifier. `pId` and `semId` represents the id of the lifeline, so it is not possible to have a semaphore and a process with the same id.
- `<tId>`

unique timer identifier; it must be provided, and may be used as the name of the timer if no option -T is provided.

6.1.2 Task creation

To trace a task creation, the command syntax is:

```
taskCreated[| -t<time>][| -M<elt_id>]
           [| -c<creatorId>][| -n<pName>][| -N<creatorName>]| <pId>|\n
```

or

```
pc[| -t<time>][| -M<elt_id>][| -c<creatorId>][| -n<pName>][| -N<creatorName>]
  | <pId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -c: identifier of creator process,
- -N: name of creator process,
- -n: name of created process; if this option is not present, the process name on the MSC trace will be its id.

parameters:

- <pId>: unique process identifier.

6.1.3 Task deletion

To trace a task deletion, the command syntax is:

```
taskDeleted[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>|\n
```

or

```
pd[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: process name,

parameters:

- <pId>: unique process identifier.

6.1.4 Messages

6.1.4.1 Message sending

To trace a message sending from a process, the command syntax is:

```
messageSent[| -t<time>][| -M<elt_id>][| -d<msgData>][| -n<pName>][| -i<mId>]
           | <pId>| <sigNum>| <msgName>|\n
```

or

```
ms[| -t<time>][| -M<elt_id>][| -d<msgData>][| -n<pName>][| -i<mId>]
  | <pId>| <sigNum>| <msgName>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -d: data of the message. Spaces are allowed and the data format is free except for the ‘|’ character that should be doubled: ‘||’. Detailed format is described in “Message parameter format” on page 67.
- -n: name of the process sending the message.
- -i: unique identifier for message, See “Common options and arguments” on page 63.

parameters:

- <pId>: unique process identifier.
- <sigNum>: signal number of the message, See “Common options and arguments” on page 63.
- <msgName>: message name.

6.1.4.2 Message reception

To trace a message reception, the command syntax is:

```
messageReceived[| -t<time>][| -M<elt_id>]  
[| -d<msgData>][| -n<pName>][| -i<mId>]| <pId>| <sigNum>| <msgName>|\n
```

or

```
mr[| -t<time>][| -M<elt_id>][| -d<msgData>][| -n<pName>][| -i<mId>]  
| <pId>| <sigNum>| <msgName>|\n
```

options:

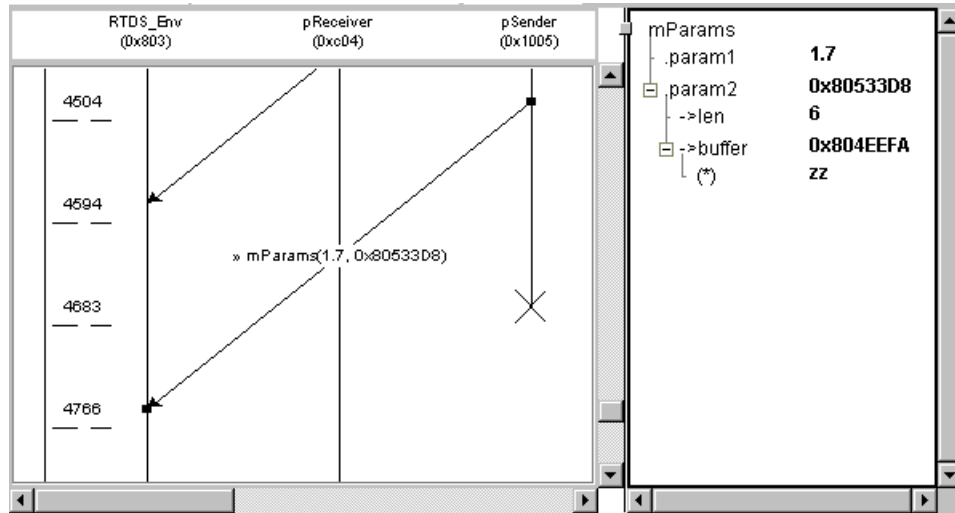
- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -d: data of the message. Spaces are allowed and the data format is free except for the ‘|’ character that should be doubled: ‘||’. Detailed format is described in “Message parameter format” on page 67.
- -n: name of the process receiving the message.
- -i: unique message identifier, See “Common options and arguments” on page 63.

parameters:

- <pId>: unique process identifier.
- <sigNum>: signal number of the message, See “Common options and arguments” on page 63.
- <msgName>: message name.

6.1.4.3 Message parameter format

The format described below applies to structured parameters in messages sent or received. Sending structured messages with the following format allows to display parameters in a tree in the MSC editor as shown below.



The format for this text or argument depends on whether the message is structured or not. Structured parameters are fully described in RTDS Reference Manual. In short, a message is structured if and only if it is declared with several parameters or with one parameter that is a pointer to a struct or a union.

- For a non-structured message, the text for the parameter must be a sequence of bytes written in hexadecimal format, exactly as they will appear in the target program memory.
- For a structured message, the text for the parameter must be written as follows:
 - The values for base types are written as in C: for example 12 or 871 are valid values for an int, X is a valid value for a char, and so on...
 - The values for pointers are written in hexadecimal, optionally prefixed by 0x, and followed by | : and the pointed value. For example, for an int*:
 - 804A51FE | : 67 will define the pointer to be 0x804A51FE and 67 will be the pointed value;
 - | : 123 will only describe the pointed value to be 123;

There is a special case for char* pointers: the value can be a full string instead of just a single char. Please note all ' ' characters must be doubled in this string.

- The values for structs or unions are coded as follows:

```
{field1|=value|,field2|=value|,...|}
```

For example, for a struct defined as:

```
struct MyStruct { int i; char *s; };
```

a valid format is:

```
{i|=4|,s|=|:abcd|}
```

In the struct, the field i will be set to 4 and the field s will point to the "abcd".

6.1.4.4 Message saving

To trace a saved message, the command syntax is:

```
messageSaved[] -t<time>[] -n<pName>[] -i<mId>[] <pId>| <sigNum>|  
| <msgName>|\n
```

or

```
mv[] -t<time>[] -n<pName>[] -i<mId>[] <pId>| <sigNum>| <msgName>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of process that saves the message,
- -i: unique message identifier, See “Common options and arguments” on page 63.

parameters:

- <pId>: unique process identifier,
- <sigNum>: signal number for the message, See “Common options and arguments” on page 63.
- <msgName>: saved message name

6.1.5 Semaphore creation

To trace a semaphore creation, the command syntax is:

```
semaphoreCreated[] -t<time>[] -M<elt_id>[] -s<semName>[] -c<creatorId>|  
| -N<creatorName>[] -a<stillAvailable>[] <pId>|\n
```

or

```
sc[] -t<time>[] -M<elt_id>[] -s<semName>[] -c<creatorId>|  
| -N<creatorName>[] -a<stillAvailable>[] <pId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -s: name of semaphore; if not provided the name of the created semaphore in the MSC trace will be its internal identifier,
- -c: identifier of the creator process,
- -N: name of the creator process,
- -a: boolean indicating if semaphore is empty (value 0) or full (value 1),

parameters:

- <pId>: unique identifier of the created semaphore.

6.1.6 Semaphore deletion

To trace a semaphore deletion, the command syntax is:

```
semaphoreDeleted[] -t<time>[] -M<elt_id>[] -s<semName>[] -c<destructorId>|  
| -N<destructorName>[] <pId>|\n
```

or

```
sd[] -t<time>[] -M<elt_id>[] -s<semName>[] -c<destructorId>|  
| -N<destructorName>[] <pId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -s: name of deleted semaphore,
- -c: identifier of the destructor process,
- -N: name of the destructor process,

parameters:

- <pId>: unique identifier of the deleted semaphore.

6.1.7 Semaphore take attempt

To trace an attempt to take semaphore, the command syntax is:

```
takeAttempt[| -t<time>][| -M<elt_id>]
[| -n<pName>][| -s<semName>][| -T<timeout>]| <pId>| <semId>|\n
```

or

```
sa[| -t<time>][| -M<elt_id>][| -n<pName>][| -s<semName>][| -T<timeout>]
| <pId>| <semId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of process taking the semaphore,
- -s: name of the taken semaphore,
- -T: timeout for the take; a value of -1 indicate that the process will wait until the take have succeeded,

parameters:

- <pId>: identifier of taker process,
- <semId>: identifier for taken semaphore.

6.1.8 Semaphore take succeeded

To trace a semaphore has been successfully taken, the command syntax is:

```
takeSucceeded[| -t<time>][| -M<elt_id>]
[| -n<pName>][| -s<semName>][| -a<stillAvailable>]| <pId>| <semId>|\n
```

or

```
ss[| -t<time>][| -M<elt_id>][| -n<pName>][| -s<semName>][| -a<stillAvailable>]
| <pId>| <semId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of the process taking the semaphore,
- -s: name of taken semaphore,
- -a: boolean indicating if semaphore is empty (value 0) or full (value 1),

parameters:

- <pId>: identifier for taker process,
- <semId>: identifier for taken semaphore.

6.1.9 Semaphore take timed out

To trace a semaphore take attempt timed out, the command syntax is:

```
takeTimedOut[] -t<time>[] -M<elt_id>[] -n<pName>[] -s<semName>[]  
| <pId>| <semId>|\n
```

or

```
st[] -t<time>[] -M<elt_id>[] -n<pName>[] -s<semName>[] <pId>| <semId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of taker process,
- -s: name of taken semaphore,

parameters:

- <pId>: identifier of taker process,
- <semId>: identifier of taken semaphore.

6.1.10 Semaphore give

To trace a semaphore give, the command syntax is:

```
giveSem[] -t<time>[] -M<elt_id>[] -n<pName>[] -s<semName>[]  
| <pId>| <semId>|\n
```

or

```
sg[] -t<time>[] -M<elt_id>[] -n<pName>[] -s<semName>[] <pId>| <semId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of the process giving the semaphore,
- -s: name of given semaphore,

parameters:

- <pId>: identifier of giver process,
- <semId>: identifier of given semaphore.

6.1.11 Timer start

To trace the start of a timer, the command syntax is:

```
timerStarted[] -t<time>[] -M<elt_id>[]  
| -n<pName>[] -T<timerName>[] <pId>| <tId>[] <timeLeft>[]|\n
```

or

```
ts[] -t<time>[] -M<elt_id>[] -n<pName>[] -T<timerName>[]  
| <pId>| <tId>[] <timeLeft>[]|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of process starting the timer,
- -T: name of the timer started,

parameters:

- <pId>: identifier of starter process,
- <tId>: identifier of timer, See “Common options and arguments” on page 63.

optional argument:

- <timeLeft>: time before timer times out.

6.1.12 Timer cancellation

To trace the cancellation of a timer, the command syntax is:

```
timerCancelled[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>]
| <pId>| <tId>|\n
```

or

```
tc[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>] <pId>| <tId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of process having starting the timer,
- -T: name of the stopped timer,

parameters:

- <pId>: process identifier,
- <tId>: identifier of timer stopped, See “Common options and arguments” on page 63.

6.1.13 Timer timed out

To trace a timed out timer, the command syntax is:

```
timerTimedOut[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>]
| <pId>| <tId>|\n
```

or

```
tt[| -t<time>][| -M<elt_id>][| -n<pName>][| -T<timerName>] <pId>| <tId>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of process having started the timer,
- -T: name of the timer that times out ,

parameters:

- <pId>: process identifier,
- <tId>: identifier of the timer that times out, See “Common options and arguments” on page 63.

6.1.14 Task state changed

To trace a task state has changed, the command syntax is:

```
taskChangedState[| -t<time>][| -M<elt_id>][| -n<pName>] <pId>| <stateName>|\n
```

or

```
ps[| -t<time>][| -M<elt_id>][| -n<pName>] <pId>| <stateName>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of process involved,

parameters:

- <pId>: process identifier,
- <stateName>: name of the new state for process.

6.1.15 Action symbol

To trace an action in a lifeline, the command syntax is:

```
information[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>| <message>|\n
```

or

```
in[| -t<time>][| -M<elt_id>][| -n<pName>]| <pId>| <message>|\n
```

options:

- -t, -M: time and model element identifier for the event; See “Common options and arguments” on page 63.
- -n: name of process involved,

parameters:

- <pId>: process identifier,
- <message>: information to be displayed in the action symbol.

6.1.16 Start a new MSC trace

To start a new MSC trace, the command syntax is:

```
newTrace[| -f<fileName>]|\n
```

or

```
n[| -f<fileName>]|\n
```

options:

- -f<fileName>: file name is the file name of the new trace.

In graphical mode, this command closes the current trace if it exists, saves it if needed and launches a new viewer window. In no window mode, the previous trace is stopped and saved and a new trace starts, ready to accept commands.

6.1.17 Pause MSC trace

To pause the current MSC trace, the command syntax is:

```
pause|\n
```

or

```
p|\n
```

The trace is resumed with the *resume* command.

6.1.18 Resume MSC trace

To resume the current MSC trace, the command syntax is:

```
resume|\n
```


or
r|\n

This command is used only after the trace has been paused via the graphical interface or with the *pause* command.

6.1.19 Close MSC trace

To stop the MSC trace, save and close the file, the command syntax is:

close|\n

or
c|\n

Use to indicate that the current trace will be definitively stopped and then saved. If a file name has been provided when the current trace started, it will be used to save the diagram; otherwise *PragmaDev Tracer* will generate a file name like "mscTracer.rdd" in no window mode or open a dialog asking to choose a file name.

6.1.20 Exit PragmaDev Tracer

Exit *PragmaDev Tracer*. If a trace was running, it will be stopped and saved as if a command close was sent.

exit|\n

or
e|\n

6.1.21 Set directory

To set the default directory, the command syntax is:

setDirectory| <dir>|\n

or
dr| <dir>|\n

parameters:

- <dir>: default directory.

Used to save traces when no file name is provided or when a file name without an entire path is given.

6.1.22 Acknowledgment

In order to be sure all commands sent have been received, it is possible to ask for an acknowledgment from *PragmaDev Tracer*. The command syntax is:

waitingForAck|\n

or
wa|\n

The acknowledgment sent back after receiving the waitingForAck command is the string:
ack

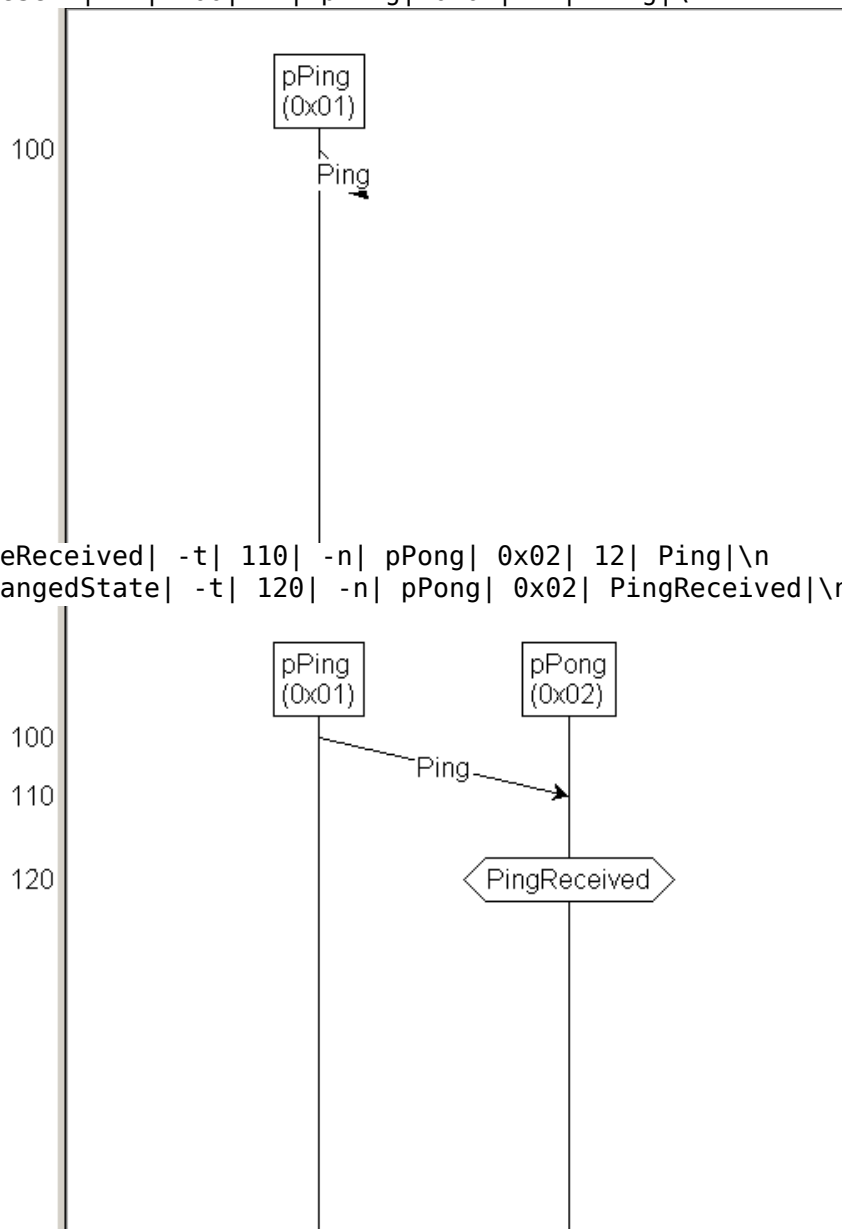
6.2 - Tracing example

This example will illustrate the use of the tracing feature of *PragmaDev Tracer*.

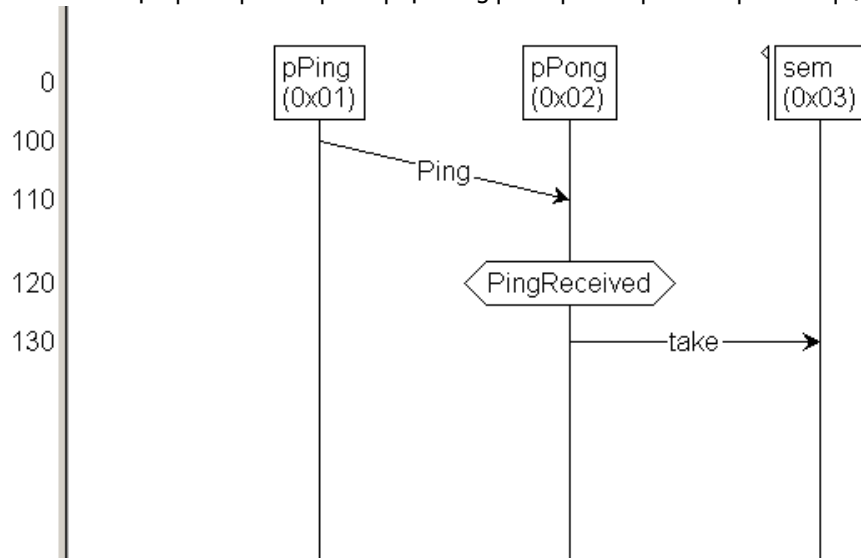
Commands sent:

- taskCreated| -n| pPing| 0x01|\n
- messageSent| -t| 100| -n| pPing| 0x01| 12| Ping|\n

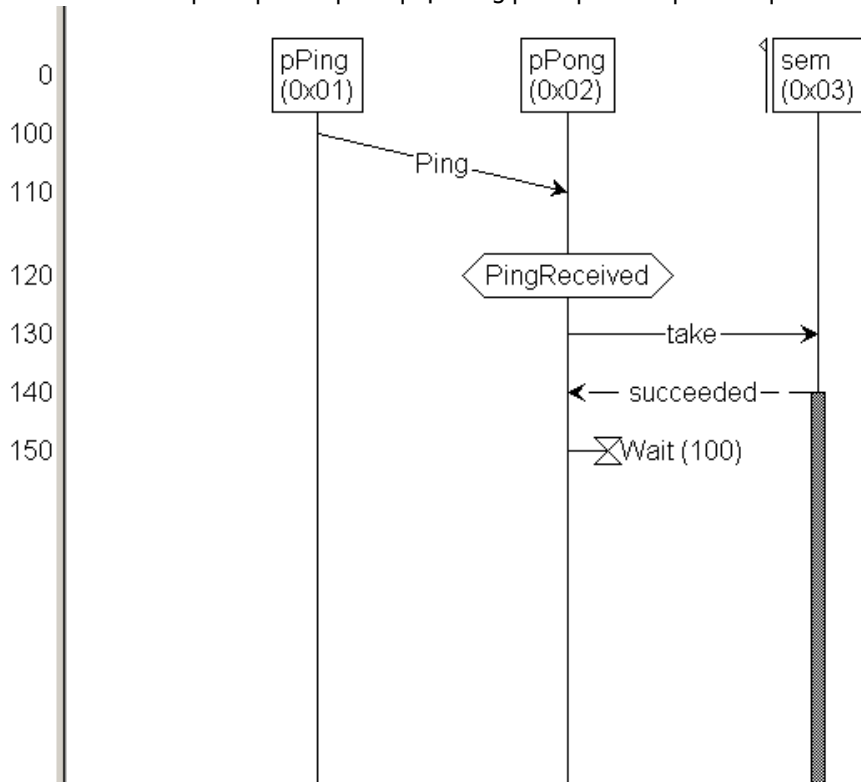
- messageReceived| -t| 110| -n| pPong| 0x02| 12| Ping|\n
- taskChangedState| -t| 120| -n| pPong| 0x02| PingReceived|\n



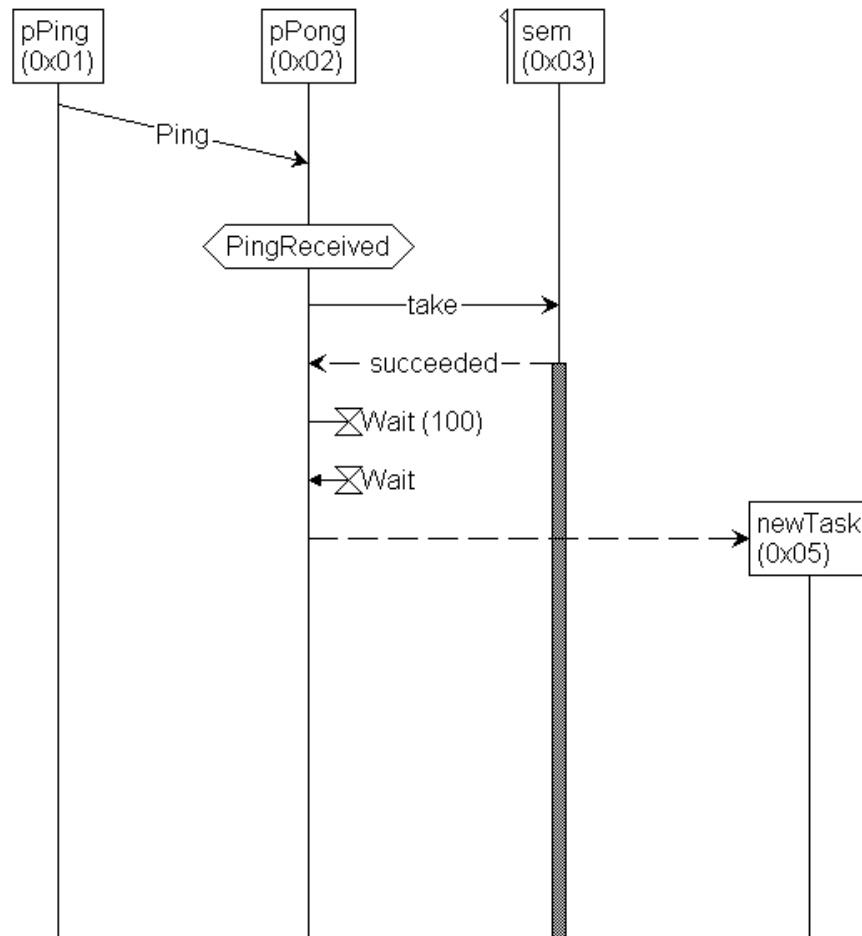
- semaphoreCreated| -s| sem| 0x03|\n
- takeAttempt| -t| 130| -n| pPong| -s| sem| 0x02| 0x03|\n



- takeSucceeded| -t| 140| -n| pPong| -s| sem| 0x02| 0x03|\n
- timerStarted| -t| 150| -n| pPong| -T| Wait| 0x02| 0x04| 100|\n



- timerTimedOut| -t| 250| -n| pPong| -T| Wait| 0x02| 0x04|\n
- taskCreated| -t| 260| -c| 0x02| -n| newTask| 0x05|\n



- giveSem| -t| 270| -n| pPong| -s| sem| 0x02| 0x03|\n
- taskDeleted| -t| 280| 0x04|\n

