1 Organizational Aspects

Project:	**D1.Project1**
Name:	**Simulator Simple 2**
Document:	**Requirements Specification**
Responsible:	**Th. Hillenbrand**
File-Pathname:	/amd/ag-avenhaus/hillen/Simulator/Requirements_Specification

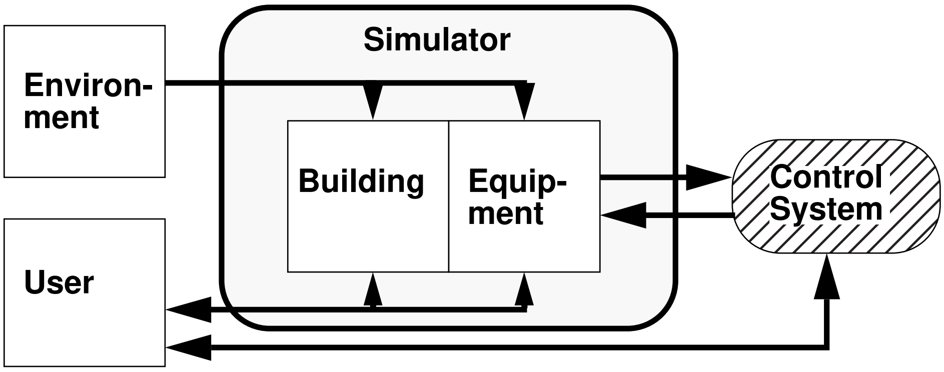
Type of change	Date	Version	Person
Start	22.11.95	1.0	Th. Hillenbrand
Minor rework completed	23.01.97	1.1	Th. Hillenbrand

2 Problem Statement

2.1 Introduction

The main purpose of the system to be developed is the following: Given an adequate model of some arbitrary building and its equipment, the system simulates some of the physical properties of parts of the building during a certain period of time and with an appropriate accuracy.

Some of the situations the system shall be useful in:
- Properties of a future building and its installation can be tested before construction; this is useful in the context of architecture.
- The owner of a building may want to test new equipment, new automatization soft- and hardware or new settings for existing ones.
- Developers of automatization software can validate their products in an environment where faults and errors are less unpleasant.



The arrows in the above illustration depict dependencies and interactions in the physical world, and the shadowed simulator box demonstrates that the system has to play the role of the building and its equipment, including all of the modelled arrow relations.

2.2 Behaviour at the User Interface

Before starting, it is possible to set:
- installation parameters, such as the standard heat output of the radiators;[1]
- initial values of physical quantities and equipment state, the so-called visible variables; an example is the air temperature of a room,
- input and output configuration,
- halting-points,
- simulation-specific parameters, such as the mode for simulation time advancing or the slack.

For everything not explicitly set, default values are assumed. It is possible to dump the initial values. The simulation can be run until a specified point in time or “forever”.

When running, it is possible to stop and resume the simulation. When stopped, it is possible to change the values of all visible variables, but not of installation parameters. Furthermore, the simulator accepts inputs during a run without being stopped.

Inputs are setting of or request for the values of visible variables.
Requests for values are responded to the requester immediately and without increase of simulation time.
Concurrent sources of input are: the user via keyboard while the simulator is running or halting; files; sensors; external programs controlling a part of the installation.

1. In contrast, the topology of the building is fixed: the size of the building, the number of rooms, etc.

Input collisions – two different attempts to set a visible variable at the same time – are detected and reported.
The format of input is a triple *(timestamp, object, message).* In case of periodic events, a fourth element *period* may be added. Furthermore, there is a special timestamp meaning “as soon as possible”.

Outputs are sent to several destinations and consist of values of visible variables. The format of output is similar to that of input. Outputs can be written periodically, on request or after change of value. This may differ for every pair of source – some visible variable – and destination.

The simulator starts with a specified internal **simulation time** and may advance in two different manners: as fast as possible or a given **timing-factor** faster than real time, the time going by when the simulator is executed. In the latter case, there is a close binding between real time and simulation time: If the factor is e.g. *24*, it should take exactly an hour to simulate the behaviour of the building during a complete day. Simulation time is allowed to be a maximum amount of time behind; this deviation will be called slack. Violation of this condition must be reported; the simulation run will stop in this case if this is chosen before starting.

2.3 Design Constraints

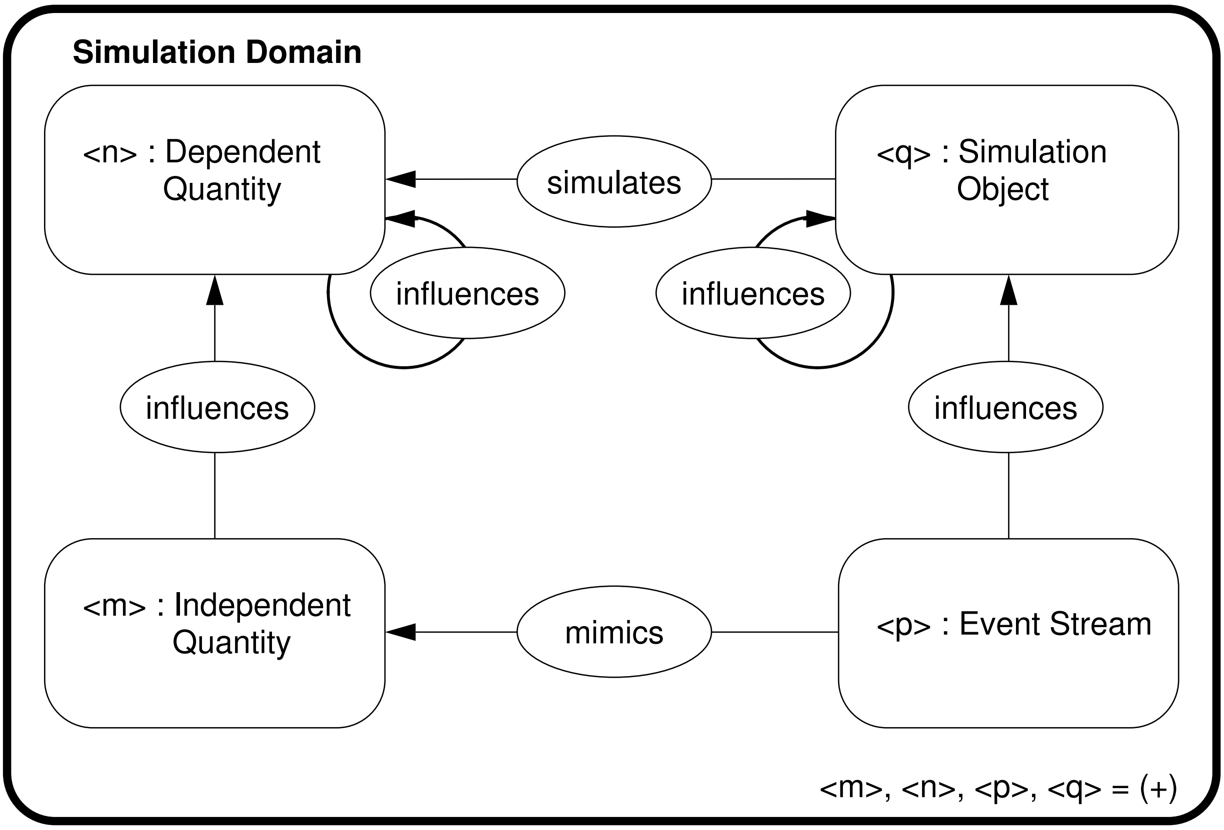
The **simulation** is to be done as an **event-driven** one with discretely advancing simulation time. The physical properties of the building and its installation shall be encapsulated in “simulation objects” which have to be invoked periodically to perform their simulation. To ensure numerical stability, there is a maximum invocation period for each object. This may change during a simulation run. The whole process of modelling shall be done in an **object-oriented** manner, thereby giving rise to re-use and forcing a generic approach. To avoid the situation that at time t an object o_1 does its simulation using part of the state information of object o_2 which has just changed at time t, every object has to buffer its previous state. This technique will be called **double-buffering.**

Errors occurring during a simulation run are reported. The program continues as long as the simulation results are not affected. Incoming data has to be checked for **incorrectness,** which must be stated. As soon as the simulation objects themselves realize that their internal state is **inconsistent**, the simulation run is aborted.
The system is able to write **dumps** containing all the visible state information of the simulation objects and **checkpoints** which serve to re-start an identical simulation run.

On a typical workstation, it shall be possible to simulate the example given in the problem description such that simulation time is cut down by a **timing-factor** of *500*.

3 Concept Model

First of all, the basic notion of simulation shall be revisited, with the design constraint in mind that the simulation is to be done in an object-oriented fashion. We thereby get the following general picture of the simulation process to take place:



In the left half of the diagram we find representations of objects in the real world. These are looked at from the physical point of view. In our context, **independent quantities** are such as the dimensioning of the rooms, the size of the windows or the standard heat output of radiators. These have an **influence** on the **dependent quantities**, for example on the indoor air temperature or the state of a radiator valve that might be opened by an inhabitant or even by an automatic control system. The outdoor climate could be subject to simulation as well. However, we are aiming at automatization of buildings and interested in observing the indoor climate. Therefore, the outdoor climate will be regarded as independent quantity, which itself is subject to change in dependence of time.

Given the dependent and independent quantities on the left, the right half shows their counterparts in the simulation process. The precise contents and behaviour of these counterparts stem from that model of the physical processes which we have shaped in our mind.

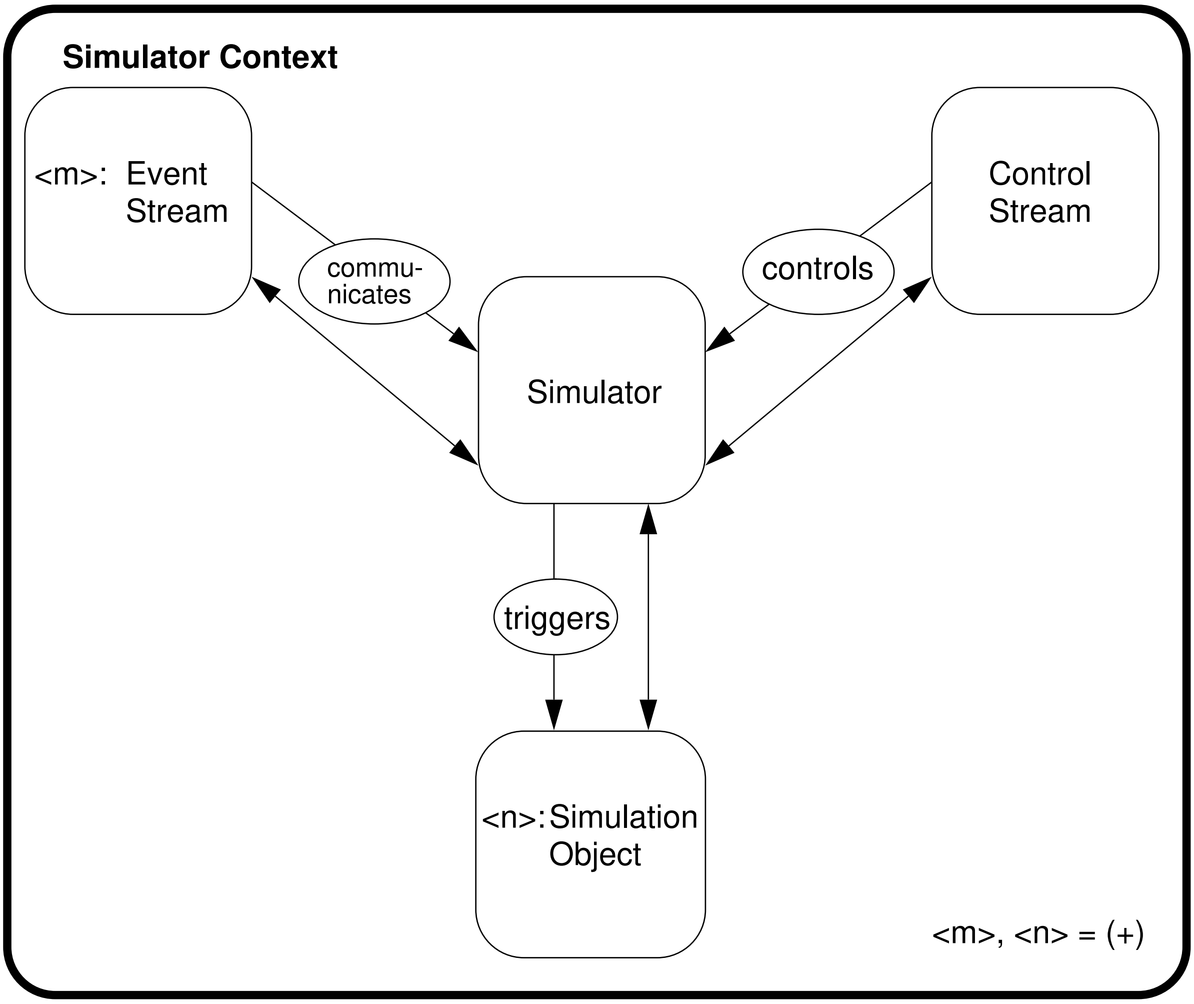
The independent quantities are **mimiced** by **event streams** with respect to the dynamic behaviour. The intention is that the relevant information is accessible at the right time in a processible format, with the notion of “relevant” left open to the focusses and accuracy of modelling.

As a design constraint, the dependent quantities are mirrored by the **simulation objects** encapsulating the former’s physical properties. Therefore, the **influence**-relation is found both on the right and on the left. Finally, we have the **simulate**-relation including the aim of the simulation domain: namely to draw conclusions about some physical circumstances in the future from the model we have built. – In our terminology, the aim of the simulation process is that this diagram commutes in some way.

4 Context Diagram

The next step will be to identify the system: We will have to decide what should be part of it.

Let us have a look at the entities first. Obviously, neither the dependent nor the independent quantities belong to the system, as they are part of the physical world. From the picture of the simulation domain on the previous page we conclude that the main task of the system is to realize the **influence**-relation between the different event streams and the various simulation objects. The **simulate**-relation, that is the actual modelling of the physical, dynamic behaviour of the dependent quantities, is encapsulated in the simulation objects (design constraint). The **mimic**-relation demands correspondence of independent quantities and event streams, which have a format the system is able to process, and is subject to modelling as well. Hence, only a few entities and relations of interest remain:

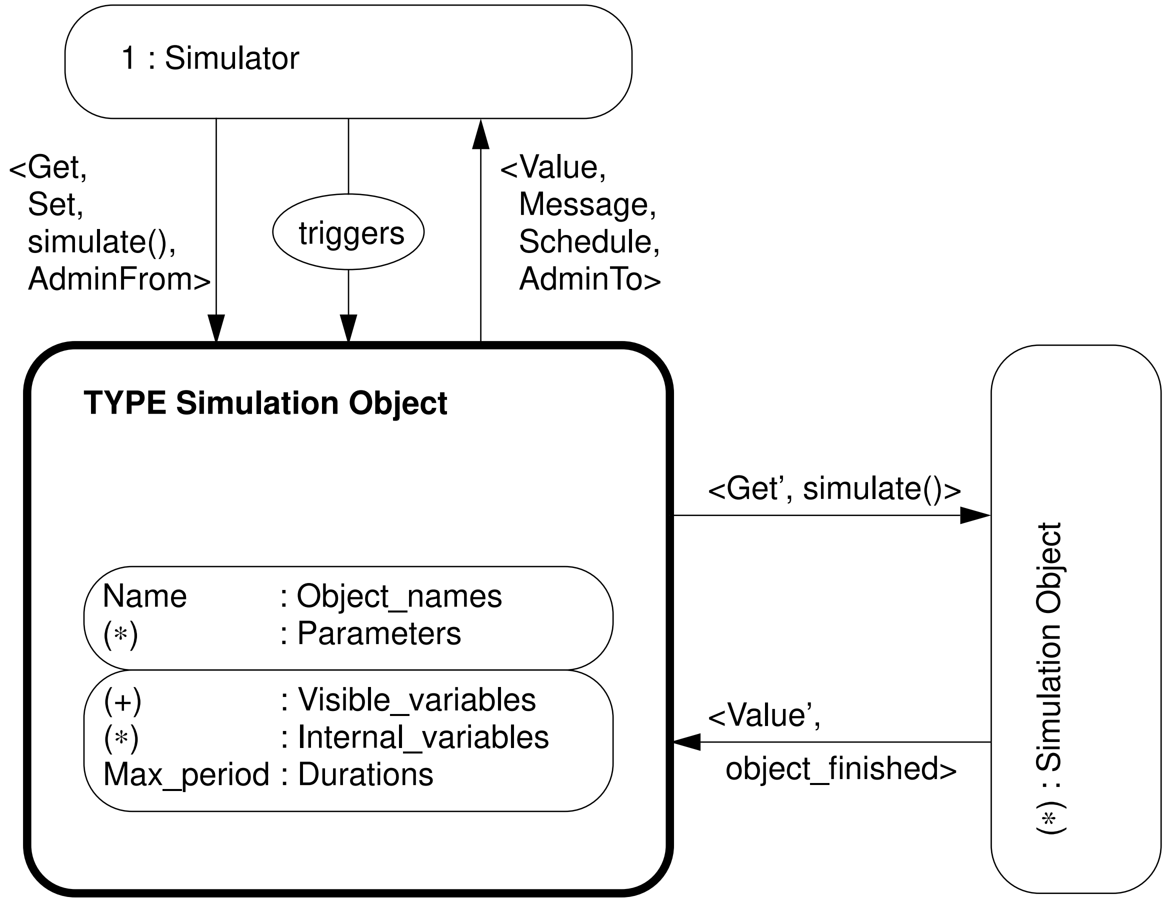


From the simulation domain, only the event streams and the simulation objects are left. The **influence**-relation is decomposed into the entity **Simulator**, which on the one hand **triggers** the **Simulation Objects**. On the other hand, the **Event Streams** are **communicating** with it. There is a special **Control Stream** which is used to supervise the behaviour of the simulator itself.

We will now go into detail and have a closer look at each of the entities in the simulator context, especially with respect to the signal ways. To give a more precise image of the data flows, signals are allowed to carry tuples of typed information. This is denoted by an empty pair of brackets following the signal name inside of the diagrams. The signal signature will be given below. Data types will be explained in an informal way as they occur; a list of them is given at the end of this chapter. It is possible to construct pairs and lists made up of simpler types. The names of types and of signal lists start with a capital letter. Data types usually have names in the plural. From now on, entities, signals and data types cited in the text will be written in the same typeface as in the diagrams. There is a data type Names, a specialization of character strings which comprises the allowed names for event streams, files, visible variables, and so on. All of these entities can be referenced externally simply by means of unique names. This allows easy design of event streams and simulation objects and also simplifies re-starting the simulator from earlier system dumps. To achieve easy referencing, every externally named entity has to send a register signal to the simulator in the initialization phase. For every domain, such as event streams, simulation objects etc., the simulator has to keep a separate dictionary realizing the mapping from names to internal references.

4.1 Simulation Objects

The entity related most closely to the simulator is that of the simulation objects.



The corresponding type is a self-referring one: Objects are allowed to communicate with each other without assistance of the simulator. Therefore, their internal references must be known to each other. As the actual calculations take place inside the simulation objects, sufficient effort has to be spent on minimizing the communication effort. We therefore introduce the paradigm that on the lower level referencing shall be done directly with internal references. When designing simulation objects or retrieving system dumps, however, using their external references – their unique names – is appropriate.

Inside every simulation object, there may be the following attributes:
- their unique name,
- a list of individual parameters which cannot be changed during simulation runs,

- a non-empty list of visible variables containing both dependent and independent quantities,
- a list of internal variables representing the internal state variables of the object,
- the maximum amount of time between two subsequent invocations of the object which guarantees appropriate accuracy.

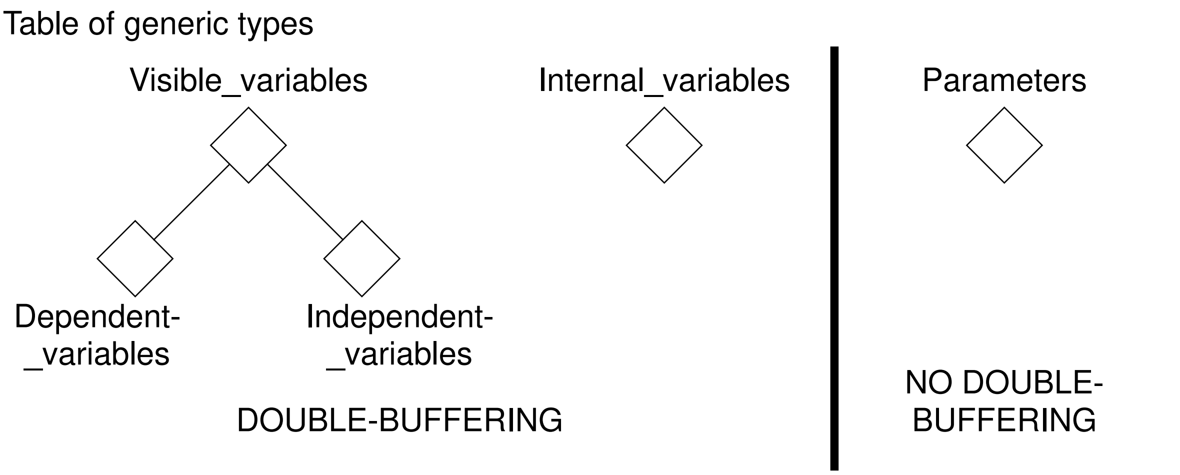
Following this paradigm, every instance of Parameters, Visible- and Internal_variables has its own unique name, including the name of the enclosing object as prefix, e.g. *kitchen/radiator/valve-state* if the object is the radiator in the kitchen and the variable is the state of its valve.

The data types in the lower attribute box have to meet yet another requirement: They are subject to double-buffering. An advisable general solution is to refine the corresponding get and set operations. They are extended with an additional parameter, namely the current simulation time. The latter is an element of Timestamps, the set of all possible simulation times. The double-buffered variable now keeps an elder and a younger value together with a limit time, the last simulation time at which the elder one is valid. We may assume that the query timestamps are monotonously increasing.
For get access, if the query timestamp is less than or equal to the limit timestamp, the elder value is returned. As soon as the query time exceeds that timestamp, the younger value is delivered.
For set access, writing with an earlier query timestamp is left undefined. If the query is equal to the limit time, a write collision occurs: two attempts at the same time to alter the same variable. A warning should be issued, and the second attempt be ignored.[1] Finally, if the query is later, the formerly younger value becomes the elder one, the query time is the new limit time, and the argument of set gives the younger value. – Designing simulation objects, one must take care of this way the semantics of several set attempts at the same simulation time is fixed.
We introduce a new, special timestamp $-\infty$ which is previous to any other and visible only for the simulator and the simulation objects. Any double-buffered variable will have a default for its younger value, and the initial time border is $-\infty$. Consequently, only one default value is necessary despite of double-buffering. To simplify retrieval of dumps, we demand the exception that a set operation with the special timestamp $-\infty$ alters the younger value.
When writing a dump at some simulation time serving to restart an identical run, one cannot use the variable's values currently read by get, because current set accesses, which are visible exactly from the very next point in discrete simulation time, would not be considered. Hence, if a variable is polled for dump purposes, its value at the following simulation time must be delivered.[2]

Let us recall the model of the simulation domain. Within this framework, both dependent and independent quantities have to be mirrored inside simulation objects as visible variables. To ensure that the simulation objects are used reasonably, another distinction has to be introduced. As an example, the temperature of a wall surface is the result of frequent calculations with strong constraints concerning numerical stability. There is hardly any reason for directly assigning another value during a simulation run, as this would be a contradiction to the aim of simulating. For independent quantities such as the state of a radiator valve or the outdoor climate however, direct write access does make sense – as inhabitants might alter the valve as well. This is why we will distinguish two subclasses, henceforth referred to as Dependent_variables respectively Independent_variables. Every instance of Visible_variables belongs to exactly one of them.

The data type Visible_variables has to be refined further: The problem statement requires that specific output – a value of a certain simulated dependent quantity – is sent to some event

1. See below for the uncritical case of simulate collisions.
2. See Appendix A for a formalization of double-buffering.



stream on explicit request, periodically or on change. The first and the second one will be initiated by the simulator. But as simulation objects are allowed to activate other objects as well, a value of a visible variable may change without notice of the simulator. Therefore the type Visible_variables must include also a – possibly empty – list of event stream references, the variable's output configuration. The simulator has to inform the corresponding event streams as soon as the value of the variable alters.

The **signals between the simulator and the simulation objects** are arranged in several groups. The Get group provides read access to the simulation relevant of the object's attributes. The signals dump_values and dump_anything stimulate the object to return all the values of parameters and visible variables and in the latter case even of internal variables. As mentioned above, dump_anything asks for the values not at the current, but at the very next simulation time.

The Set group realizes the corresponding write access. For visible variables, there is a common signal set_visible_variable referring both to dependent and independent variables, which should be used in the simulator initialization phase only. The signal set_independent_variable is to be used at run-time. (Consequently, direct write access to dependent quantities at run-time is disabled.) – The output configuration of visible variables is modified by adding or removing references to event streams or just by emptying it.

The Value group comprises signals used to return the requested values. As signals are allowed to overtake each other, the Value signals also carry an identifier indicating the origin of the information. The signal value_changed notifies that a visible variable has changed at the current simulation time and that certain event streams are interested in being informed.

<Get> = [get_visible_variable(Visible_variable_IDs, Timestamps),
dump_values(Timestamps),
dump_anything(Timestamps)]

<Set> = [set_parameter(Parameter_IDs, Values),
set_visible_variable(Visible_variable_IDs, Values, Timestamps $\cup \{-\infty\}$),
set_independent_variable(Independent_variable_IDs,
Values, Timestamps),
add_output_configuration(Visible_variable_IDs, Event_stream_IDs,
Timestamps $\cup \{-\infty\}$),
empty_output_configuration(Visible_variable_IDs, Timestamps $\cup \{-\infty\}$),
remove_output_configuration(Visible_variable_IDs, Event_stream_IDs,
Timestamps $\cup \{-\infty\}$),

set_internal_variable(Internal_variable_IDs, Values, Timestamps $\cup \{-\infty\}$),
set_max_period(Durations, Timestamps $\cup \{-\infty\}$)]

<Value> = [parameter(Parameter_IDs, Values),
visible_variable(Visible_variable_IDs, Values),
output_configuration(Visible_variable_IDs, LIST OF (Event_stream_IDs)),
internal_variable(Internal_variable_IDs, Values),
max_period(Object_IDs, Durations),
value_changed(Visible_variable_IDs, Values,
LIST OF (Event_stream_IDs))]

The signal simulate triggers the object to perform a single simulation step. According to the course of its calculations, the object returns an answer from the Message group. A signal object_finished is sent exactly in the successful case. In contrast to warnings, errors disable the continuation of the simulation run.

simulate(Timestamps)

<Message> = [object_finished,
object_warning(Strings),
object_error(Strings)]

Objects shall be allowed to schedule and unschedule events. Scheduling shall be restricted to such events that trigger a simulation step. This is realized by the signal schedule_event. It also carries a parameter of type Durations, the period. In case of non-periodic events, this must be set to zero. The schedule signals are answered by the simulator via the scheduled_event signal from the AdminFrom group (see below) carrying the event identification. With this identification, the event may be removed via the unschedule signal, or its period may be changed with the signal reschedule_event. If the event could not be accepted, e.g. because of a reference to a non-existing object, a signal schedule_error is delivered.

<Schedule> = [schedule_event(Timestamps, Object_IDs, Durations),
reschedule_event(Event_IDs, Durations),
unschedule_event(Event_IDs)]

Finally, there is a number of signals for administration purposes. At start-up time, the simulator sends an initialize signal. The object answers with register_object and thereby reveals its name, its maximum invocation period and all the parameters, visible and internal variables with internal and external references. The simulator now establishes the dictionaries for the mapping from external to internal references. Furthermore, for every object an initial periodic simulate event is scheduled; and its identification is sent to the object via the signal scheduled_event.
Any object may have been designed with references to other simulation objects or their visible interior. These references are necessarily external ones. During the initialization phase, the object is allowed to ask for corresponding internal references by the help of the get_ID signal. The result is returned via the id signal, which also carries the query name to avoid conflicts in case of signals overtaking each other. The data type IDs is the most specific generalization of Simulation_Object_IDs, Parameter_IDs, etc. The Message signal group can be used here as well. The object sends object_finished if and only if its initialization terminated successfully. After all the objects and streams are initialized, but still before starting, the user is enabled to modify the system state, e.g. retrieve a dump or change a few installation parameters. Thereby some of the objects might reach an inconsistent state. When these changes are done, the simulation objects shall receive a signal get_ready to re-establish consistency. Afterwards, the simulation may begin.

<AdminToSO> = [initialize,
get_ready,
scheduled_event(Timestamps, Object_IDs, Durations, Event_IDs),
schedule_error,
id(Names, IDs)]

<AdminFromSO> = [register_object(Object_names,
LIST OF (Parameter_names, Parameter_IDs),
LIST OF (Visible_variable_names, -_IDs),
LIST OF (Internal_variable_names, -_IDs),
Durations),
get_id(Names)]

The **signals between the simulation objects** are a subset of those already mentioned. Their communication is restricted to the Get' and Value' group for read access and the activation call simulate with exactly one corresponding signal object_finished in case of successful termination. Direct write access via the Set group is not allowed.
For simplicity reasons, we assume at this early stage of modelling that all the objects are able to communicate with each other directly.

<Get'> = [get_parameter(Parameter_IDs),
get_visible_variable(Visible_variable_IDs, Timestamps)]

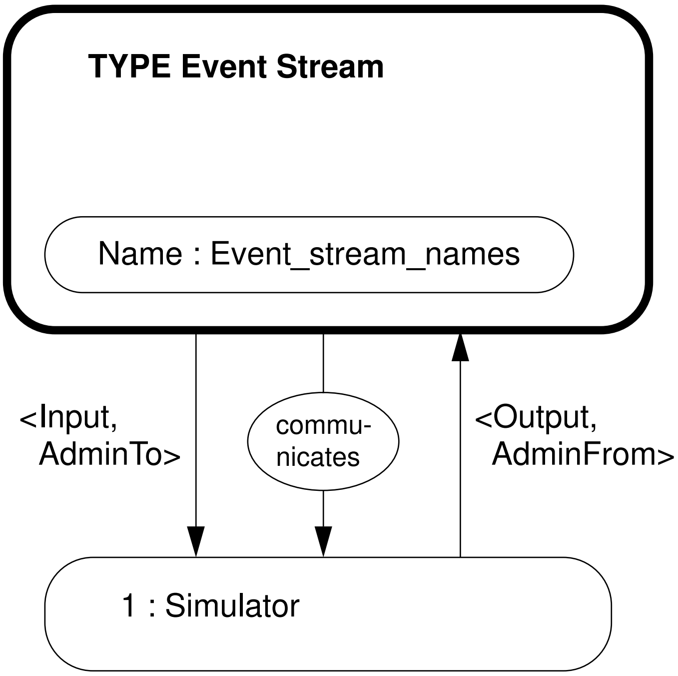
<Value'> = [parameter(Parameter_IDs, Values),
visible_variable(Visible_variable_IDs, Values)]

The problem statement includes the requirement that input collisions shall be detected and reported. There cannot be any collision between a set signal and a simulate signal, since the first refers to an independent variable and the second to an object's dependent variables, which corresponds to the distinction between independent and dependent quantities in the concept model. Any collision of set signals, which must be due to input from the event streams, is detected and reported with the help of the variable's internal limit time and resolved non-deterministically. Because of the double-buffering technique, a collision of two simulate signals is not critical: They lead to the same calculation results so that the second one can just be ignored.

Furthermore, it should be possible to write dumps serving to restart an identical run. To achieve this, the design of simulation objects must meet two constraints: First, when required, the objects must dump all the state information they need for reproducing that very state. Second, as there are no priorities for events with equal execution time, the simulator must be free to invoke the corresponding objects in an arbitrary order. This is easily achieved if the double-buffering technique is applied and if dependent and independent variables are not mixed up.

4.2 Event Streams

Every event stream is connected with nothing but the simulator:



<Inputs> = [io(Timestamps $\cup$ $\{*\}$, Object_names, Messages),
io_period(Timestamps $\cup$ $\{*\}$, Object_names, Messages, Durations)]

<AdminFromES> = [register_event_stream(Event_stream_names),
get_simulation_time,
event_stream_warning(Strings),
event_stream_error(Strings)]

<Outputs> = [io(Timestamps, Object_names, Messages)]

<AdminToES> = [initialize,
simulation_time(Timestamps),
io_error(Timestamps, Object_names, Messages, Durations,
Strings),
finished]

The concept model states that event streams have to mimic independent quantities with respect to their dynamic behaviour. In our context, typical examples are the outdoor climate or the state of a radiator valve. Advancing to the context diagram, we have embedded the independent quantities as variables into the simulation objects being accessible from the event streams, which realize the dynamic behaviour by explicit get and set operations. Hence they have to communicate with the simulator, and the problem statement requires as the format of input and output a triple made up of the sets Timestamps (all possible simulation times) $\cup$ $\{*\}$, Object_Names and Messages.

The symbol $*$ denotes a special timestamp meaning “as soon as possible”, but referring to a later simulation time than the current one. It is not visible for the simulation objects. A message is a pair of an element of Visible_variable_names, which indicates the variable the command refers to, and of Values, which is the new value for the variable in case of input and the special symbol $\perp$ in case of an output request. As the name of the enclosing object is given as first argument of the io signal, the variable name ought be given without that prefix. The signal io_period is used for periodic inputs and output requests.

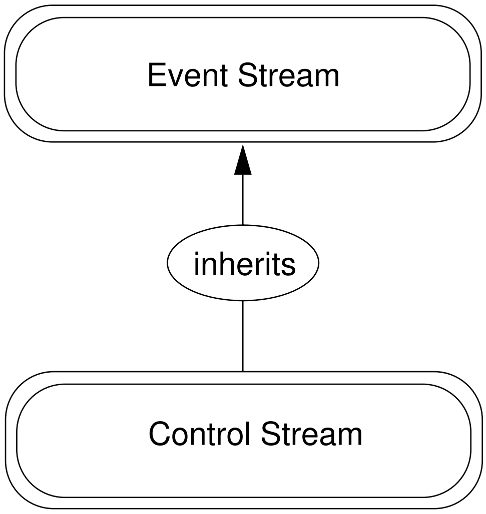
From the simulator's point of view, every event stream in the input direction consists of a sequence of time-stamped elements arriving at a specific (simulation) time. The simulator now has to compare timestamp and simulation time; and according to the result, some of the inputs will be accepted and considered at the desired moment, whereas others will be discarded. Equally, the output direction is a sequence of time-stamped elements which are sent at a specific simulation time. The simulator should ensure that these have the current simulation time as timestamp.
For concrete system realizations, precise mechanisms to synchronize the simulator and the event streams are necessary. At this early stage, this is yet left open.

A number of signals for administration purposes remains to be explained. Every event stream is granted its own initialization phase, which it enters as soon as the simulator sends an initialize signal. According to the paradigm of internal and external referencing, every stream has a unique name, which it is expected to send to the simulator with the register_event_stream signal. Afterwards, its initialization must be finished, and the simulation may begin.

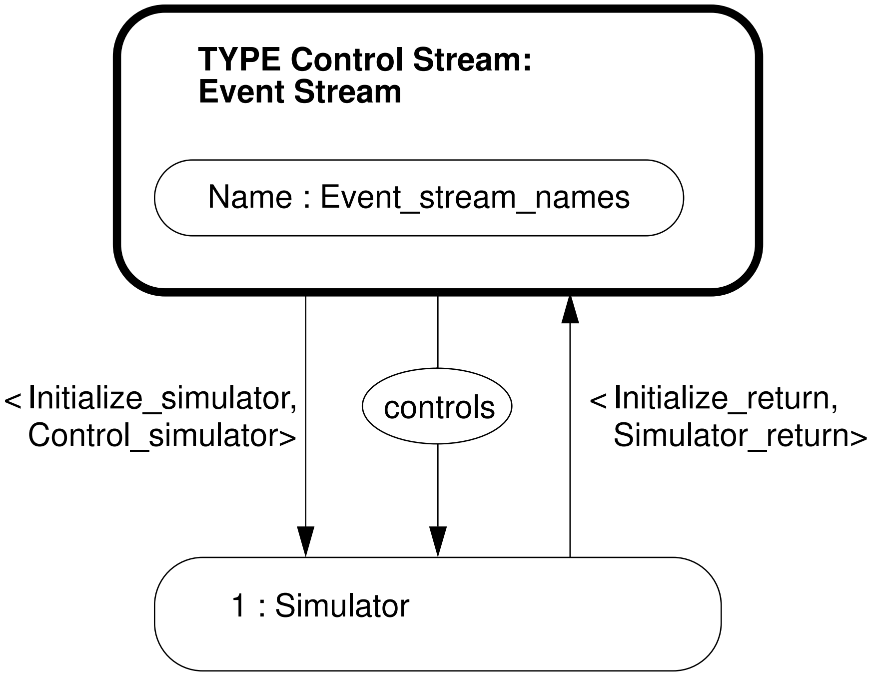
Via the get_simulation_time signal, it is possible to poll the current simulation time, which is delivered by a signal carrying just that name. If there are any irregularities in an event stream, it will signal that state to the simulator using event_stream_warning or even event_stream_error in case that continuing the simulation run makes no sense. Vice versa, io_error shows that a special io request could not be delivered. Finally, the simulator sends a signal finished when the complete simulation is done.

4.3 Control Stream

Among all the event streams, there is a special one which is allowed to alter the behaviour of the simulator. The relation among the corresponding types is the one of inheritance.



The following signals are added:



The control stream starts the initialization phase of the simulator via a start_initializing signal. Only in this phase it is possible to alter the values of parameters of simulation objects, to specify the advancing mode of the simulation time, to set a starting-point in simulation time, and to retrieve an earlier system dump. Naturally, later calls are expected to overwrite earlier ones. The parameter of the signal retrieve_dump refers to a file system which is accessible for the control stream. The set_time_mode signals trigger the different advancing modes of the simulation time: first, as fast as possible, or second, a fixed timing-factor faster than real time with an allowed maximum deviation and a flag whether to continue or not in case this slack is exceeded. The initialization phase is ended by sending the signal get_ready; afterwards, the simulation run starts.

<Initialize_simulator> = [start_initializing,
alter_parameter(Object_names, Parameter_names, Values),
set_time_mode_fast,
set_time_mode(R^+, Durations, Boolean),
set_simulation_start(Timestamps),
retrieve_dump(File_names),
get_ready]

The signals from the Control_simulator list may be sent at initialization time or later on. They allow to alter the value of an independent variable, to specify the simulation end in simulation time, to set and unset halting-points, to stop and resume the simulator, and finally to write dumps of the system state. These may be restricted to the values of parameters and visible variables or comprise the whole system's state information so that re-starting an identical run is possible. (More precisely, having received a dump signal, the simulator finishes what has to be done at the current simulation time and then makes the latter advance one resolution tick. Before any other operation, the objects are polled for the values of their variables, and afterwards, the simulation continues. From that very moment in simulation time, the retrieved dump will yield an identical run.) The parameters of the signals dump_values and dump_anything again refer to the file system of the control stream. Sending unset_halting_point($*$) removes the nearest halting-point.

<Control_simulator> = [alter_independent_variable(Independent_variable_names),
set_simulation_end(Timestamps $\cup \{*\}$),
unset_simulation_end,
set_halting_point(Timestamps $\cup \{*\}$),
unset_halting_point(Timestamps $\cup \{*\}$),
stop_simulation,
resume_simulation,
dump_values(File_names),
dump_anything(File_names)]

The simulator acknowledges the successful execution of the initialization phase as well as of the whole simulation process. Errors – they lead to abortion of the simulation run – and warnings are sent along with corresponding textual messages. The different signals in the Initialize_return group belong to different stages of the start-up process: actual initialization, modification by the user, and finally preparing to start the simulation run. If the simulator runs with a fixed timing-factor between real and simulation time, the signal simulation_slacked will indicate that the current deviation exceeds its requested bound.

<Initialize_return> = [initialize_OK,
initialize_error(Strings),
modify_warning(Strings),
get_ready_OK,
get_ready_error(Strings)]

<Simulator_return> = [simulation_stopped,
simulation_OK,
simulation_slacked,
simulation_warning(Strings),
simulation_error(Strings)]

4.4 List of Data Types

Boolean	- the set of Boolean truth values
Dependent_variables	- subtype of Visible_variables representing dependent quantities. When simulating, variables of this type cannot be explicitly set, as their values are the results of their objects' calculations.
Dependent_variable_IDs	- the corresponding internal references
Dependent_variable_names	- the corresponding external references
Durations	- the set of all the differences between two arbitrary timestamps
Event_IDs	- the set of internal event references
Event_stream_IDs	- the set of internal event stream references
Event_stream_names	- the corresponding external references
File_names	- the set of allowed names for files; depends on the file system that is accessible for the control stream
IDs	- supertype comprising all internal reference types
Independent_variables	- subtype of Visible_variables representing independent quantities, e.g. the state of a radiator valve.
Independent_variable_IDs	- the corresponding internal references
Independent_variable_names	- the corresponding external references
Internal_variables	- generic data type for internal state information of simulation objects which is necessary for re-starting; realizes double-buffering
Internal_variable_IDs	- the corresponding internal references
Internal_variable_names	- the corresponding external references
Messages	- the cross product of Visible_variable_names and Values
Names	- supertype comprising all external reference types; subtype of Strings
Object_IDs	- the set of internal object references
Object_names	- the corresponding external references
Parameters	- generic type representing information which is unchangeable during a simulation run, e.g. the standard heat output of the radiators
Parameter_IDs	- the corresponding internal references
Parameter_names	- the corresponding external references
R^+	- the set of positive real numbers
Strings	- the set of character strings
Timestamps $\cup \{*\} \cup \{-\infty\}$	- the set of all simulation times; the symbol $*$ denotes a special timestamp meaning "as soon as possible", the symbol $-\infty$ another special one, which is previous to any other
Values	- supertype comprising all the argument types for the generic types Parameters and Visible_variables; additionally includes

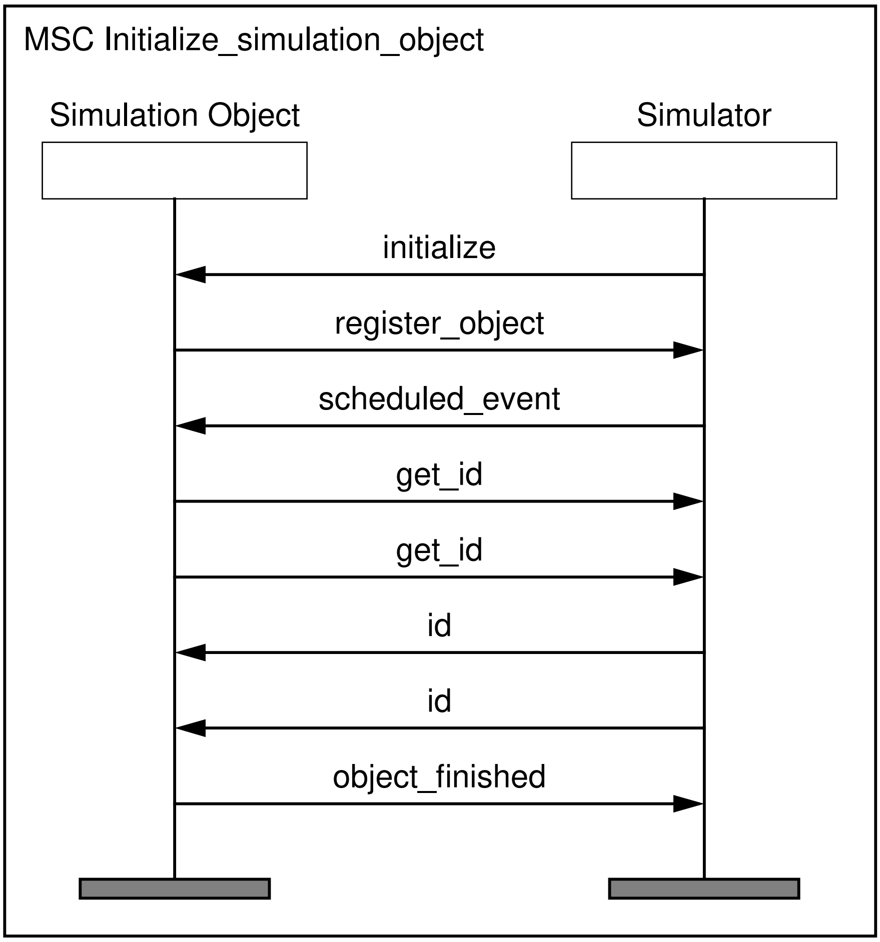
		a bottom element ⊥ which is not element of any subtype
Visible_variables	-	generic data type for that kind of state information of simulation objects which during the simulation process is visible to the event streams; realizes double-buffering; made up of the two subclasses Dependent_variables and Independent_variables
Visible_variable_IDs	-	the corresponding internal references
Visible_variable_names	-	the corresponding external references

5 Message Sequence Charts

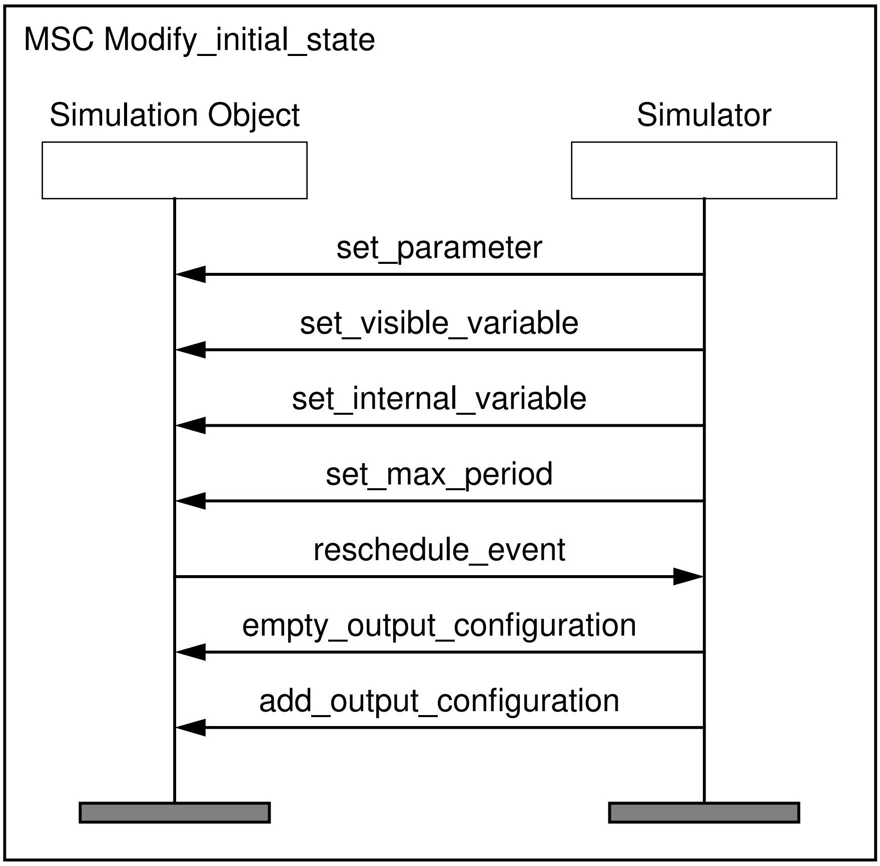
There will be at least five stages in a typical system run:
- First of all, the control stream asks the simulator to initialize.
- Second, the simulator does the same with event streams and simulation objects.
- Third, the user is allowed to modify the initial system state, e.g. change the values of some installation parameters.
- After all the modifications are executed, some of the simulation objects may have got an inconsistent state. Therefore they must be stimulated to re-establish consistency.
- Now, the actual simulation is done, which itself may be interrupted and continued.

5.1 Simulation Objects

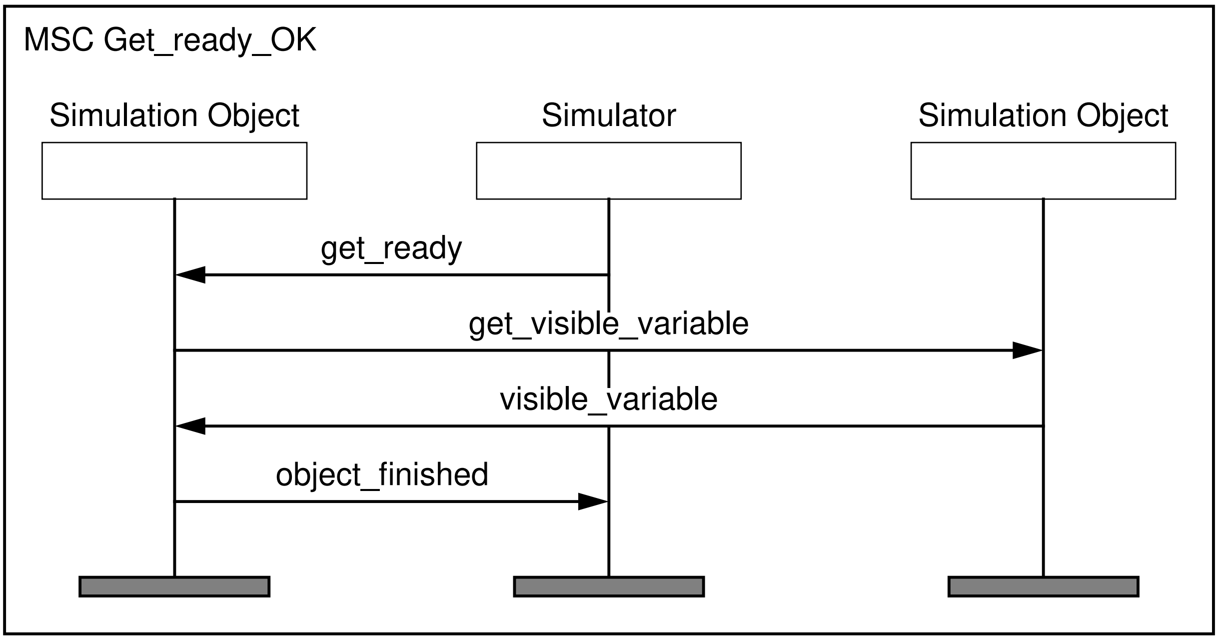
Initialization. The first chart demonstrates the initialization of simulation objects which takes place at the system's start-up time. First of all, the simulator sends an initialize signal and thereby requests the objects to reveal their names and all their parameters, visible variables, internal variables, and their maximum invocation period. With this period, the simulator schedules a periodic simulate event and sends its identification to the object via scheduled_event. Now, the object asks the simulator for the internal references of some other objects, variables or streams, sending the external references with the get_id signal. The simulator answers with the id signal. The object signals object_finished when all its requests are answered and its initialization is done or object_error in the erroneous case.



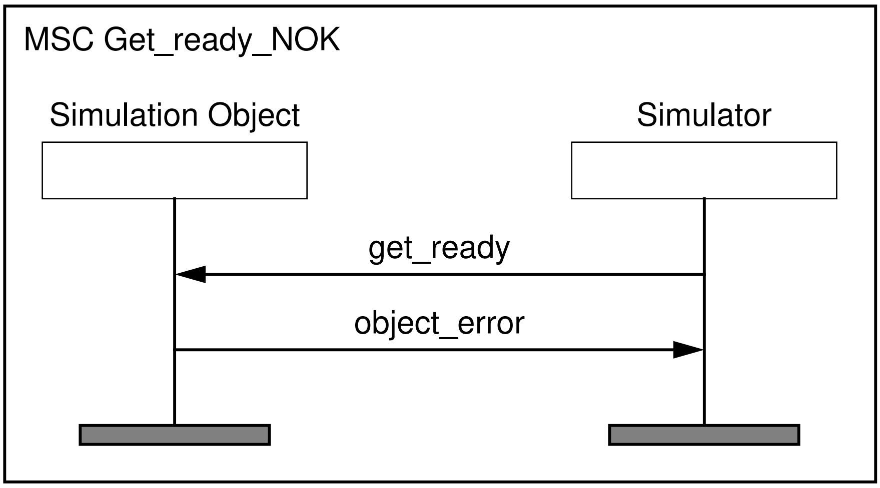
Modification. Before the simulation run is started, the user is allowed to retrieve an earlier system dump and to change some of the installation parameters. The simulator transforms dumps into a sequence of signals set_parameter, set_visible_variable, empty_output_configuration, add_output_configuration, set_internal_variable, and finally set_max_period. The latter demands re-scheduling of the initial periodic event. Those signals which must have a timestamp carry the special one $-\infty$ to make them alter the younger value of the double-buffered variable. Therefore the transferred values are valid at an arbitrary starting-point in simulation time, which is always later than $-\infty$.



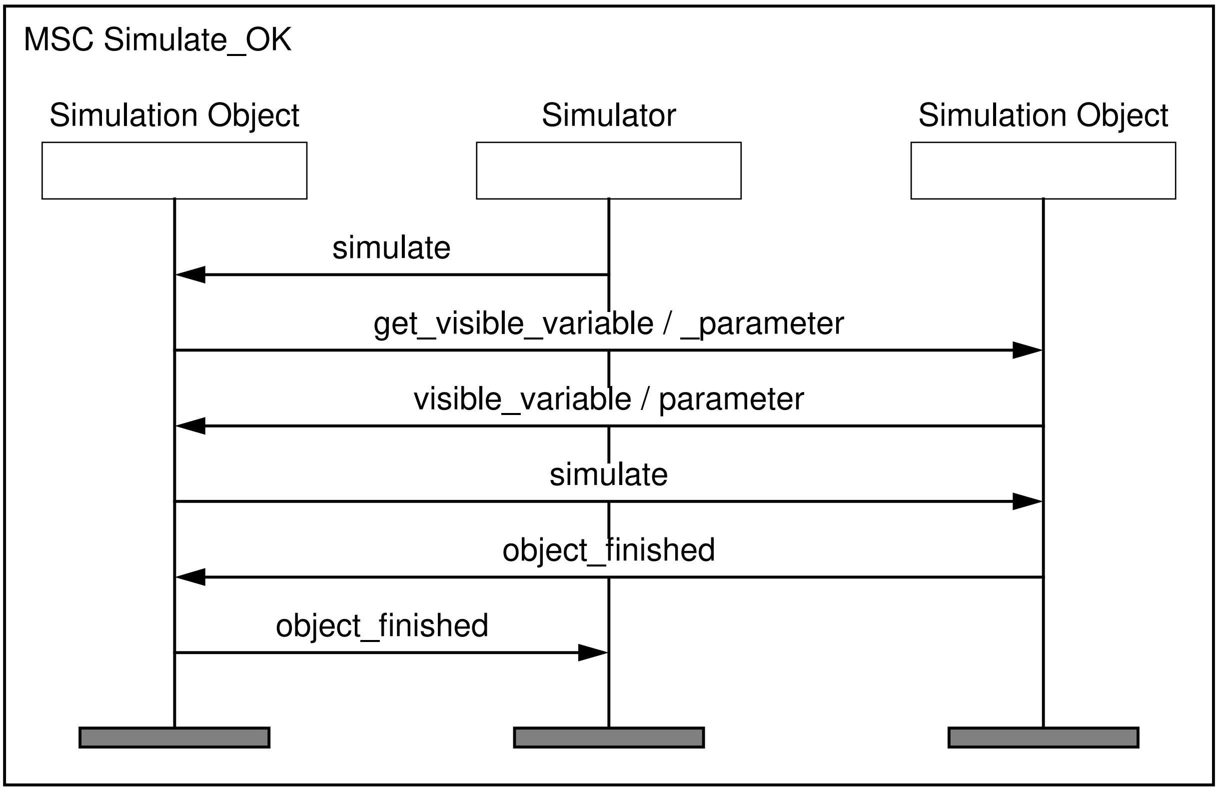
Getting ready. When the user modifies the initial system state, some objects might get an inconsistent state themselves. For that reason, consistency has to be re-established. This could be embedded into the single set operations. However, this technique is not able to cope with multiple alterations that yield consistency only as whole, such as a retrieved system dump. Hence it is advisable to activate the objects for the purpose of getting consistent again after all changes are executed. The simulator does so when sending the signal get_ready. In our first chart, the activated object needs the current value of a visible variable of some neighbouring object, does some internal calculations and gets consistent again, which is indicated via object-_finished.



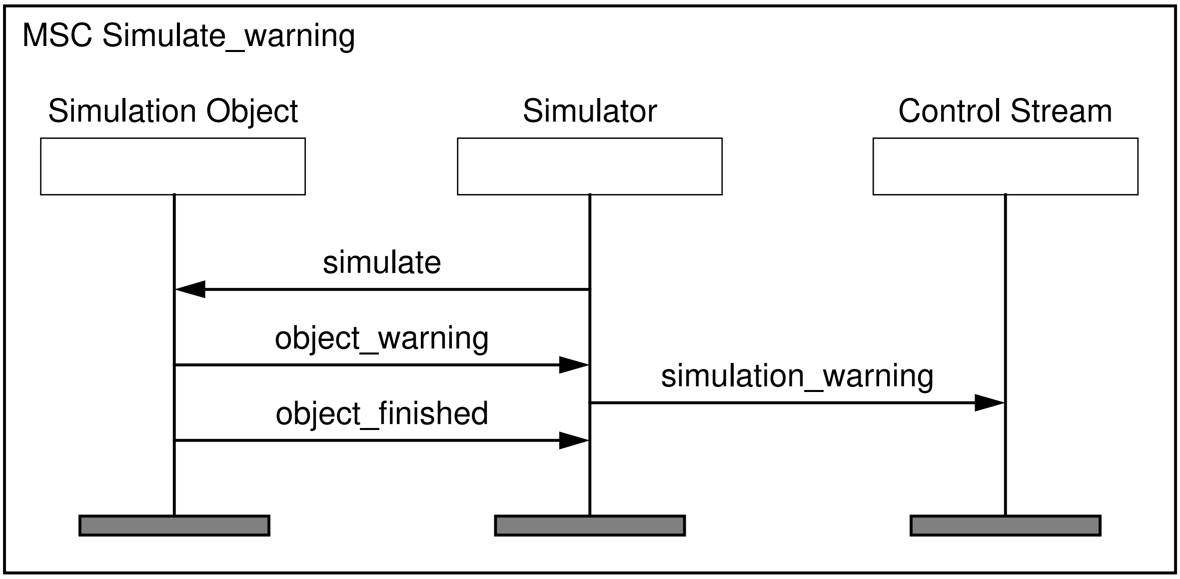
The second chart shows the erroneous case: The inconsistency inside the simulation object is too severe to be corrected. Consequently, the simulation cannot be started.



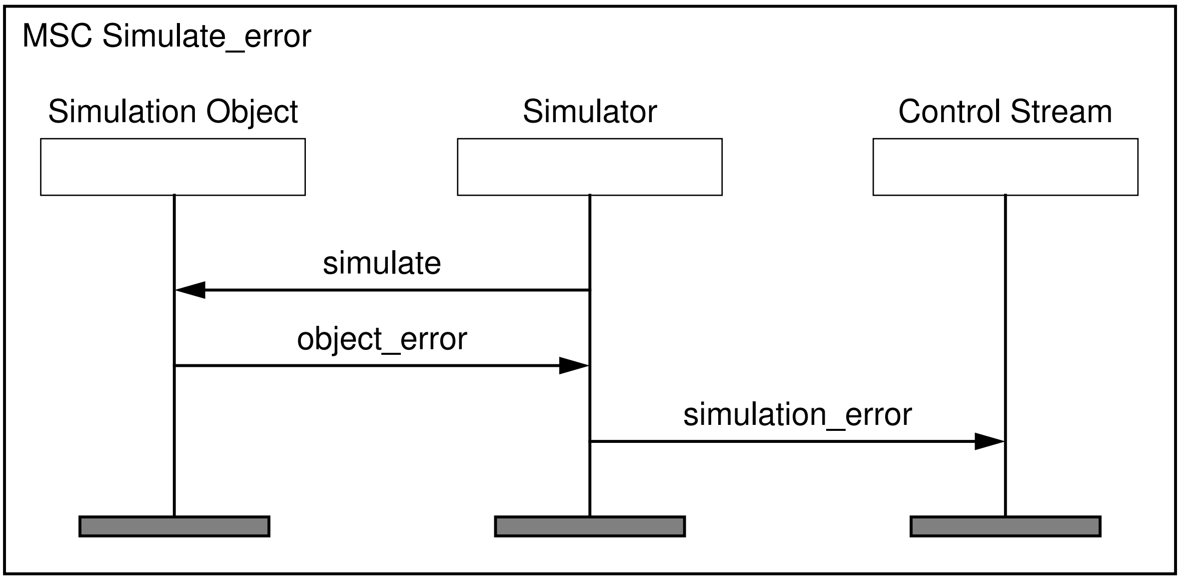
Simulate activation. The next chart shows a typical activation step of a simulation object via the simulate signal. This is initiated by the simulator or another object and carries the current simulation time. The object now has to calculate its next state which becomes the current one as soon as the simulation time advances. For the calculation, values of visible variables or parameters of other objects are needed and collected using the corresponding signals. Then the calculations are done, and the next state is present. Afterwards the object invokes another, neighbouring simulation object, which may as well poll or activate others, what is not shown here. Each object sends – having finished – a signal to that entity that has activated it; and the simulation step ends when the simulator receives a signal object_finished from that very object that has been invoked first.



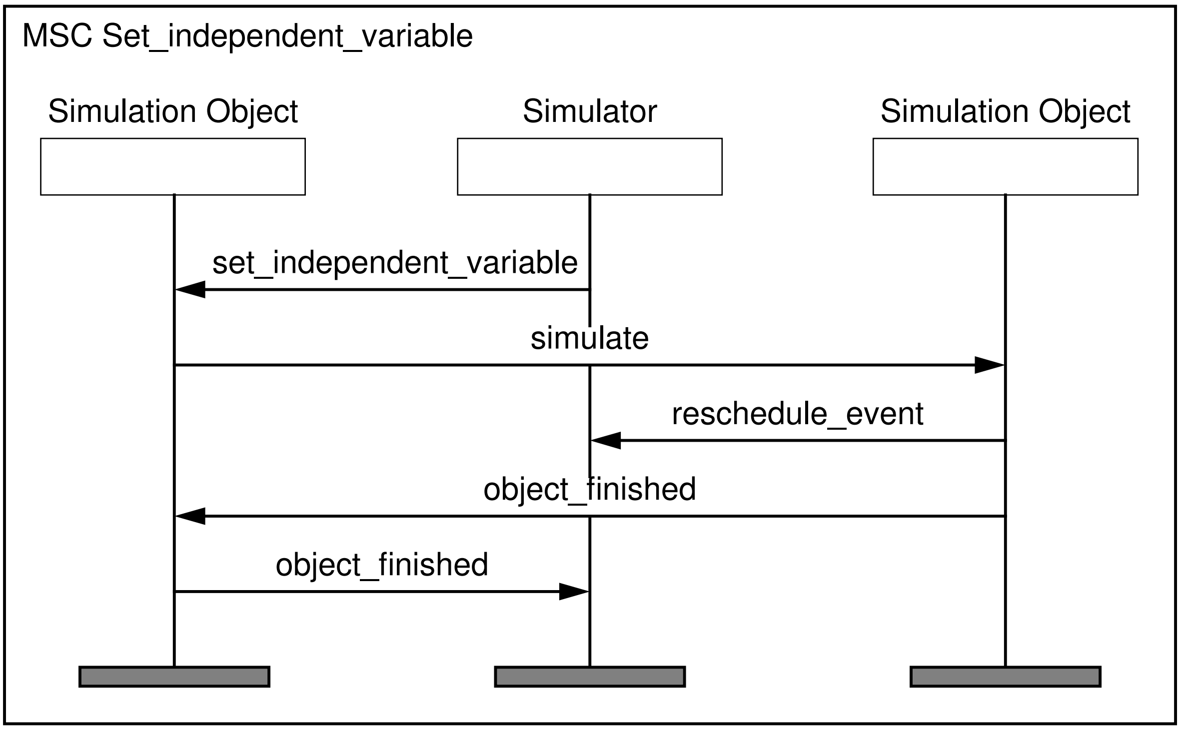
The next chart depicts a situation where a warning is issued. Activated by the simulate signal, the object calculates its next state, and e.g. a visible variable will be assigned an unusual value. As it still makes sense to continue the simulation, the object just sends a warning to the simulator. This is followed by the usual signal object_finished. The simulator furthers the textual message to the control stream.



If, however, an irreparable inconsistency arised after the activation, the simulation run would have to be aborted. To this end, the object would just send a signal object_error to the simulator.



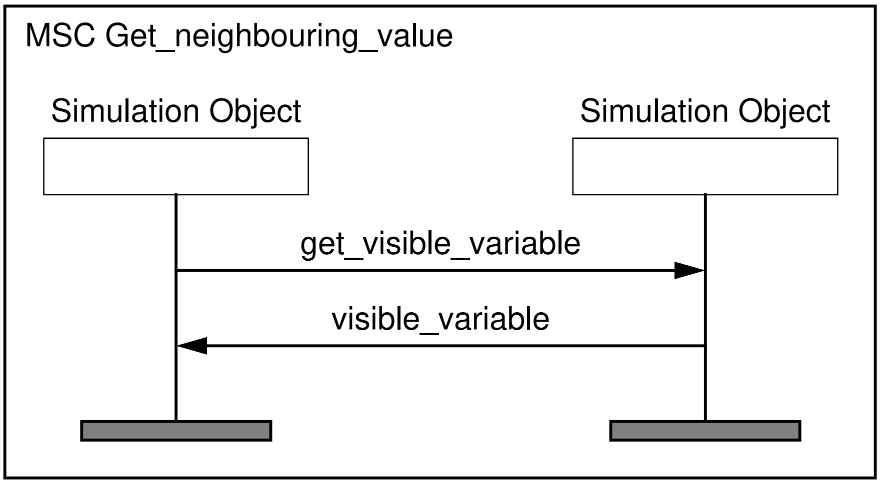
Set activation. There is another way to activate simulation objects, namely setting independent variables. This realizes set events on the lower level: The object is sent a signal set_independent_variable and becomes active. In our case, the object might be a window, which is instructed to become open. This will lead to a heat flow from the corresponding room to the outside; and the room temperature will fall significantly. Consequently, the object encapsulating that temperature is invoked, which re-schedules its periodic simulate event to ensure an appropriate accuracy.[1] In the end, the simulator receives the final signal object_finished. Just as in the previous activation case, the objects involved could also have issued a warning or even an error.



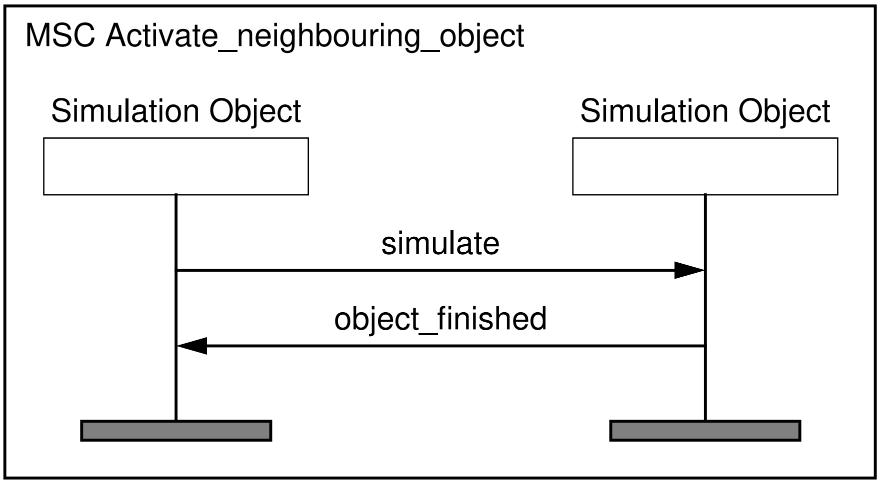
1. Note that re-scheduling can only be done by the object which has scheduled the event, for the simulation objects cannot exchange event identifications.

Actions on activation. Being activated by one of the signals simulate and set_independent_variable, there are several communicative actions the object may undertake. Some of them were already mentioned. We will describe all the actions in short. Remember that at the end of an error-free active phase a signal object_finished is issued. Warnings may be sent to the simulator as soon as something suspicious occurs, and errors are indicated via the signal object_error and lead to abortion of the simulation run.

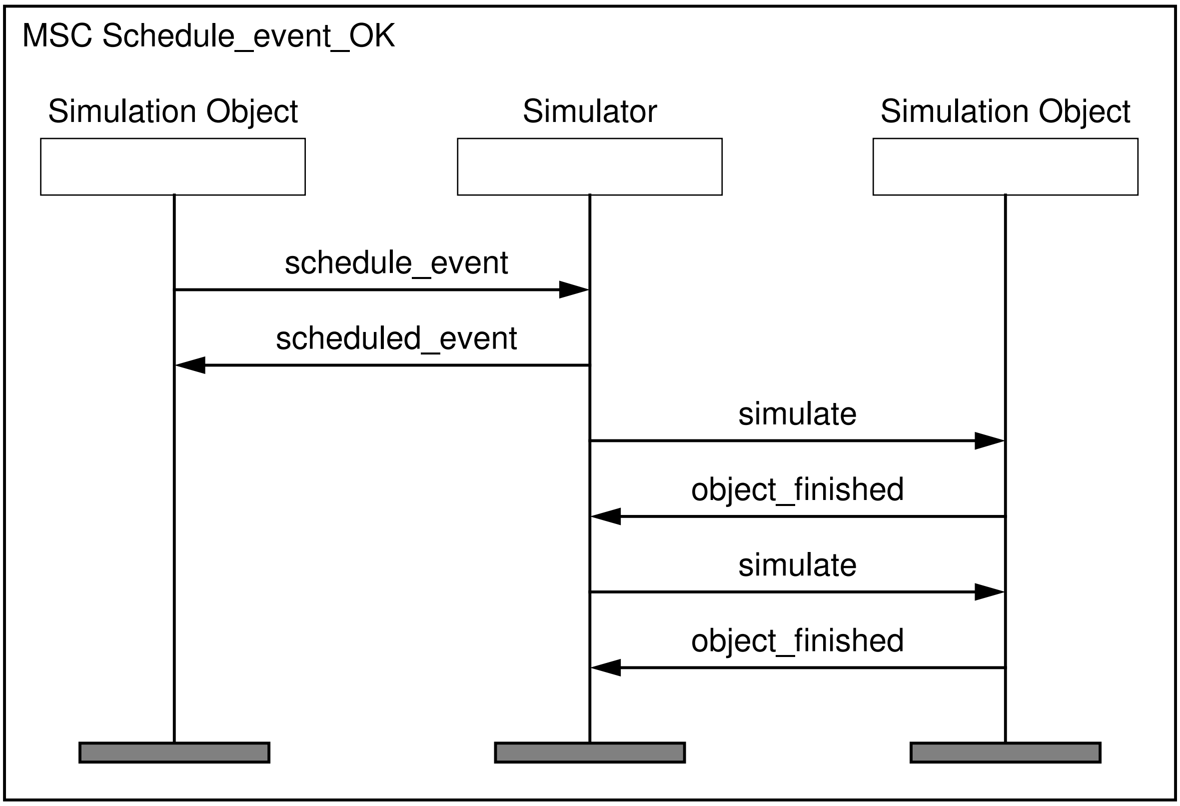
Every object may send a request for the current value of a visible variable, parameter or the invocation period of any other object via the corresponding set signal. The request is answered with the appropriate signal from the Values' group or remains unanswered if referring to a non-existing quantity.



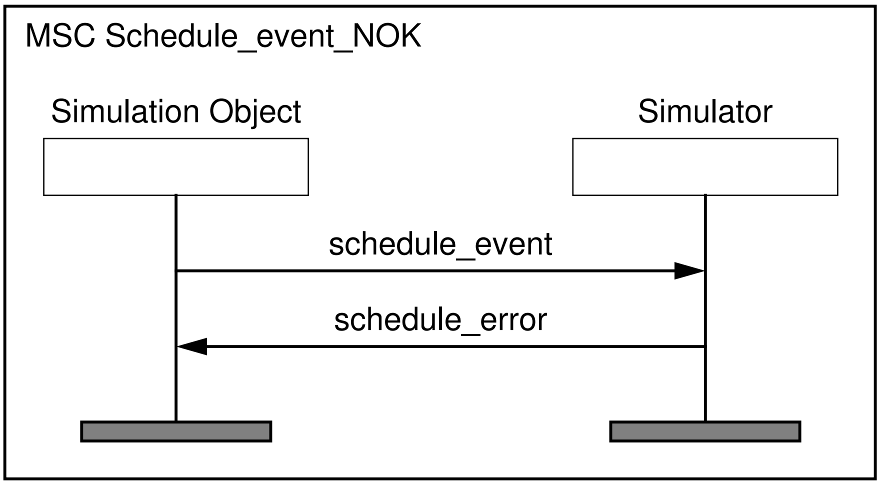
Every object may activate any other object to perform a single simulation step. As soon as the other object has finished its simulation, it sends a signal object_finished.



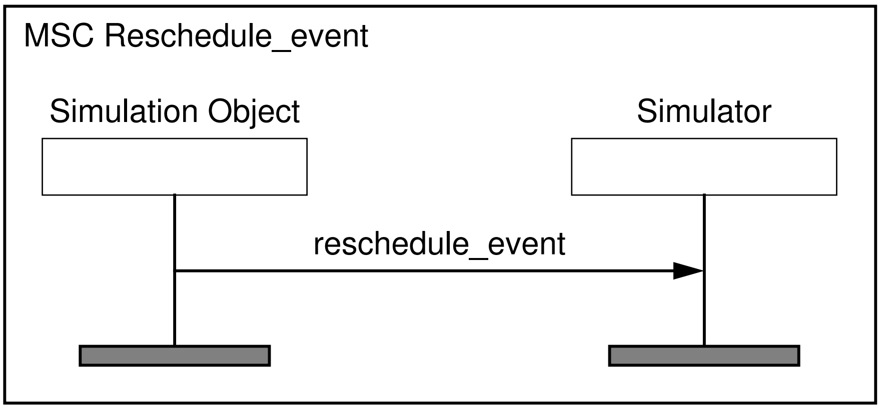
Every object may schedule simulate events for any other object. The simulator delivers the identification of the newly scheduled event. In our example, an object schedules a periodic event for another, and the first two invocations are shown.



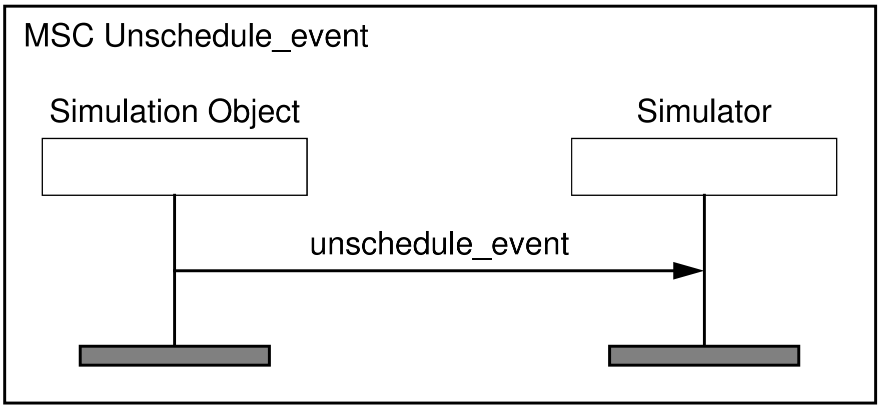
If, however, the event could not be scheduled, e.g. because of a reference to a non-existing object, the simulator sends a signal schedule_error instead.



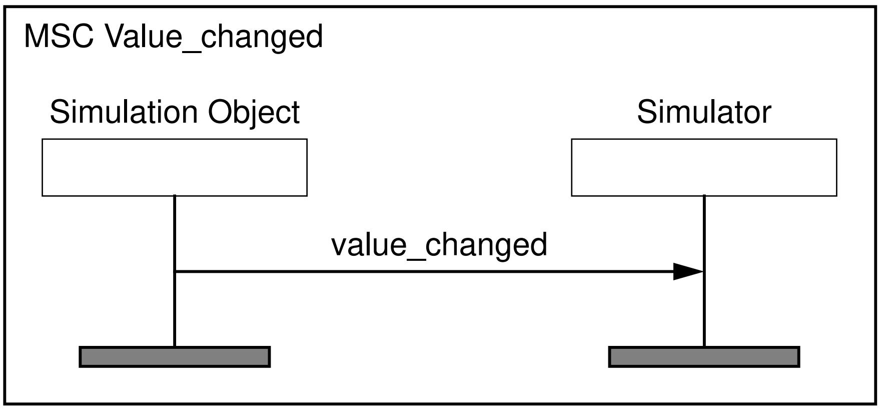
With the help of the event identification, it is possible to change the period of a periodic event that was scheduled earlier. In the successful case, there is no acknowledge. But a reference to a non-existing event leads to a signal schedule_error.
Re-scheduling is especially useful when an object detects that for its new state the current maximum invocation period is no longer appropriate with respect to numerical stability and therefore needs to change its own periodic simulate event.



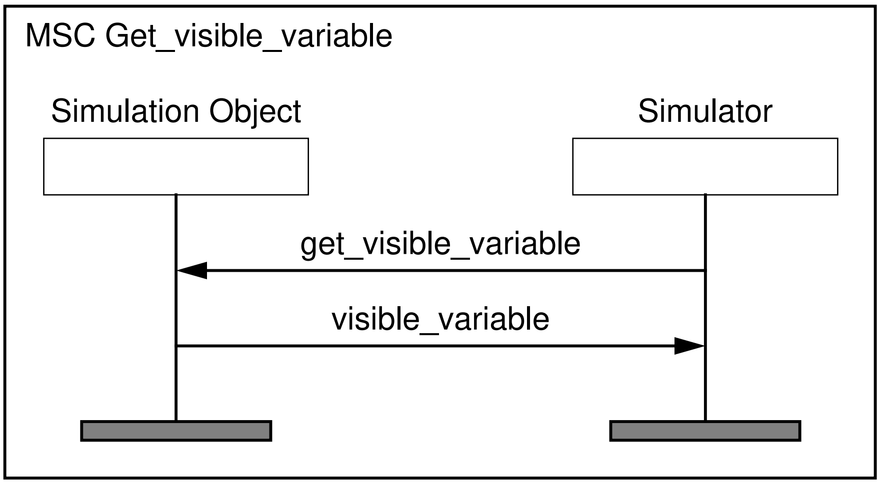
Using the event identification, it is also possible to remove arbitrary events that have been scheduled earlier. Just like before, there is only a reply in the case of failure.



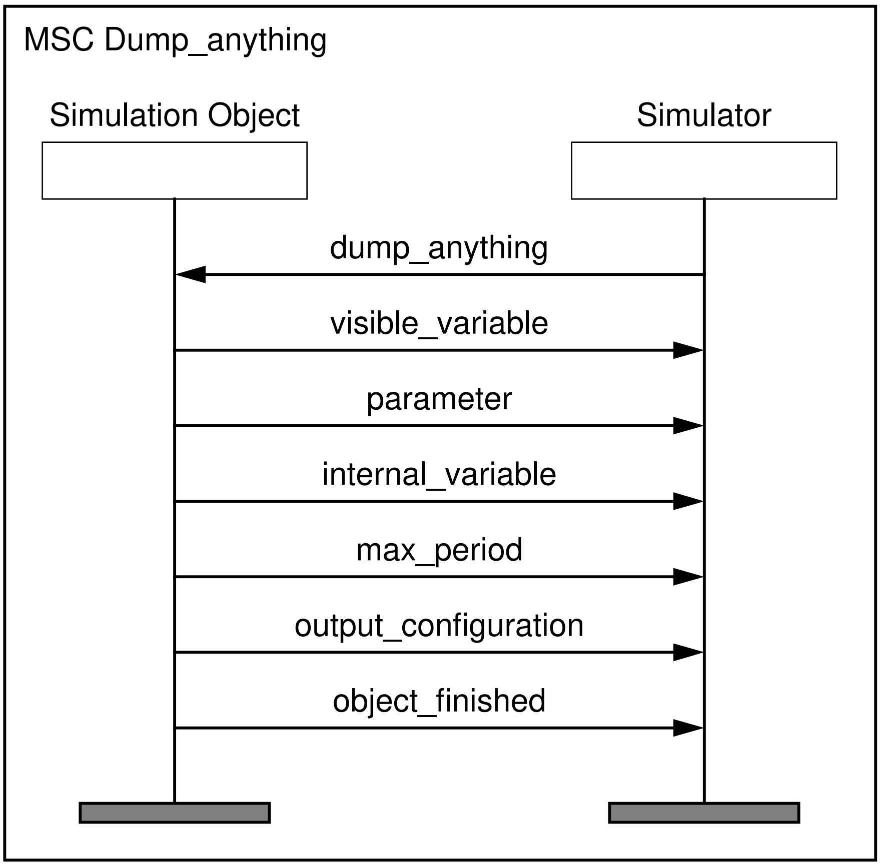
Finally, the signal value_changed indicates that a variable which is under observation of an event stream via “output on change” is going to alter its value as soon as the simulation time advances. If the future change is calculated when the object is active at time t, the new value will be valid at time $t+\delta$, where δ denotes the resolution of the discrete simulation time; but the signal value_changed is already sent at time t.



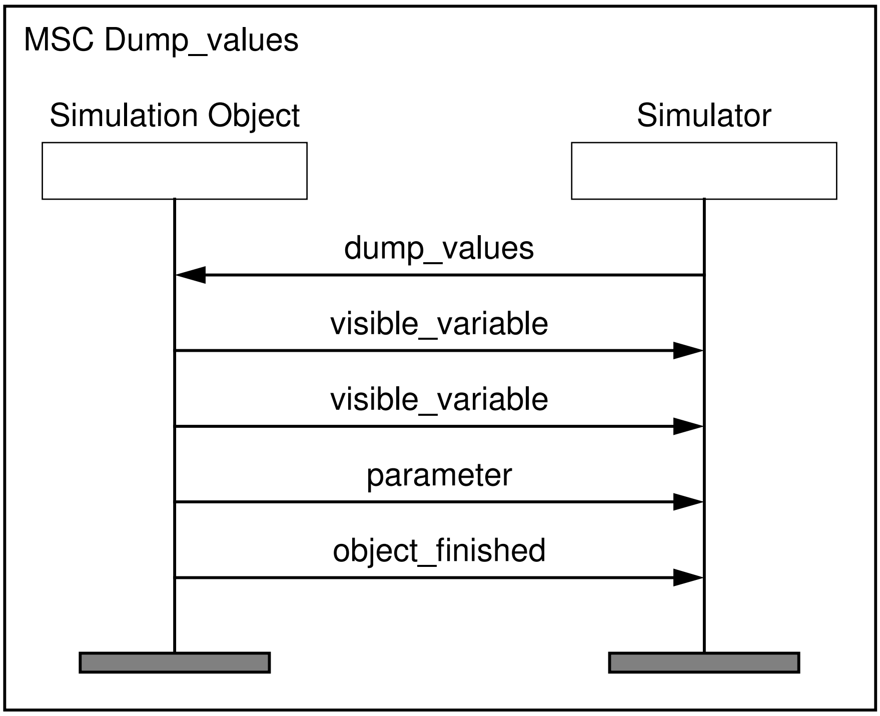
Actions without activation. The realization of get events on the lowest level is straightforward: The simulator issues a get_visible_variable signal; and the corresponding object answers with a signal visible_variable or not at all if the visible variable is not part of the object.



The last but one chart in this subsection shows how the objects are polled for their state information in case a system dump is written. After receiving the signal dump_anything, the object sends the current values of all its parameters, visible and internal variables, its invocation period and its output configuration. In this example, the object has only one attribute of each type.

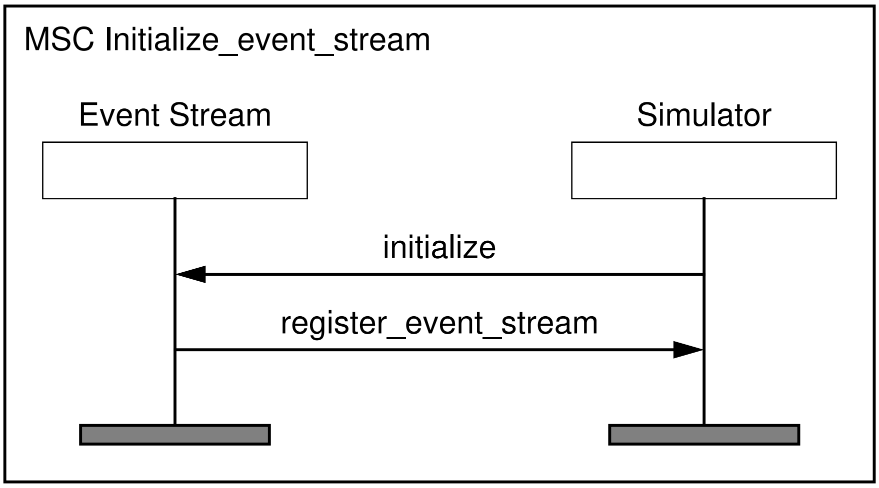


A similar situation arises when a restricted dump shall be written. The object now receives a signal dump_values instead, and only the current values of visible variables and parameters are returned.

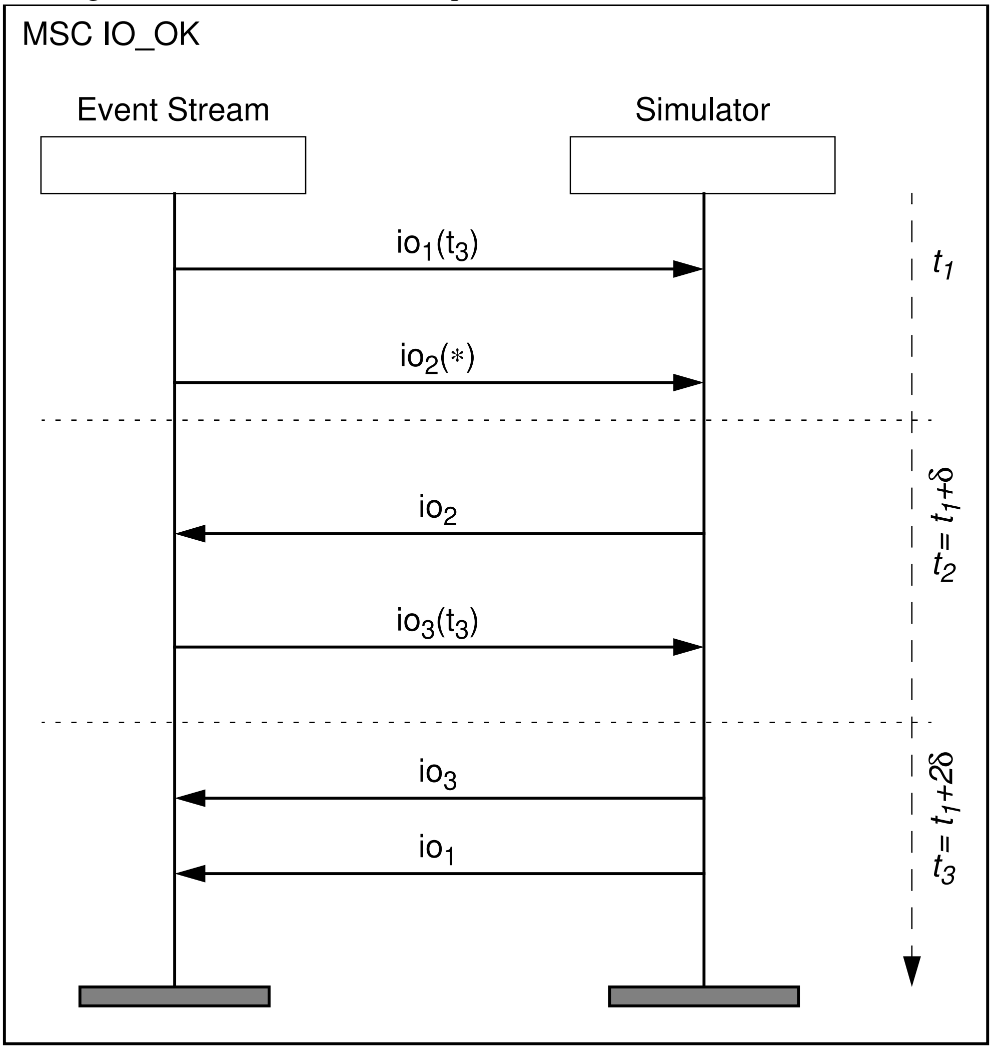


5.2 Event Streams

During the initialization phase, the simulator sends an initialize signal. The event stream performs its internal initialization and answers with a register signal revealing its name. Afterwards, the event stream must be ready for the simulation, which may begin immediately.

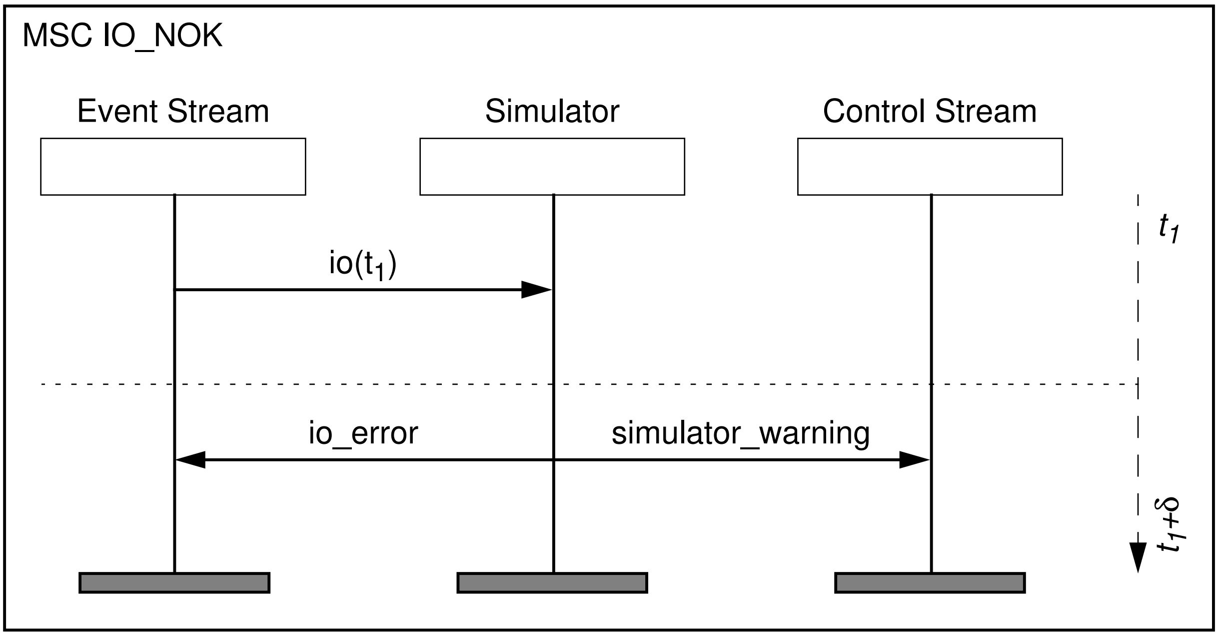


The following chart depicts a typical successful communication while the simulation is running. To demonstrate the dependency from simulation time, the usual chart semantics has been extended. Along the axis of real time, the equal intervals of discrete simulation time are given.

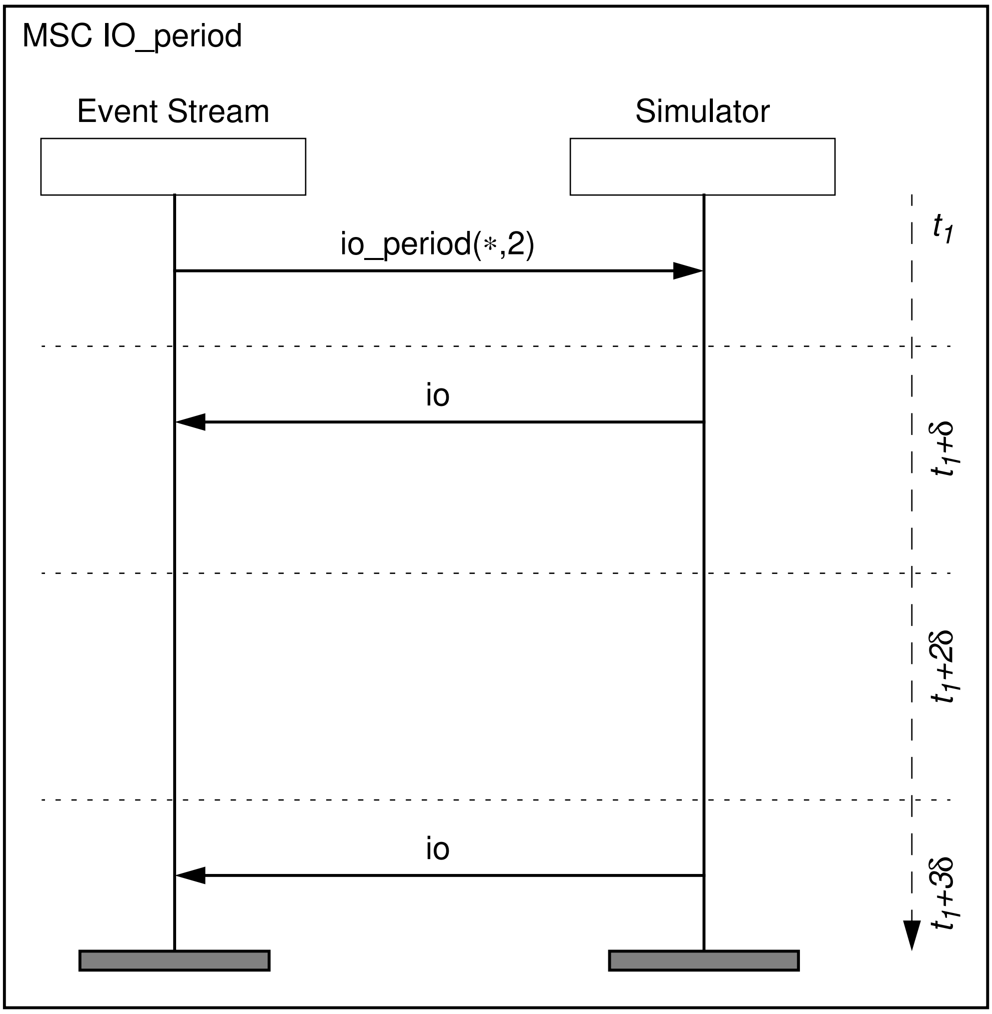


The resolution of the simulation time is denoted by δ. Furthermore, the timestamps of the simulator inputs have been added to the signals, and each of these carries the same number as its corresponding output. The output signals carry the timestamp of the simulation time during which they are sent. For simplicity reasons, we only show the case that the inputs are requests for values, as the corresponding timed simulator reaction refers to the event stream again. Otherwise, simulation objects are involved, which would enlarge the diagrams; but the timing is just be the same.
During simulation time interval t_1, two inputs are delivered, the first of which referring to the next but one time slice, the second to be considered as soon as possible. Consequently, the latter is answered during simulation time interval t_2, which is just one resolution tick later than t_1. Remember that the notion of the special time symbol ∗ – “as soon as possible” – has been fixed as at a later simulation time than the current one. At the end of the time slice t_2, the event stream sends another input to the simulator, which shall (and will) be responded at simulation time t_3, followed by the output in return to the first request.

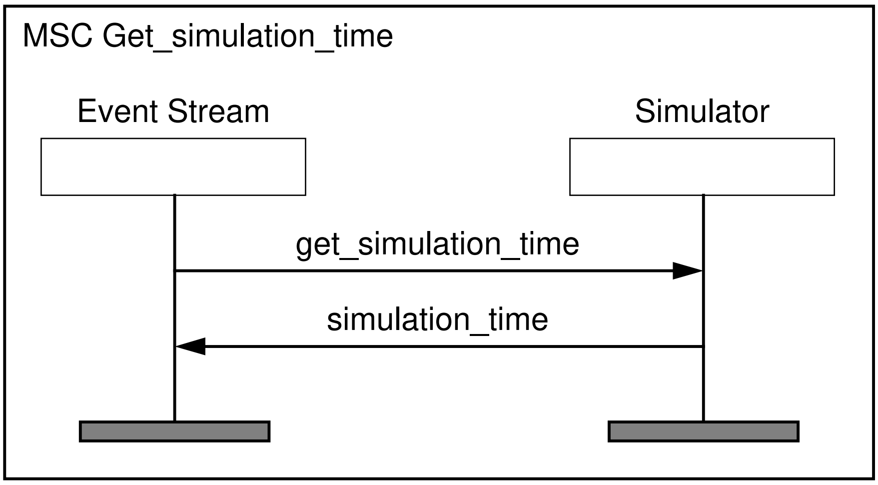
Inputs that are stamped with the current simulation time or even with a past timestamp are not accepted. The simulator returns a signal io_error and sends a warning to the control stream so that there all the warnings occurring in the system during the whole run can be collected. If necessary, the event stream now could also issue a warning of its own, see below for details. The moments of the advancing of simulation time in the chart are arbitrary with respect to the error and warning signals.



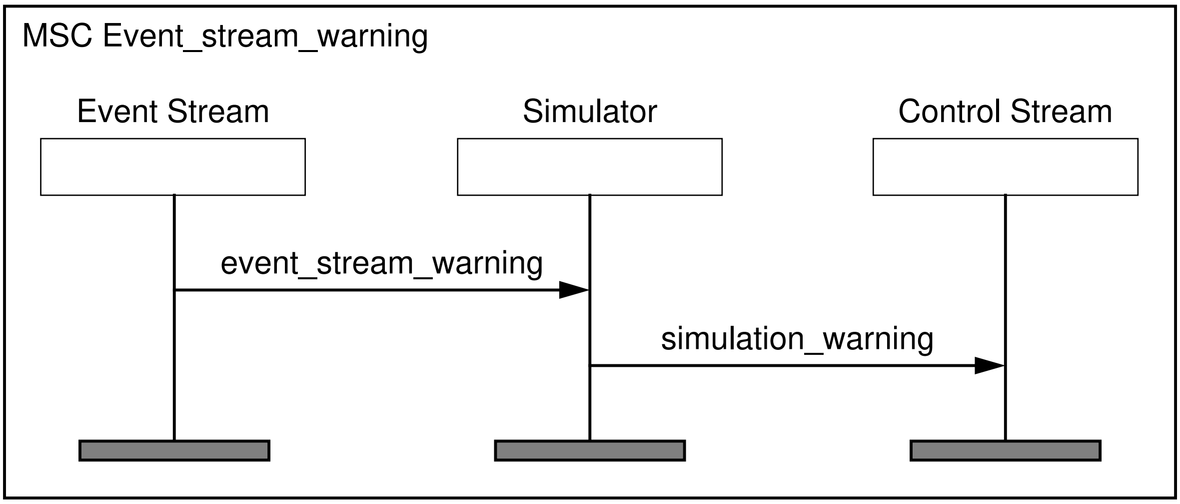
Every event stream may also send periodic output requests. In our example, at time t_1 an output is polled, starting as soon as possible and with a period of two units. The first output arrives during the subsequent time slice, the second two slices later.



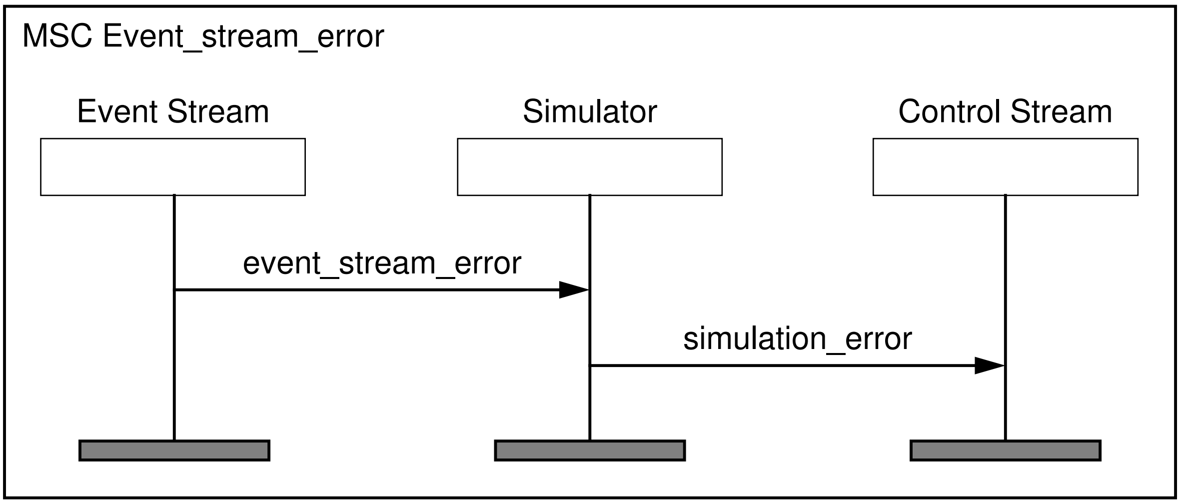
It is also possible to ask the simulator for the current simulation time, which will be returned immediately.



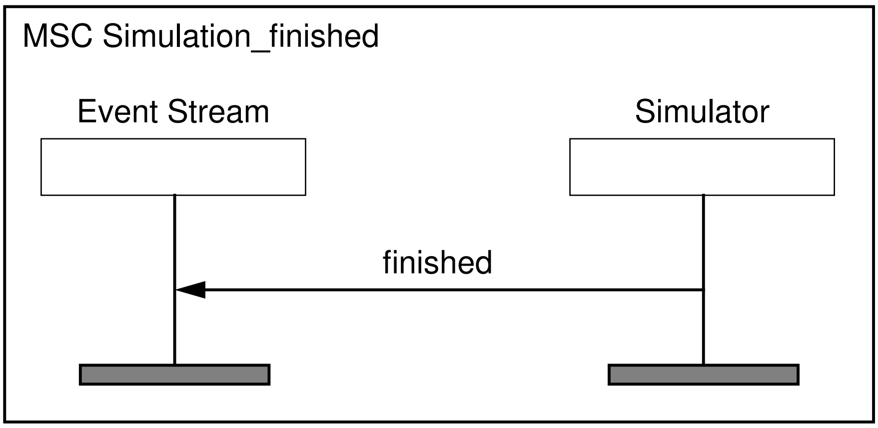
The event stream sends a warning signal to the simulator whenever an unusual situation arises in which it is yet sensible to continue simulating. The simulator furthers the warning to the control stream.



If the inconsistency is so severe that continuing the simulation run makes no sense, the event stream sends an event_stream_error signal.

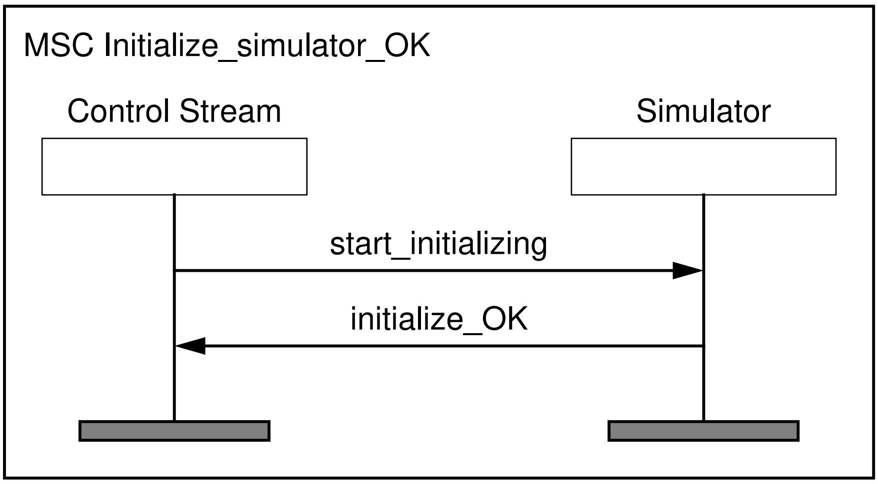


As soon as the simulation run is ended – as well if due to an error or if the end in simulation time is reached – the simulator sends a signal finished to the event stream.

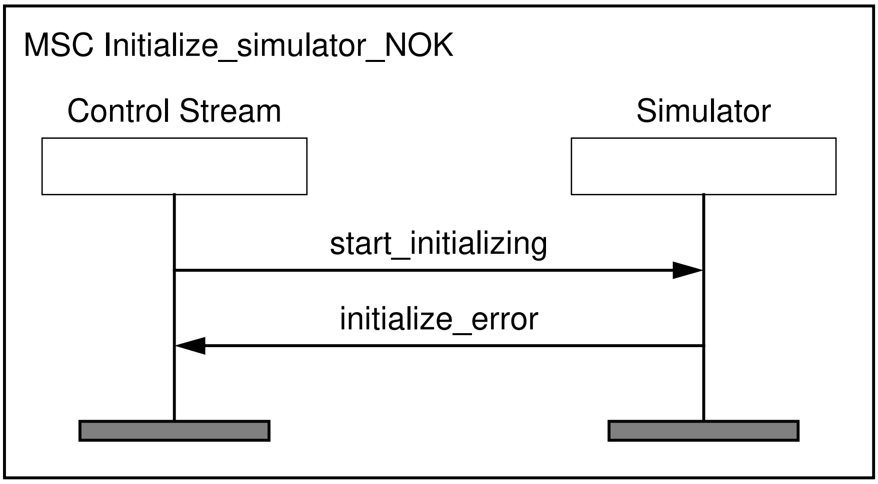


5.3 Control Stream

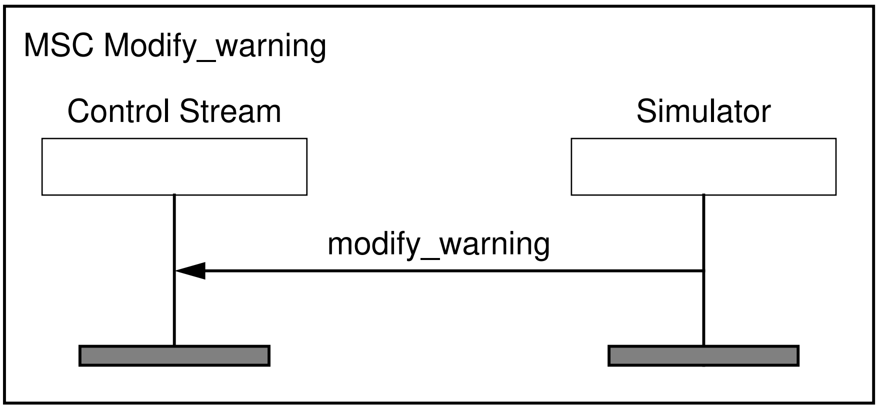
Initialization. First of all, the control stream sends a signal start_initializing to the simulator. The latter now performs its internal initializations and furthermore asks the event streams and simulation objects to do the same. Besides, the dictionaries for the mapping from external to internal references are established. If these operations can be completed successfully, the simulator issues a signal initialize_OK.



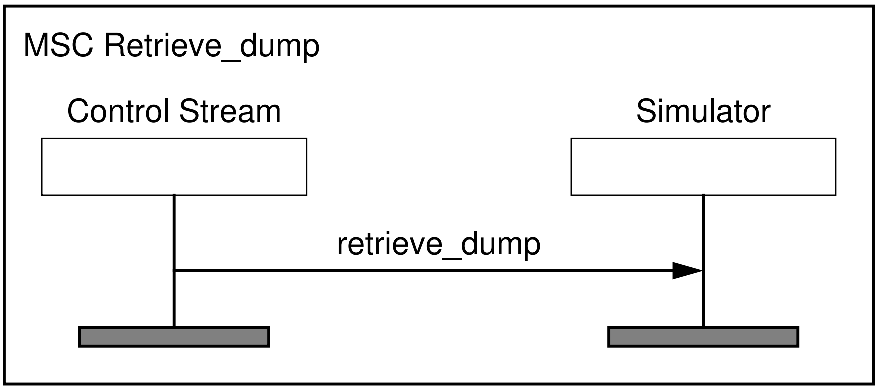
Otherwise, a signal initialize_error is delivered to the control stream, e.g. if a simulation object has a reference to another object that is not existing in the system. Consequently, the system start-up cannot be continued.



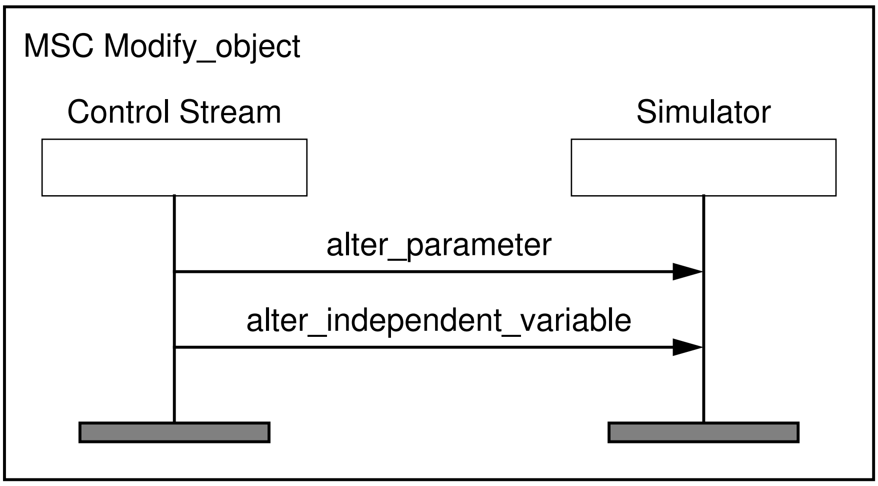
Modification. During the second phase, the user is allowed to retrieve dumps, to change some of the default simulation object and system parameters,[1] and to dump the current values.
All these actions may be arranged in an arbitrary order. Later re-settings are expected to overwrite earlier ones.
There will be no acknowledge if the operations are executed successfully. Otherwise, the simulator sends a signal modify_warning, and the simulator state is the same as before.



In the first chart, a dump of an earlier system run is retrieved.

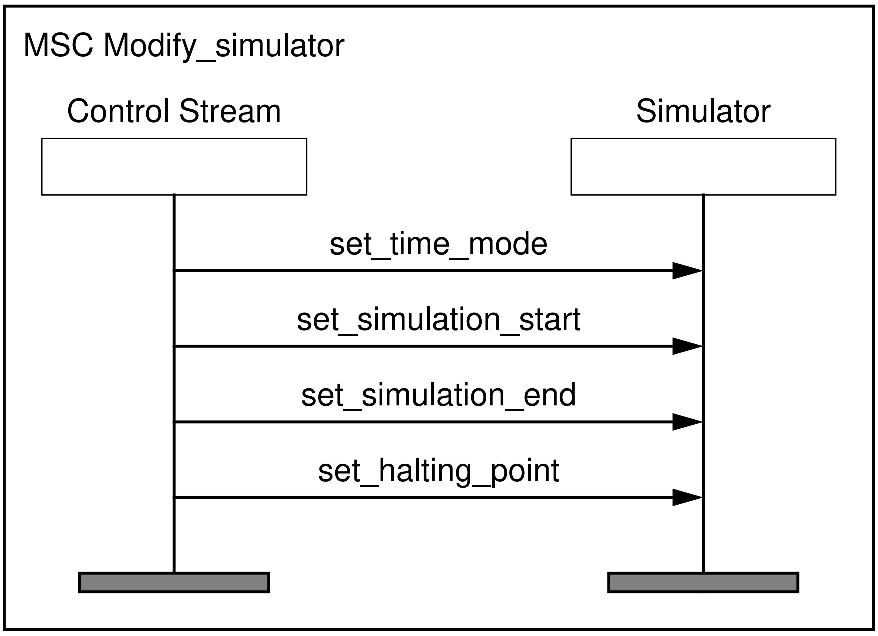


Second, a parameter in a simulation object and an independent variable are assigned new values.

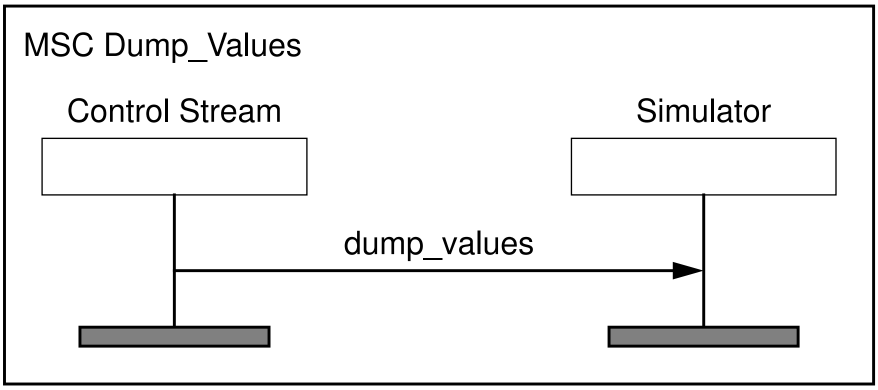


1. Note that the current state need not be consistent now, as re-establishing consistency is done later on.

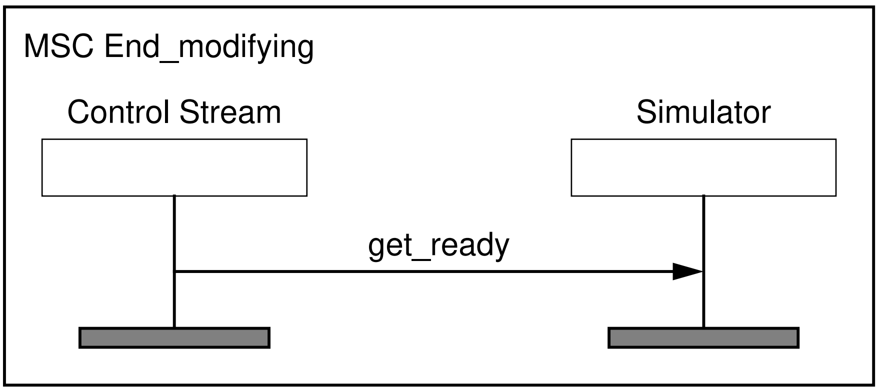
Furthermore, the advancing mode for the simulation time is changed, as well as the starting- and the finishing-point in simulation time. A future halting-point is set.



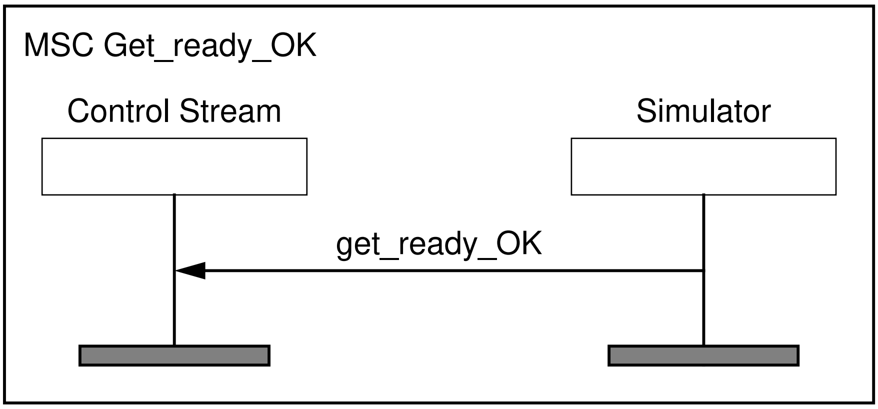
Now, a restricted dump is to be written. A complete dump serving to re-start an identical run could be output as well using the dump_anything signal instead. As we expect complete dumps to represent only consistent system states, the simulator would execute that command just before the simulation start.



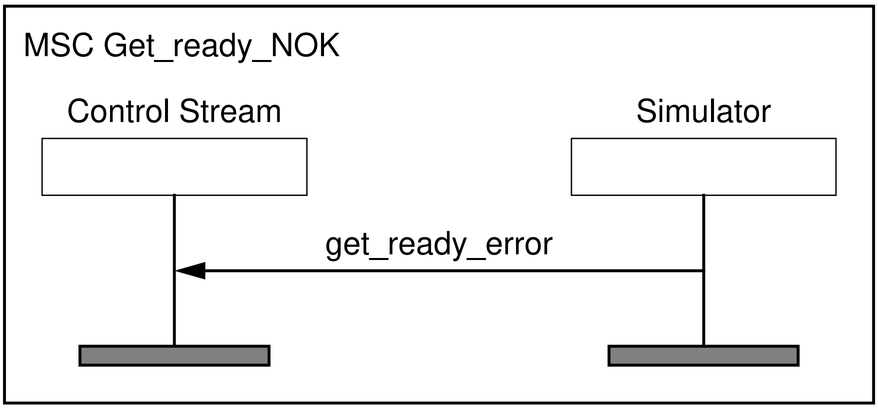
Finally, the signal get_ready ends the modification phase.



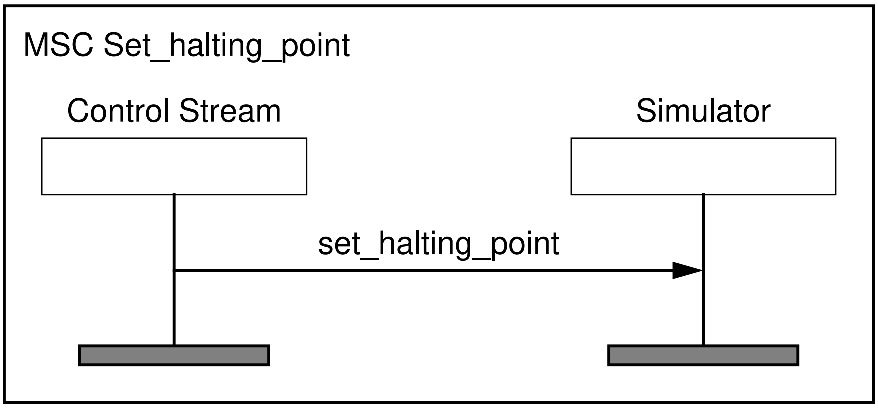
Getting ready. The simulator now enters the stage where a consistent state of the whole system has to be re-established. The initial state may be looked upon as consistent as well as the one after retrieving a dump. However, the manual changes of installation parameters and independent variables in some of the simulation objects might have led to inconsistencies. This is why now all the objects are asked to make their state consistent.
If the objects succeed in doing so, the system run can continue. In case a complete system dump was requested in the previous stage, it will be written onto the file system right now.



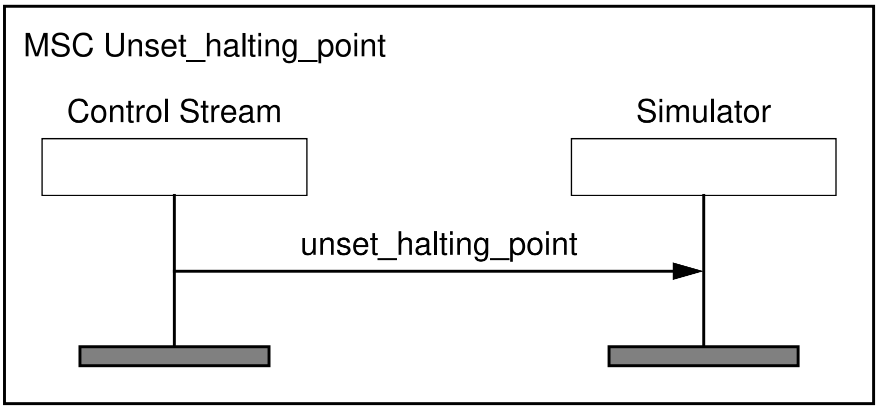
Otherwise nothing remains to be done.



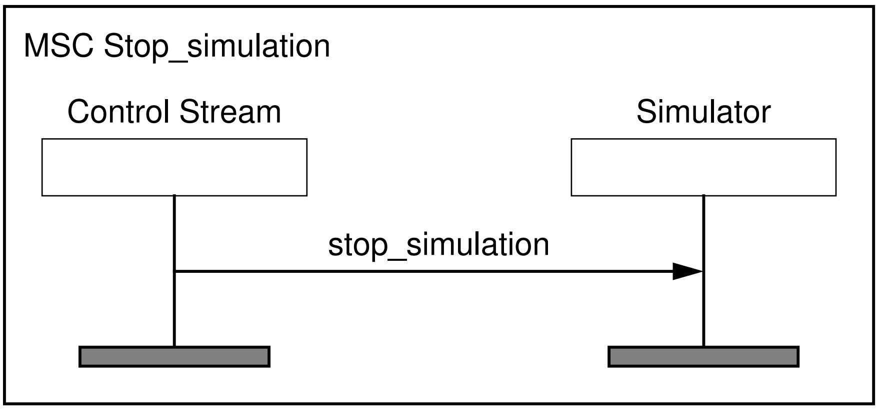
Simulation. In the following, the different communications between control stream and simulator will be presented. The signals on the one way represent commands changing some of the current simulator's setting. A group of signals allows interrupting the simulation run. Furthermore, there are signals to manipulate objects and to request dumps. These signals are not acknowledged in the successful case. Otherwise, a warning is issued, and the system state remains unchanged. The other signals on the way back denote among other things success and failure.
The first chart shows how a request for the setting of a breakpoint is delivered. If the specified simulation time has already passed by or another breakpoint has already been specified for that very simulation time, a warning will be returned.



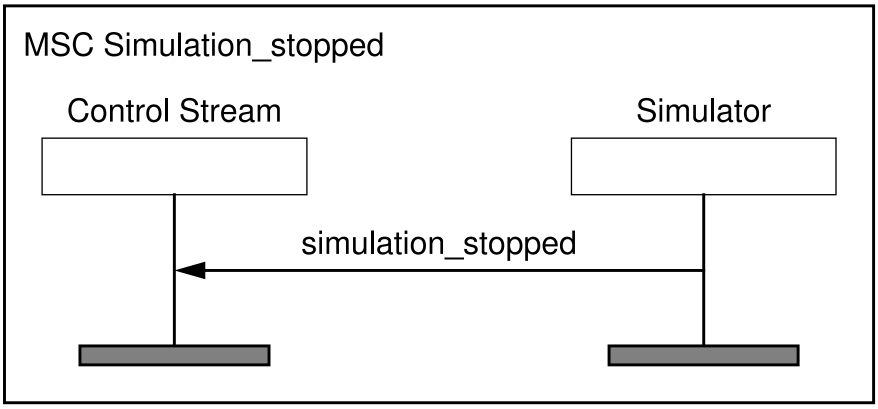
Breakpoints that have been set earlier and do not refer to the past may be removed; other removal tries will yield a warning.



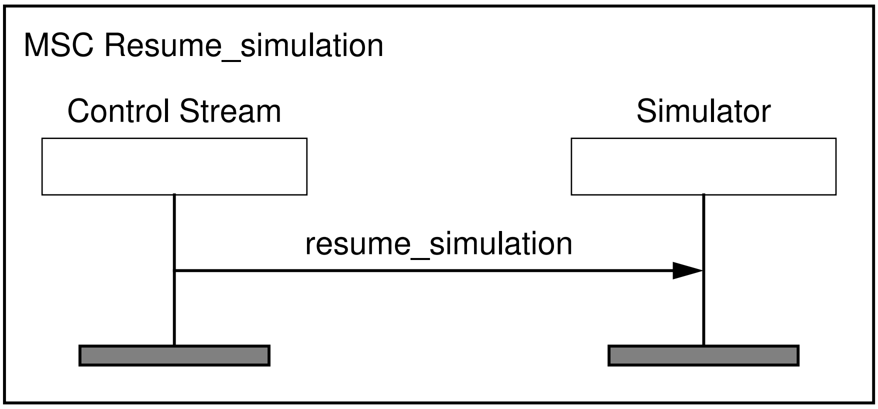
In the next chart, a stop command is issued. That is a request to interrupt the simulation immediately.



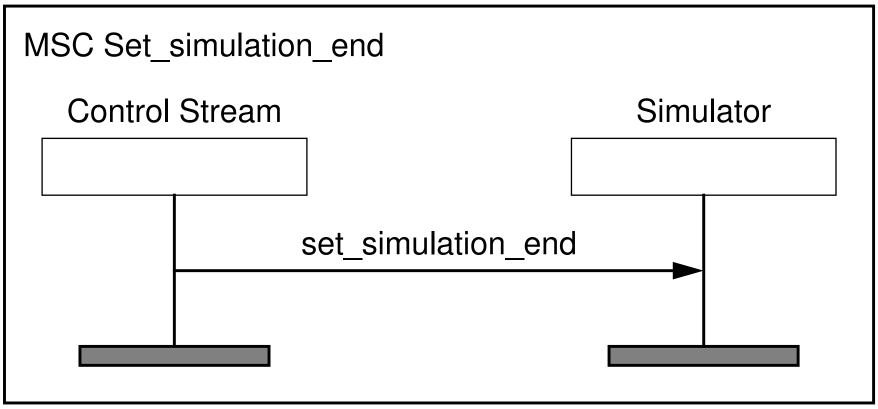
Having reached a specified breakpoint or being stopped manually, the simulator sends a signal simulation_stopped. There are no commands working only when the simulator has stopped.



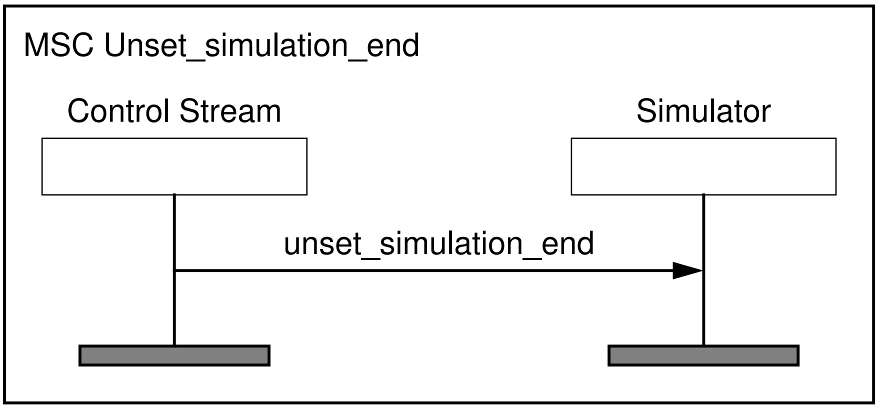
If the simulation is stopped, the signal resume_simulation will ask the simulator to continue. If the simulation is running, a warning will be returned.



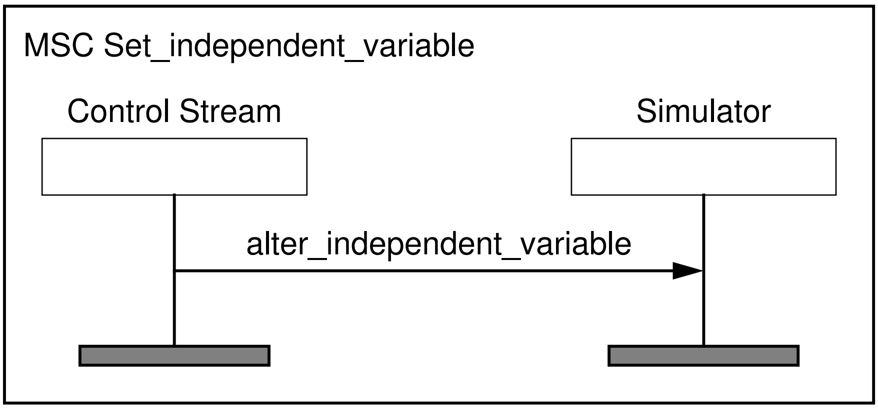
It is possible to re-specify the end of the simulation run. If the specified simulation time has already passed by, a warning will be issued. If it is the current one or the special time symbol $*$, the system should shut down immediately, even if the simulator is stopped.
However, if a halting-point is specified for that very simulation time the simulation end is set at, the simulator should enter the halting stage first, as one might be willing to write a final dump of the visible variables before the shutdown.



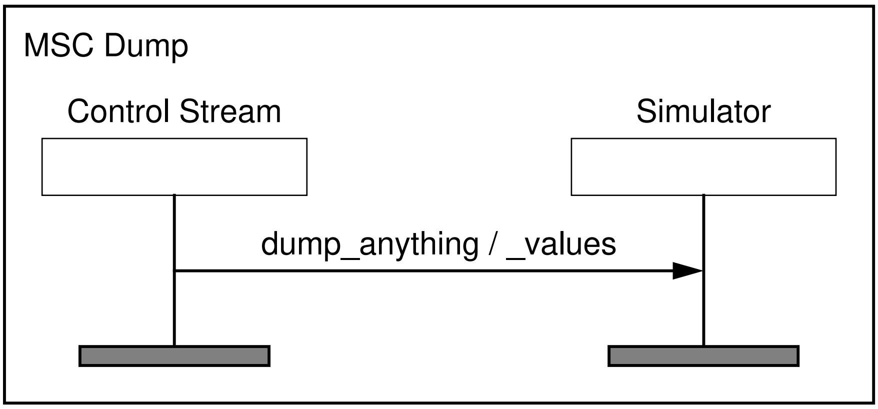
The simulation can also be run forever.



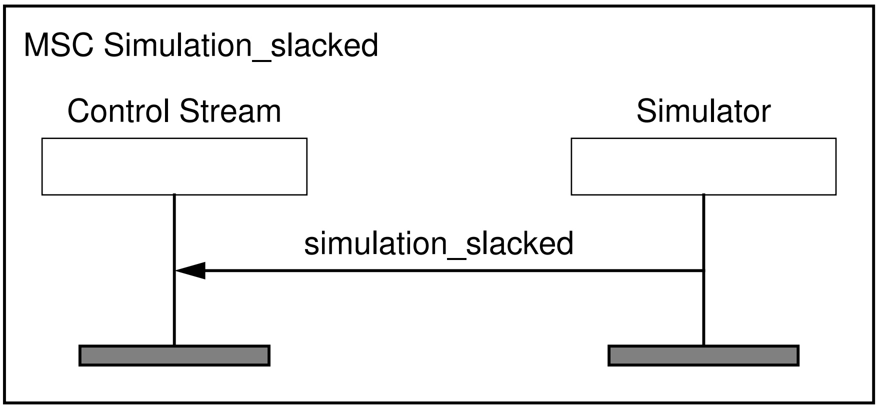
The values of visible variables that represent independent quantities may be changed via the signal alter_independent_variable.



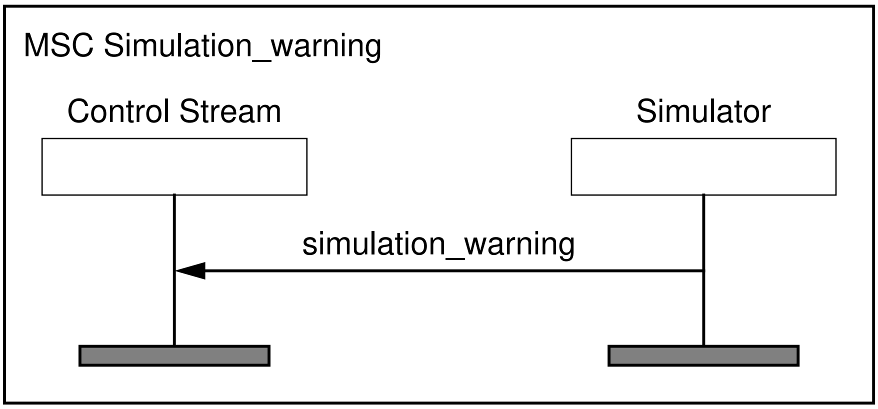
In the next chart, the dump signals express the request for writing a complete image of the current system state or a restricted one including only the values of parameters and visible variables.



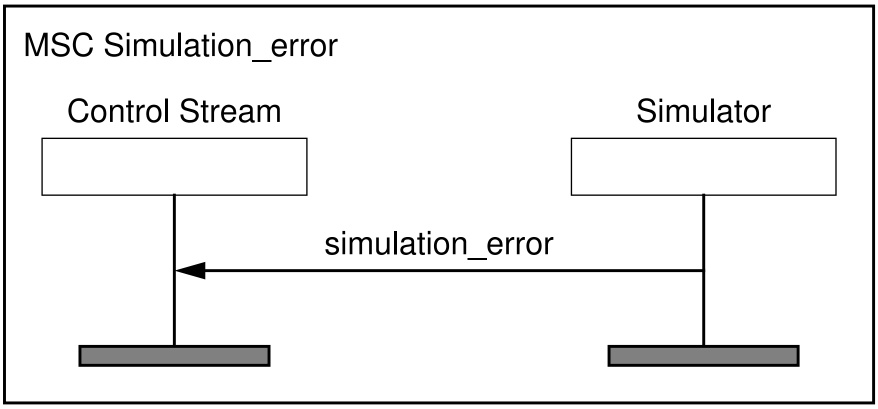
The first of the remaining signals from the simulator to the control stream is sent when the simulator is run with a close binding between real and simulation time and the maximal allowed deviation is exceeded. If it was specified to abort the simulation in such a situation at the beginning, a signal simulation_error will be issued afterwards.



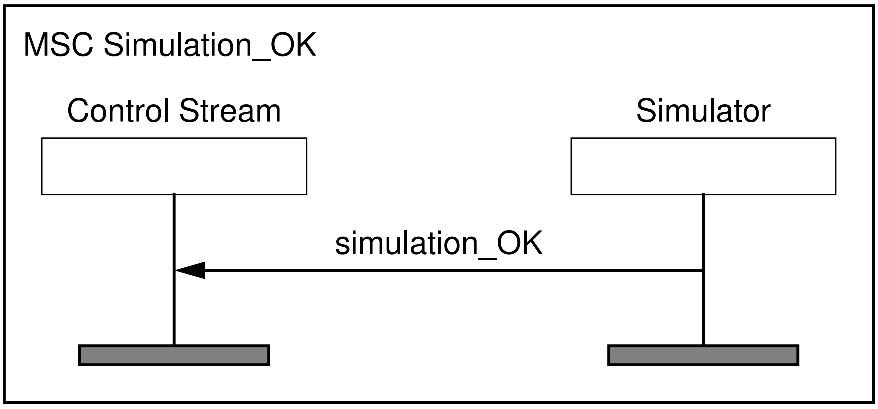
The signal simulation_warning is sent whenever an unusual situation is reached in which however it still makes sense to continue the simulation.



Otherwise the simulation is aborted, and the signal simulation_error is sent.



Finally, the signal simulation_OK indicates that the simulation run has finished unerroneously.



6 Behaviour Projection

The following subsections include partial descriptions of the behaviour of the simulator which are seen in the three views of the surrounding entities: simulation objects, event streams, and control stream.

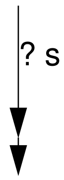
The state names init, mod, and ready appear in more than one subsection. In fact they denote stages in the different behaviour projections which necessarily coincide (and have already been suggested at the beginning of the previous section), and they precede the actual simulation. A closer look at the transitions towards such a state reveals that there is always exactly one view in which this state is reached via an input. Immediately, the corresponding state is accomplished in the other views as well, but just by an output transition.

The state name error denotes a stage when the simulation has to be aborted for the corresponding reason, whereas end is the ordinary exit.

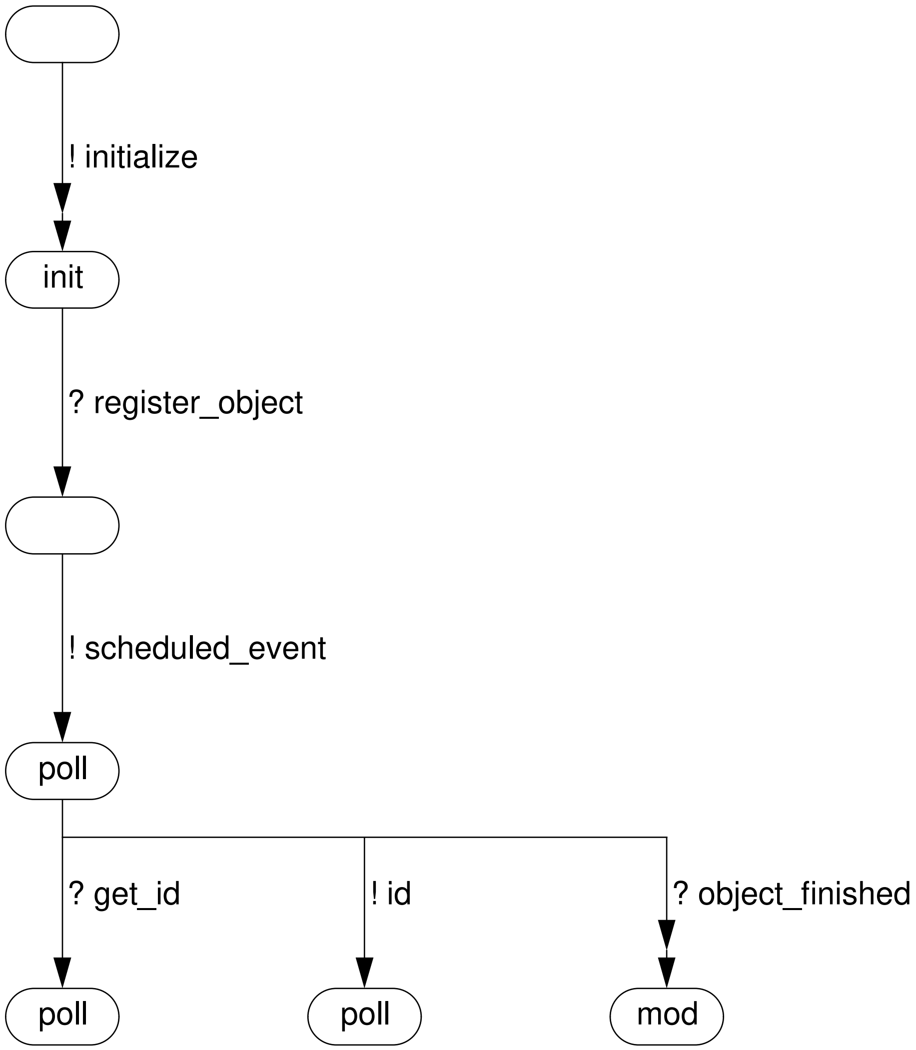
6.1 Behaviour in Relation to the Simulation Objects

"To derive a more complete definition of interface behaviour from a set of sequence charts one must focus on one object at a time and the interactions it is involved in." (Bræk, Haugen: Engineering Real Time Systems, p. 87) This is all what transition charts are able to express and restricts their application to the interactions between the simulator and a single simulation object. However, we wish to distinguish those transitions that will reasonably be done for all objects.

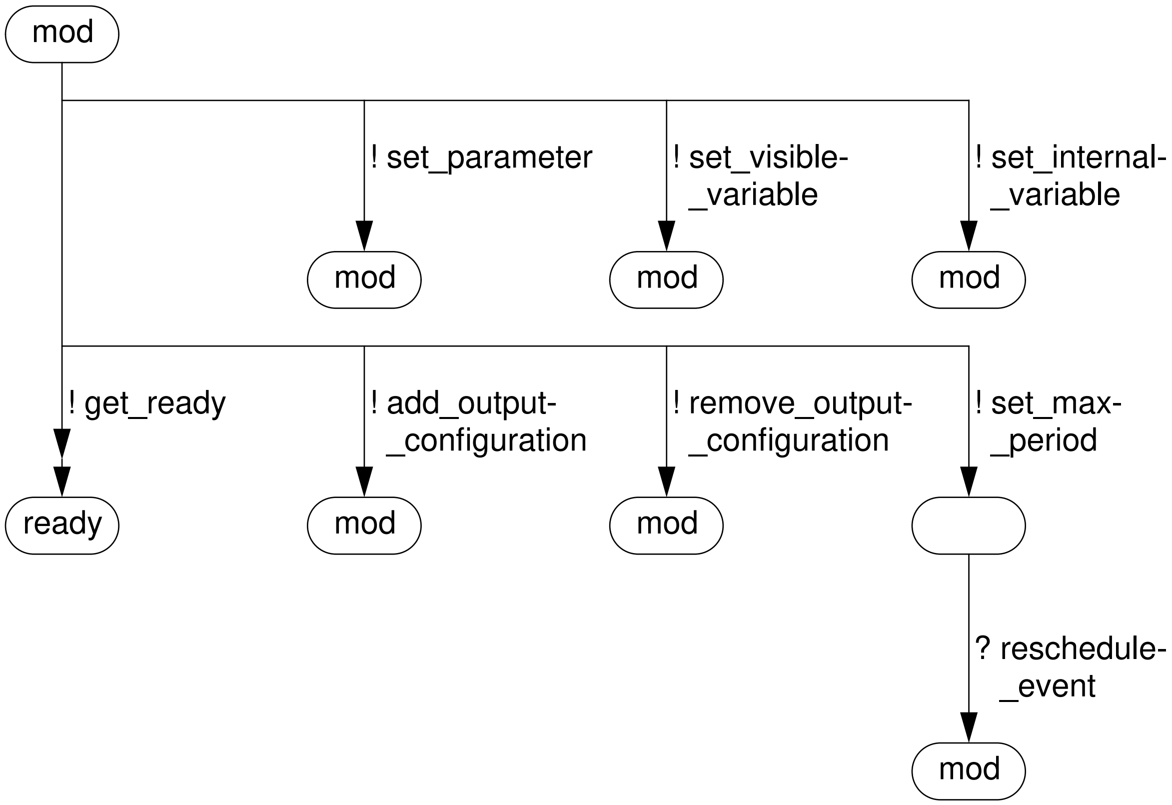
An input-triggered transition will be marked with two subsequent arrowheads such as depicted on the right if the reached state is left not before that specific input signal arrives from every simulation object. An output-triggered transition is marked that way if the simulator sends the output signal to all the objects simultaneously.



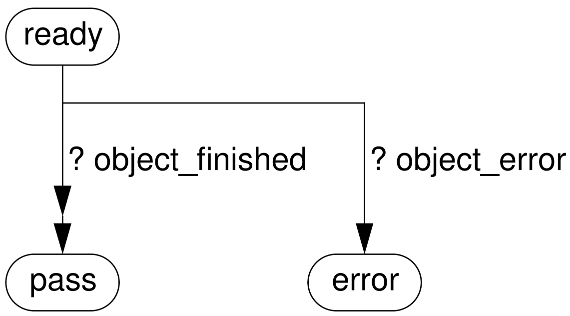
The first diagrams depicts the interactions in the initialization phase.



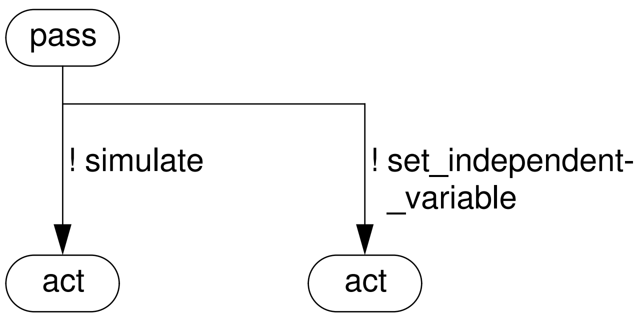
The second shows what happens at modification time.



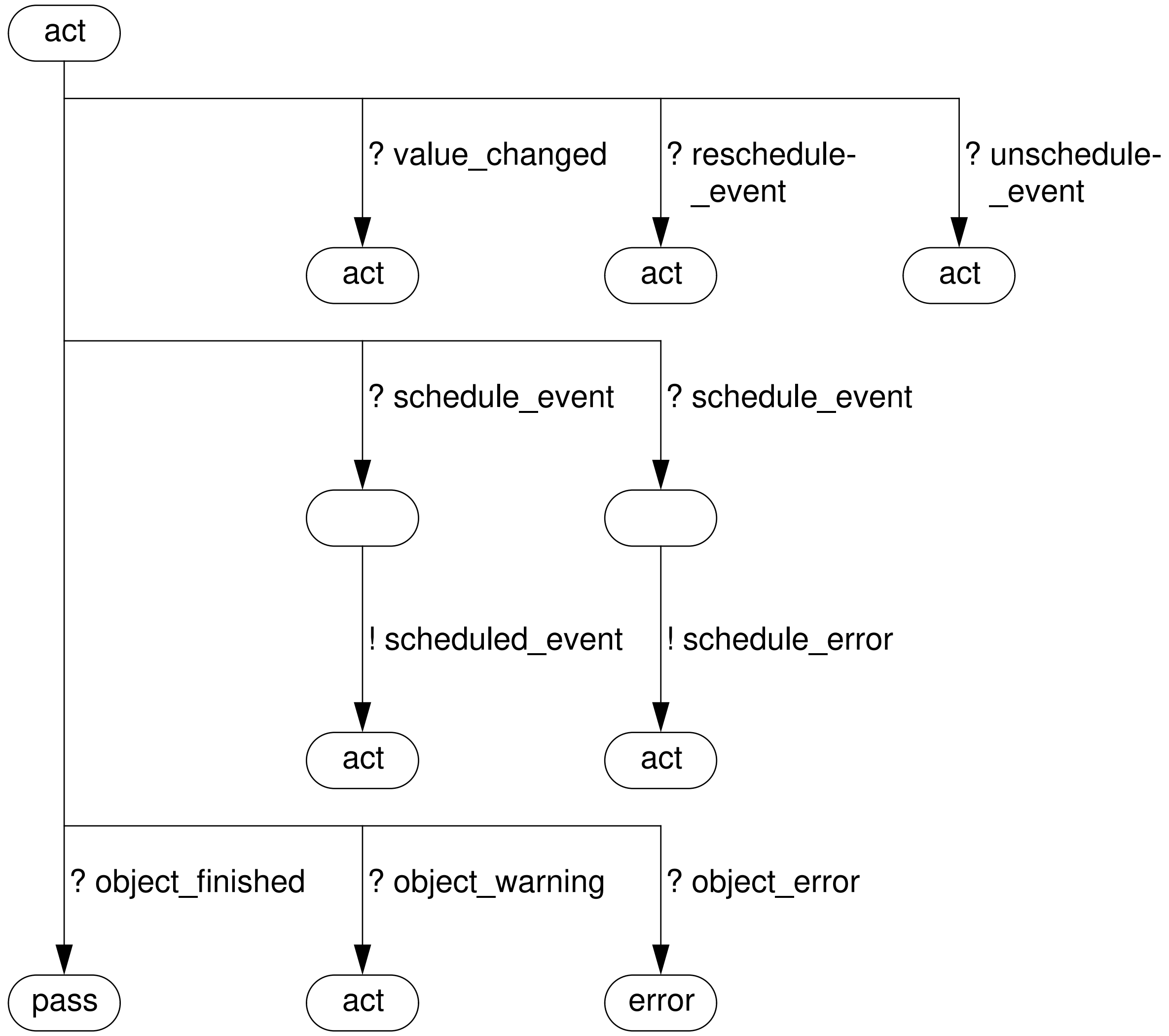
The third diagram depicts the behaviour when the objects are requested to re-establish consistency.



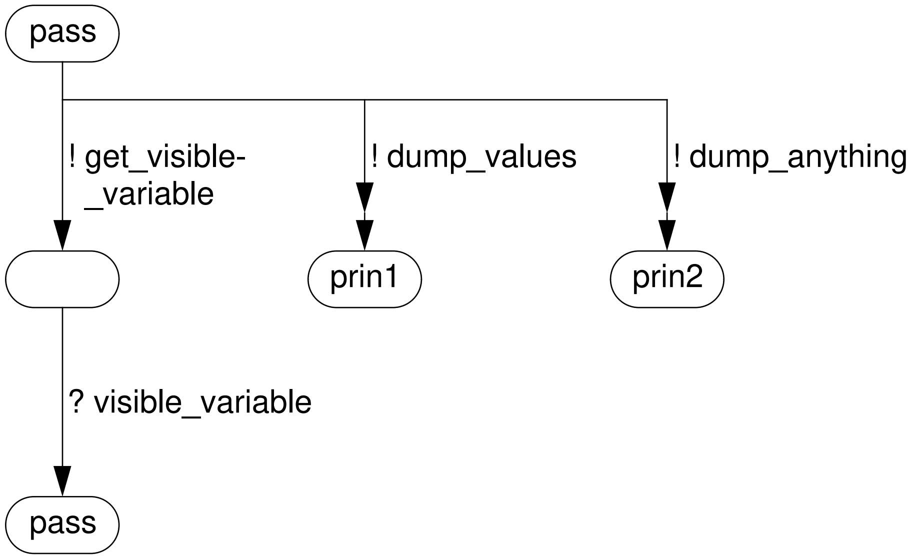
We now have reached the stage of actual simulation. The simulator activates a simulation object by one of the signals simulate and set_independent_variable.



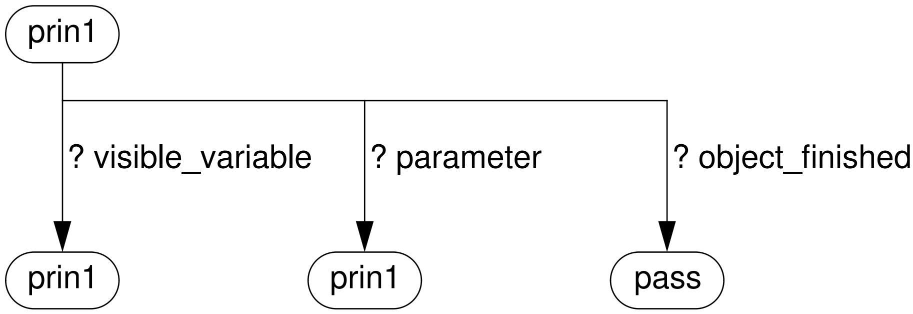
The next diagram shows the transitions from the state when a simulation object is active. Note that simulation objects are allowed to activate other objects without notice of the simulator, which cannot be shown here. However, they may send all the signals to the simulator that appear in the diagram as transitions from state act – except the signal object_finished. To become inactive, the latter is sent to the very object that has activated them.



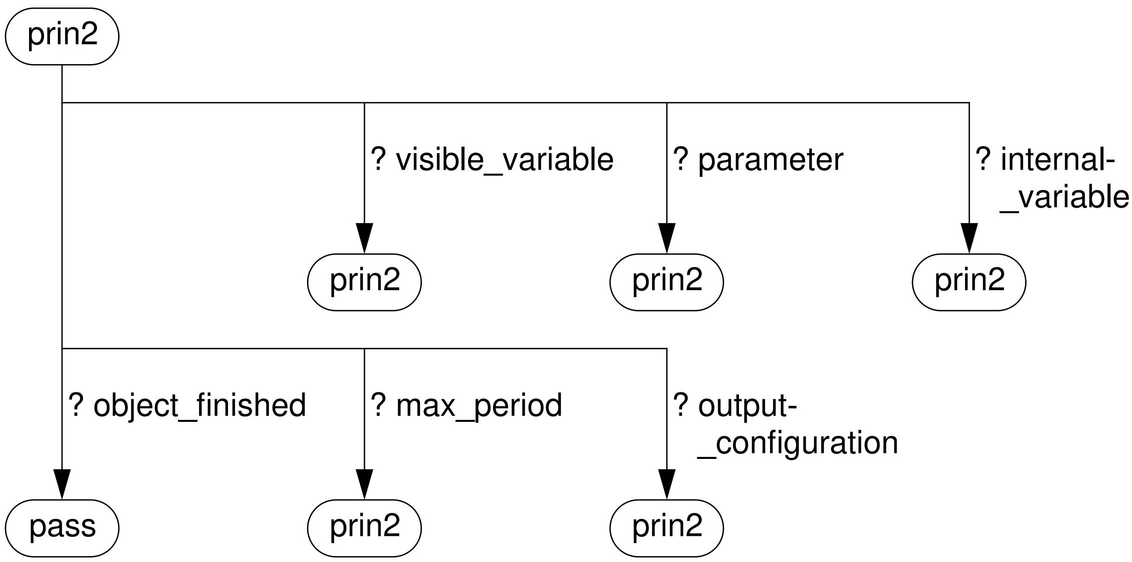
Finally, we have the “actions without activation”: reading the values of visible variables and requesting dumps.



Restricted dumps demand the transfer of the values of visible variables and of parameters only.

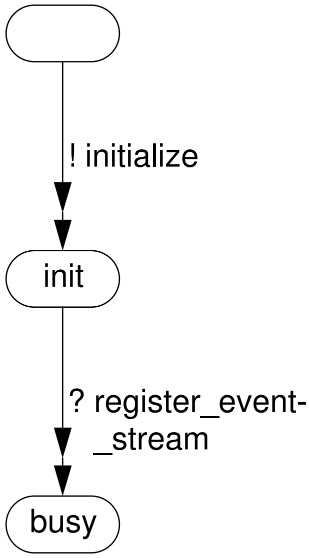


For complete dumps, the values of the internal variables, the maximum invocation period and the output configuration have to be passed on well.

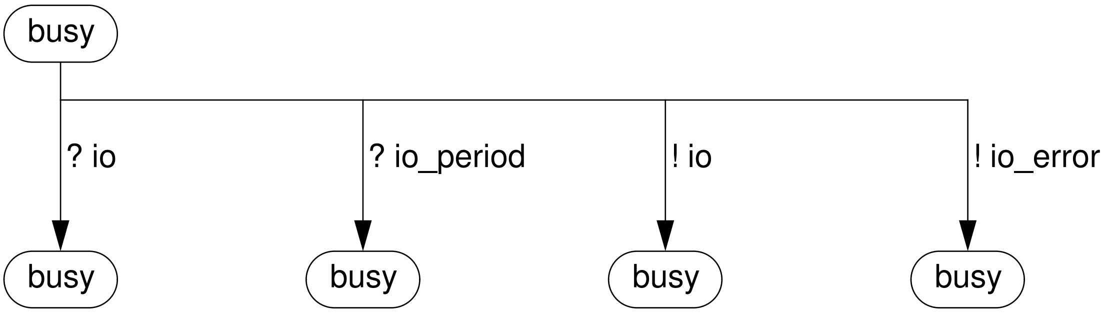


6.2 Behaviour in Relation to the Event Streams

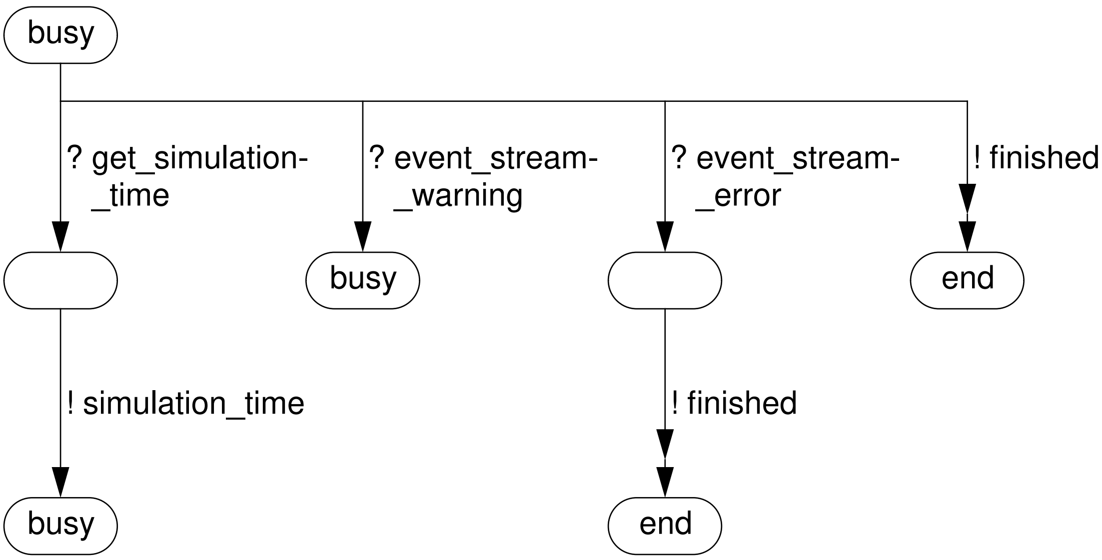
At start-up time, the simulator requests the event streams to perform their internal initializations and to report their names.



When simulating, the simulator and the event streams exchange commands and messages in a unique format; besides, error messages concerning unsuccessful set and get commands are handed on if necessary.

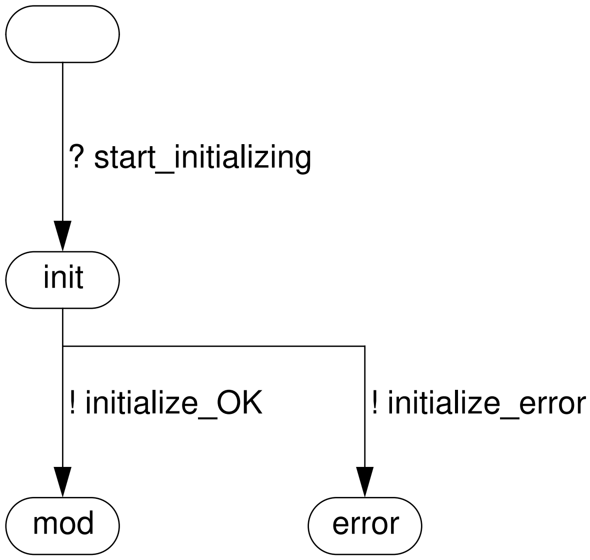


They also exchange a number of administration signals as long as the simulation is running.

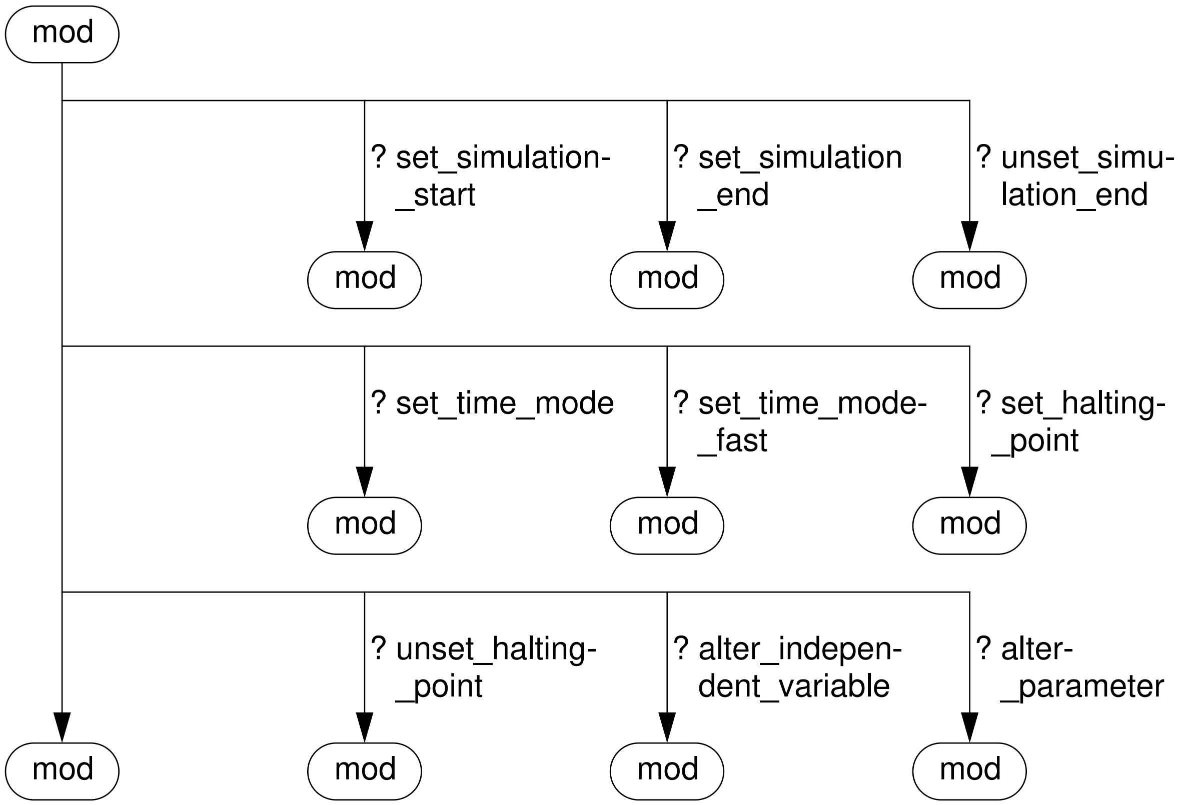


6.3 Behaviour in Relation to the Control Stream

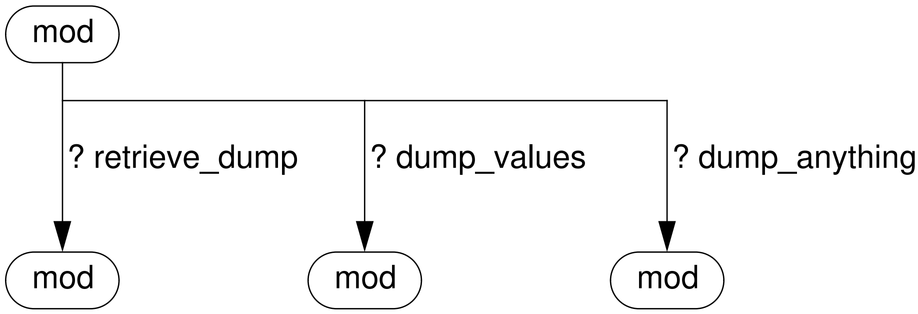
First of all, the control stream requests the simulator to initialize.



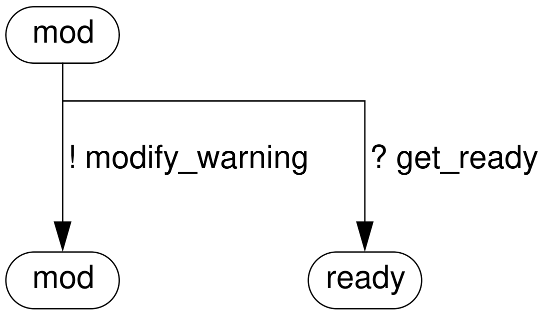
Second, the modification stage is entered, where lots of settings can be changed as well as parameters and independent variables inside of simulation objects.



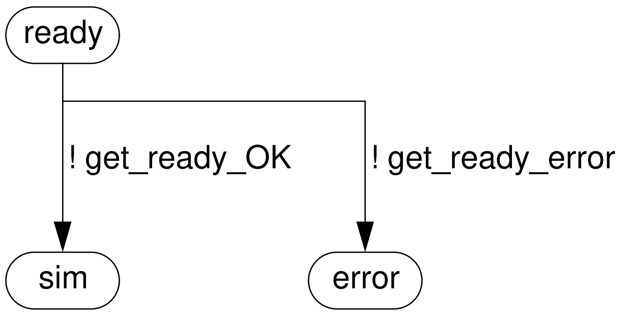
It is also possible to retrieve and to request dumps.



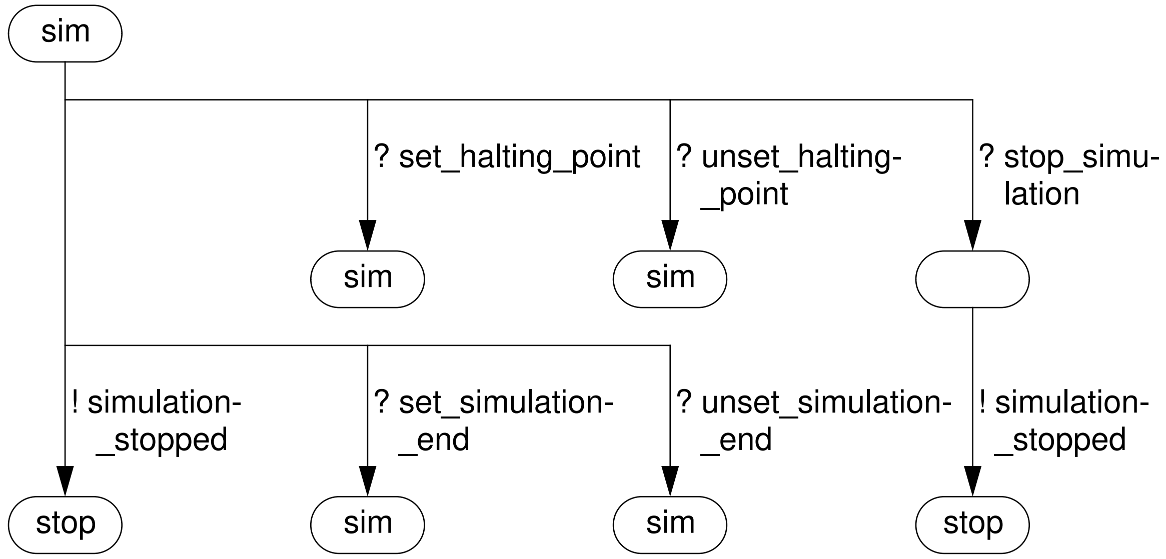
Anything strange is reported via the modify_warning signal. –The modification stage is left as soon as the simulator receives the signal get_ready.



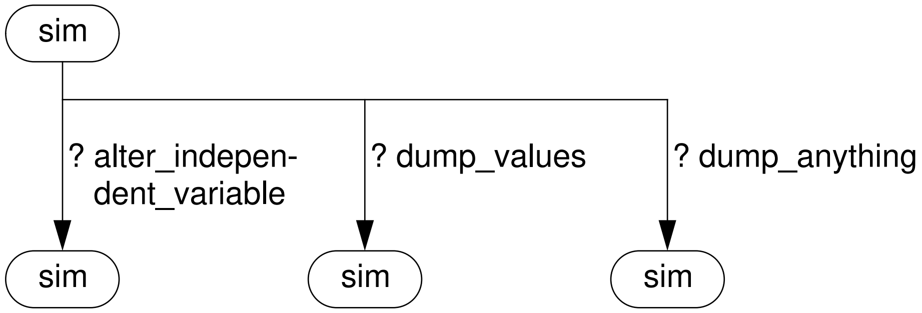
Now the simulation objects are activated to re-establish consistency. The actual simulation can start unless they fail to do so.



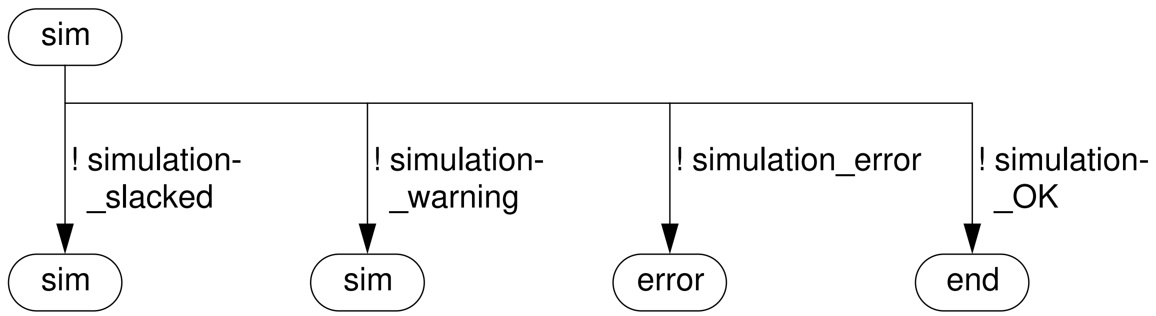
Having reached the simulation stage, it is possible to interrupt the simulator directly or via setting halting-points. Furthermore, it is possible to re-specify the end of the simulation run.



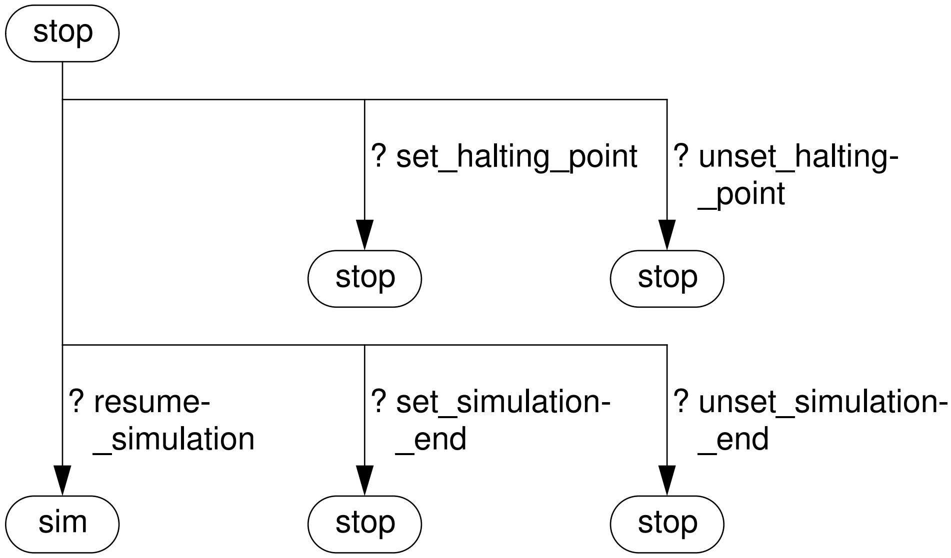
One may also change the values of independent quantities and request restricted or complete dumps.



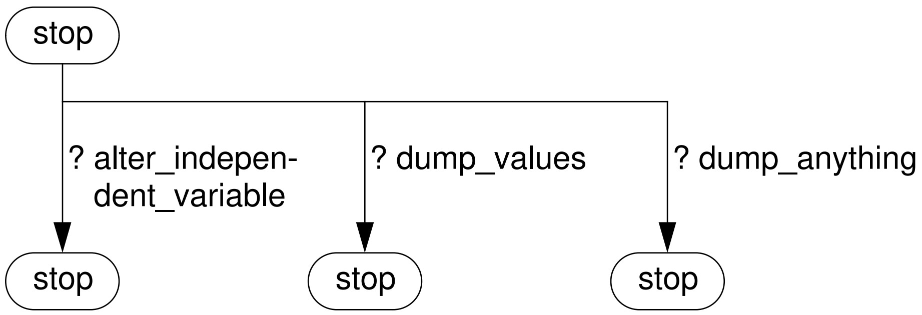
The simulator signals warnings and errors to the control stream. The signal simulation_OK indicates that the whole simulation has been completed successfully.



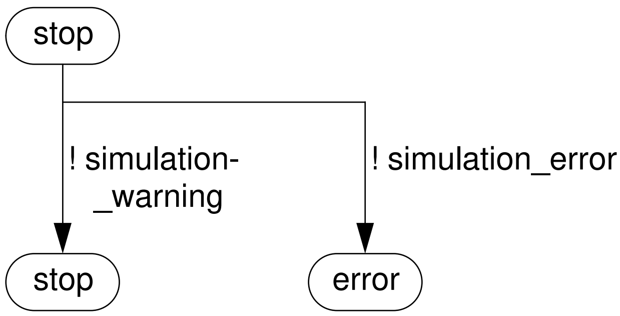
Most of the signals have the same effect when the simulator is not running. Let us have a look at those concerned with halting-points and the simulation end. Furthermore, the signal resume-_simulation yields a transition back to the state sim.



Setting independent variables and requesting dumps is just as above.



Warnings and errors may occur in this stage as well.



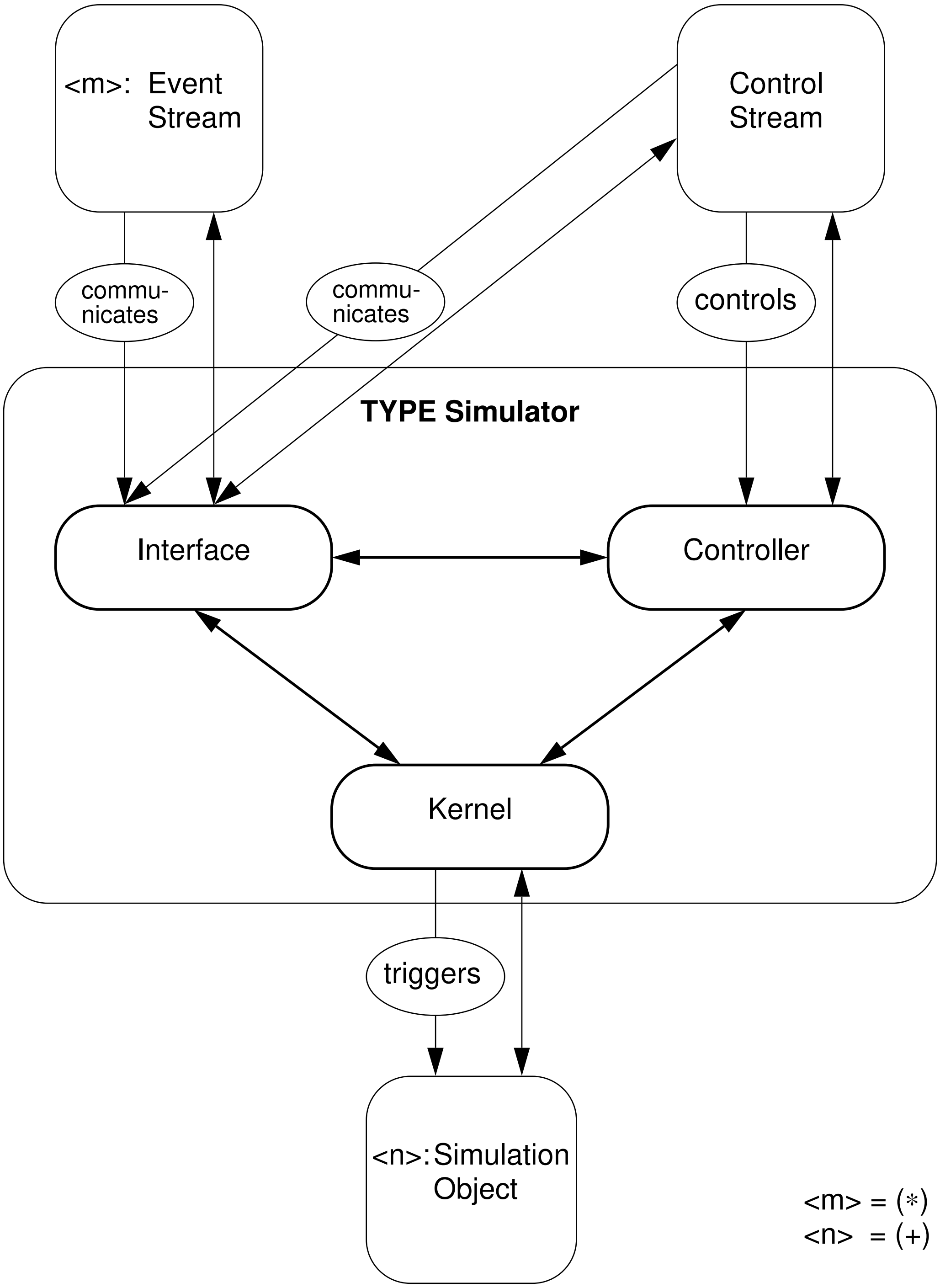
7 Owner Requirements

The non-functional requirements include the following aspects:

- The design has to be object-oriented.
- The accuracy should not be higher than necessary to test control algorithms and provide reasonable responses for the event stream of the building.
- Dumping the system should influence the simulation as less as possible. That is, it should cost as less real time as possible.
- The simulator should run in at least real time.
- Simulation time has to be as close to real time as possible. This is especially important for outputs.

8 System Structure

The system structure reflects that for each kind of entity in the environment, there is a corresponding part inside the simulator. As the control stream is a specialization of the event streams, it interacts both with the interface and the controller. The relationship controls shown in the context diagram was refined into the relationships communicates and controls.



A Appendix: Formalization of Double-Buffering

For simplicity reasons, we only consider a single simulation object o enclosing a single variable with values over some non-empty set V with $\perp \notin V$.

Def. 1: An **object state** is a triple $o = (\alpha, \beta, \tau)$, where α denotes the elder value, β the younger value, and τ the limit timestamp.

Def. 2: An **input sequence** is a sequence $(v_i, t_i)_{i \in \mathbb{N}}$ where
(1) $v_i \in V \cup \{\perp\}$,
(2) $v_i \neq \perp$ or $t_i \neq -\infty$,
(3) $t_i \in \mathsf{Timestamps} \cup \{-\infty\}$,
(4) $(t_i)_{i \in \mathbb{N}}$ monotonously increasing w.r.t. the total ordering extended from $\mathsf{Timestamps}$ onto $\mathsf{Timestamps} \cup \{-\infty\}$.
For a given pair (v_i, t_i), $v_i = \perp$ denotes **read access**, and $v_i \neq \perp$ denotes **write access**.

Def. 2: An **output sequence** is a sequence $(w_i)_{i \in \mathbb{N}}$ over $V \cup \{\perp\}$ where $w_i = \perp$ denotes **empty output** and $w_i \neq \perp$ denotes **value output**.

Def. 3: Given an initial object state $o_0 = (\alpha_0, \beta_0, -\infty)$ and an input sequence $(v_i, t_i)_{i \in \mathbb{N}}$, the **corresponding object state sequence** $o = (\alpha_i, \beta_i, \tau_i)_{i \in \mathbb{N}}$ and **output sequence** w are recursively defined as:

(1) $\underline{v_i = \perp \text{ (read access):}}$

$$w_i = \begin{cases} \alpha_i & \text{if } t_i \leq \tau_i \\ \beta_i & \text{if } t_i > \tau_i \end{cases}$$

$o_{i+1} = o_i$

(2) $\underline{v_i \neq \perp, t_i = -\infty \text{ (initial write access):}}$
$w_i = \perp$
$o_{i+1} = (\alpha_i, v_i, -\infty)$

(3) $\underline{v_i \neq \perp, t_i \neq -\infty \text{ (ordinary write access):}}$
$w_i = \perp$

$$o_{i+1} = \begin{cases} (\alpha) \text{ undefined} & \text{if } t_i < \tau_i \\ (\beta)\ o_i & \text{if } t_i = \tau_i \text{ (input collision)} \\ (\gamma)\ (\beta_i, v_i, t_i) & \text{if } t_i > \tau_i \end{cases}$$

Now, let o_0 denote an initial object state, $(v_i, t_i)_{i \in \mathbb{N}}$ an input sequence, o and w the corresponding object state respectively output sequence.

Prop. 1: **Correctness of write query timestamp**
If $v_i \neq \perp$ (write access at i),
then $t_i \geq \tau_i$ (no write access with earlier timestamp).

Proof: Inductively along i.
i=0: $t_0 \geq \tau_0 = -\infty$ for all $t_0 \in \mathsf{Timestamps} \cup \{-\infty\}$
i→i+1: Distinguish different accesses according to v_i.
(1) $v_i = \perp$ (read access):
$o_{i+1} = o_i$, hence $\tau_{i+1} = \tau_i \leq t_i \leq t_{i+1}$ (monotonicity of $(t_i)_{i \in \mathbb{N}}$)
(2) $t_i = -\infty$ (initial write access):
$o_{i+1} = (\alpha_i, v_i, -\infty)$,
hence $-\infty = \tau_{i+1} \leq t_{i+1}$ for all $t_{i+1} \in \mathsf{Timestamps} \cup \{-\infty\}$
(3) $t_i \neq -\infty$, $v_i \neq \perp$ (ordinary write access):

(α) $t_i < \tau_i$: not possible because of induction hypothesis
(β) $t_i = \tau_i$: then $\tau_{i+1} = \tau_i \leq t_i \leq t_{i+1}$
(γ) $t_i > \tau_i$: then $\tau_{i+1} = t_i \leq t_{i+1}$

Prop. 2: **Effectiveness of write access**

If $v_i \neq \perp$ (write access at i),
$\forall j<i$: $v_j = \perp$ or $t_j < t_i$ (no earlier write access at current simulation time),
$v_{i+1} = \perp$ (read access at i+1), and
$t_{i+1} > t_i$ (read later w.r.t. simulation time than write),
then $w_{i+1} = v_i$ (the previously inscribed value is returned then).

Proof: We have to distinguish initial and ordinary write access.

(2) Initial write access:
$o_{i+1} = (\alpha_i, v_i, -\infty)$, and because $t_{i+1} > t_i \geq -\infty$, we have that $t_{i+1} > -\infty = \tau_{i+1}$, hence $w_{i+1} = \beta_{i+1} = v_i$.

(3) Ordinary write access: Now $t_i > -\infty$.
Starting from $o_i = (\alpha_i, \beta_i, \tau_i)$, we have $t_i \geq \tau_i$ because of Prop. 1.
Therefore, only cases (β) and (γ) remain.
(β) $t_i = \tau_i$: Because of this, we have $\tau_i \neq -\infty$. Inductively, given such a limit timestamp τ_i, then $\exists j<i$: $v_j \neq \perp$ and $t_j = \tau_i$.. This contradicts the second precondition.
(γ) $t_i > \tau_i$: Here, $o_{i+1} = (\beta_i, v_i, t_i)$, and because of $t_{i+1} > t_i = \tau_{i+1}$ we have that $w_{i+1} = \beta_{i+1} = v_i$.

Prop. 3: **Stability of read access**

If $v_i = v_{i+j} = \perp$ and $t_i = t_{i+j} \neq -\infty$, then $w_i = w_{i+j}$.

Proof: Inductively along j.
j=0: obvious
$j \rightarrow j+1$: If there is any $k \in \{1...j-1\}$ such that $v_k = \perp$, then the induction hypothesis already applies. Otherwise, $v_k \neq \perp$ for all such k. The remaining weaker property is shown via induction along k.
k=0: Now, $v_i = v_{i+1} = \perp$. Apparently, $o_{i+1} = o_i$, and because of $t_{i+1} = t_i$ we have that $w_{i+1} = w_i$.
k=1: Obviously, $t_{i+1} = t_i$, and $o_i = o_{i+1}$. Hence, either $o_{i+2} = o_{i+1}$ (write collision with $t_{i+1}=\tau_i$), and instantly $w_{i+2} = w_i$. Or $t_{i+1} = t_i > \tau_i$ and $o_i = (a_i, b_i, \tau_i)$, $w_i = b_i$, $o_{i+1} = o_i$, $o_{i+2} = (b_i, v_{i+1}, t_{i+1})$, and $w_{i+2} = b_i = w_i$.
$k+1 \rightarrow k+2$: Here, $v_{i+1}, ..., v_{i+k+2} \neq \perp$. Because of $t_{i+k+2} = t_{i+k+1}$, we have that $o_{i+k+2} = o_{i+k+1}$ (write collision). Hence, the induction hypothesis applies.

Def. 4: A **dump index** is an index $n \in \mathbb{N}$ such that $\forall i \geq n$: $t_i > \tau_n$.
Due to monotonicity of $(t_i)_{i \in \mathbb{N}}$, $t_n > \tau_n$ is sufficient.

Def. 5: Given a dump index n, a **restart** consists of the object state sequence o' and the output sequence w' corresponding to the initial object state $o'_0 = (\nu, \beta_n, -\infty)$, where ν is an arbitrary element of V, and the input sequence $(v'_i, t'_i)_i = (v_{i+n}, t_{i+n})_i$.

Prop. 4: **Identity of restart**

Let n denote a dump, o' and w' a restart, $r := 1+\min\{j \in \mathbb{N}: v'_j \neq \perp\}$ the index of the first state with a newly set value.

(1) $o'_{r+i} = o_{n+r+i}$
(2) $w'_i = w_{n+i}$

Proof: For (1) we only have to show that $o'_r = o_{n+r}$. Clearly, $o_{n+r-1} = ... = o_n = (\alpha_n, \beta_n, \tau_n)$. Now, o_{n+r-1} is followed by o_{n+r}, which is triggered by (v_{n+r-1}, t_{n+r-1}) with $t_{n+r-1} \geq t_n > \tau_n = \tau_{n+r-1}$, so we have ordinary write access without input collision and $o_{n+r} = (\beta_n, v_{n+r-1}, t_{n+r-1})$. Correspondingly, $o'_{r-1} = ... = o'_0 = (\nu, \beta_n, -\infty)$, and o'_{r-1} is succeeded by o'_r via $(v'_{r-1-1}, t'_{r-1-1}) = (v_{n+r-1}, t_{n+r-1})$. Since $t'_{r-1} \geq t_n > \tau_n \geq \tau'_{r-1} = -\infty$, we have that $o'_r = (\beta_n, v'_{r-1-1}, t'_{r-1-1}) = o'_{n+r}$.
Applying (1), property (1) is obvious for i $\geq$r. For i<r, we have $t'_i > -\infty$ because of $t'_o = t_n > \tau_n \geq -\infty$, hence $w'_i = \beta_n$. Correspondingly, $w_{n+i} = w_n = \beta_n$.