Project:	**D1.Project1**
Name:	**Building Simulator Simple1**
Document:	**Requirement Specification**
Responsible:	**Gruppe 1**
File-Pathname	**irz1:~sfb501/requirement_specification/ reqspec.4**

Type of change	Date	Version	Person
Start	2/1/95	1.0	Gruppe 1
New	2/14/95	2.0	Gruppe 1
Addition	02/17/95	3.0	Gruppe 1
Addition	02/28/95	3.1	Gruppe 1
Addition	03/1/95	3.2	Gruppe 1
Merge	03/08/95	4.0	Gruppe 1/2
Addition	03/10/95	4.1	Gruppe 1/2
Addition	03/14/95	4.2	Gruppe 1/2
Change 1.5-1.7	03/20/95	4.3	Gruppe 1/2
New 1.3-1.4	04/20/95	4.4	Gruppe 1/2
Change 1.5-1.7	04/20/95	4.5	Gruppe 1/2
Change / Addition	05/11/95	4.6	Gruppe 1/2

Requirement Specification

I. Functional Requirements

The numbering scheme follows the outline of the SOON and SDL methodology as described in “Engineering Real Time Systems” by Rolv Braek and Oystein Haugen. The complete section “1.0 Problem Statement” is identical to the section 2 “Functional Requirements” by AG Rombach.

1.0 Problem Statement

1.1 Problem Statement:

Certain persons, such as architects, customers, inhabitants, and engineers of control systems are interested in dynamic properties of a given building. Therefore it must be possible to control and monitor the state of the building. A part of the state which can be monitored, is e.g. the air temperature in a single room. The persons may be interested in these properties over a certain period of time, e.g. a day or a year. Using a simulation of the building the time to get the same information by controlling and monitoring the simulated system may be reduced dramatically. Furthermore it is possible to analyse buildings which are still under construction. The simulated system should behave in the same way as the real system with an appropriate accuracy.

The simulator should be able to run in real time or multiples thereof. As in the real building it should be possible to influence part of the simulated building, such as e.g. windows, radiator valves, and the location of inhabitants, during an ongoing simulation run. In the same way it should be possible to monitor the simulated building. Output can be made on request, periodically, and on change of a monitored value.

Neither the input nor the output are restricted to a single stream. Conflicts between incoming streams shall be resolved by priorities between the input streams. All input and output streams have the same format, so that each output could be used as an input of another simulation run.

To control a simulation run it should be possible to initialize the state of the building and to start, stop or continue the run. It should be possible to store the state of the simulation system and continue from this state at another time. Before starting and after stopping a simulation run it is possible to adjust several parameters such as:

- parameters of the building (e.g. standard heat output of a radiator, but not the geometry of the building).
- the timing factor and the slack
- the input configuration (the priorities)
- the output configuration (which output should be sent to which stream at which times)

The simulator should be tolerant with respect to wrong user inputs.

Besides these general properties a certain realization of the simulator is demanded. The building is partitioned into individual objects, such as rooms and radiators. These objects are given by the customer. The objects encapsulate the physical properties of the building and perform the actual simulation. To do this they are invoked at appropriate times (as demanded by the stability of the underlying numerical method) to compute a new state of the corresponding part of the simulated building. Thereby they may ask other objects about their state. The underlying realization of the simulator should be an event driven one.

As it is planned to use similar simulators to check properties of other buildings too, the design should be done with reuse in mind to allow at some time in the future to generate the simulator from a description of a house.

1.2 Dictionary

command event. an event of the form *(timestamp, command [, ,parameter]*)*. This means, that at time *timestamp*, the simulator performs *command* with *parameters*.

dictionary. a bijective function from a set of names of objects (external names) onto a set of object identifiers (internal names).

domain object. specialisation of simulation object; encapsulates the physical properties of a part of the building.

dump. a representation of the state of the simulator, including the values of all state variables of all domain objects, the event collection and the internal state of the interface. A dump can be retrieved to get identical initial states for several simulation runs.

event, scheduled, (kernel event). each event is a 4- or 5-tuple consisting of a time stamp (representing the simulation time at which the event shall happen), a priority (to order events with equal time stamp), an object reference (representing the simulation object affected by the event), the concrete type of the action that shall be performed, and the frequency in ticks of simulation time in case of periodic events; the events are collected in an event list, which is partially ordered by the timestamp of the events and the priorities.

event, (interface event). an interfaceevent is a 3-tupel consisting of a time stamp, an object name, and a event type (e.g. set, get). Input events comprise *command events* and *value events*.

(event) stream. communication relation between an origin and a destination consisting of a sequence of events.

input (to the simulator). event of an input stream; used to control the simulation run (e.g. initialize, start, stop) or to change some values of the objects. There are two kinds of such input events: value and command events. (??? only command events ???)

input configuration. association of priorities to input streams.

input stream. event stream with the simulator as destination; origin could be a file, a person (e.g. user who enters the inputs on-line with a keyboard or writes them in an input file) or another system (e.g. control system). Each input stream has an associated priority.

interface. part of the simulator responsible for managing the communication with the outside world (i.e. event streams). Incoming events are distributed to the kernel or the simulation objects. Output events are forwarded to a particular output stream.

invoke. if the kernel schedules an event then it must invoke the object named in the event; the object then has to perform the associated action

multiples of realtime. It should be possible to compress the realtime, such that e.g. a day can be simulated in 5 minutes.

output (of the simulator). event of an output stream; e.g. used to monitor the state of a simulation object.

output configuration. determines which output is made periodically or on change of a state variable. Furthermore it describes the distribution of these outputs and of warnings, error messages, outputs on demand, etc. onto the output streams.

output stream. event stream with the simulator as origin; destination could be another system (e.g. control system) or a person (via a terminal) or a file.

parameters. are settings of the simulator and of the simulated system which are considered invariant during a simulation run and could be initialized at the beginning of the run. (To be more precise, to change these parameters, a simulation run has to be stopped and restarted after the input of the changes). Examples are the timing factor and the standard heat output of each radiator in the building.

priority of inpuut streams. every input stream has an associated priority which is used to resolve conflicts between input streams. The priorities imply a partial order of the input streams.

priority of scheduled events. determines the ordering of scheduled events with equal time stamp; belongs to a priority class.

priority class. are defined to partition events with equal time stamps. Priority classes are defined for scheduled events only. The following priority classes are known to the simulator: Set-Events (2 classes), Get-Events (1 class), Simulation-Events (1-class), Command-Events (2 classes). Each class consists of an upper and lower bound with a minimal and class-specific number of elements. Priority classes as well as the priorities within a class are ordered.

realtime. the time passing by during the execution of a simulator on one or more computer systems.

simulated system. a model of the real system (building) which behaves in all relevant aspects the same way as the building (with an appropriate accuracy); it consists of several communicating (simulation) objects, each representing the behaviour of an individual part of the real system

simulation object. simulates the physical behaviour of a certain part of the real system.

simulation run. the simulation of the building within an interval of realtime. It typically consists of an initialization phase and the actual run with inputs to and outputs from the simulated system.

simulation time. internal time of the simulator. It is discrete and it can be stopped. It should correspond as close as possible to the (compressed) real time. If all inputs are known in advance, it should also be possible to advance the simulation time as far as possible.

simulator. responsible for simulating the building and managing the communication with the users; consists of several simulation objects, one kernel, and one interface; it should realize an event-driven simulation (user requirement)

(simulator) kernel. part of the simulator responsible for scheduling the events at the appropriate times.

slack. defines the allowed deviation from real-time if the simulation is performed with a given real-time binding.

state of the building. values of all relevant parts of the real system (building) like room temperature, opening state of the valve of a radiator, etc.; part of the state can be controlled (i.e. the values can be set), another part can be monitored (i.e. the values can be measured); the two parts could have a non empty intersection and their union does not have to represent the whole state. That is, there is some kind of internal state which is not visible.

state variable. an object encapsulating the value of a variable in a domain object. There are several types: input, output, parameters, and internal. These objects also encapsulate the double buffering of the values. The value of the state variables can be set and retrieved

directly without informing the corresponding domain object.

timestamp. each event has a timestamp indicating at which (simulation) time it shall take place.

timing factor. describes by which factor the simulation timed (compressed realtime) is related to the realtime.

value event. an event of the form *(objectname, variablename, value)*. This means that the variable *variablename* of object *objectname* shall be set to *value*.

1.3 Concept Model

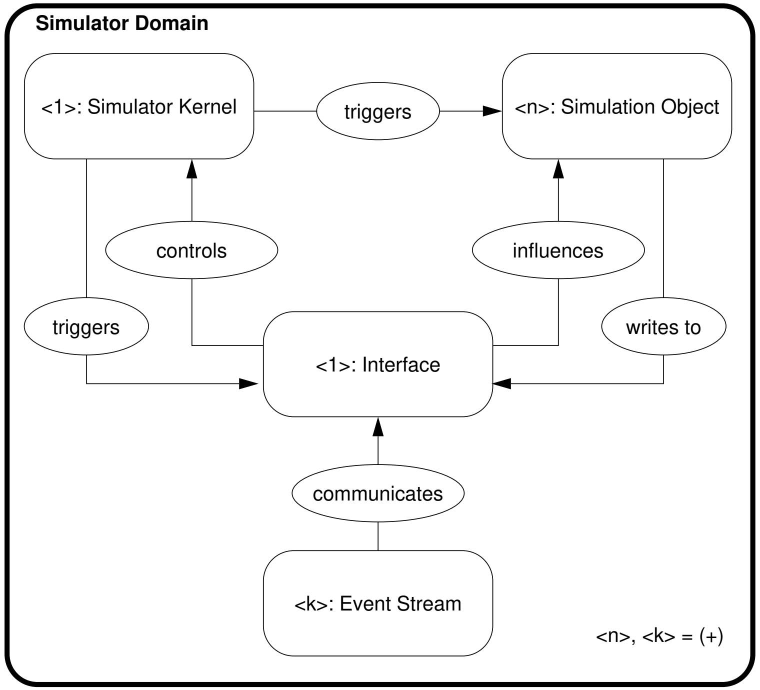


Fig. 1: The simulator domain

The simulation is performed by a collection of *simulation objects* which are specific to the application-domain (in this case simulating the physical behaviour of a given building). The simulation itself is *event-driven* (requirement). For this purpose, the *simulator kernel* manages at each time t in real time a collection of events to be scheduled in the next future:

$ecoll_t := \{event_1, ..., event_{nt}\}$

Each kernel $event_i$ = (st, prio, obj, etype) comprises a time stamp **st** in simulation time, a priority **prio** which defines an execution order for events with equal simulation time stamp, an object identifier **obj** identifying the object the event is scheduled for, an event type **etype** specifying the concrete action the object performs when the event is scheduled, and an **frequency** in case of periodic events. A tick in simulation time has a given and fixed resolution (e.g. ns). This resolution has to be known to the simulation objects in order to schedule events properly. There is—at any real time t—a partial ordering among the events in $\mathbf{ecoll_t}$ based on the time stamps in simulation time and the priorities. The kernel guarantees to schedule events with monotone increasing simulation time stamps.

By every scheduling action, the kernel selects a set $e_{trigset}$ of events from $ecoll_t$ with smallest time stamp $st_{trigset}$ and smallest priority $prio_{trigset}$. All events $e_{trig} \in e_{trigset}$ have the form:

$(st_{trigset}, prio_{trigset}, obj_{trig}, etype_{trig})$ or
$(st_{trigset}, prio_{trigset}, obj_{trig}, etype_{trig}, frequency_{trig})$.

The kernel triggers a simulation step in each object as defined by the events in $e_{trigset}$. If two or more events are elements of $e_{trigset}$, the execution order of these events is non-deterministic and the execution may even happen in parallel. As a consequence of each simulation step, the triggered simulation objects $\mathbf{obj_{trig}}$ ($\forall$ $e_{trig} \in e_{trigset}$) themselfs or any other simulation object (due to interdependencies between the simulation objects themselves - not visible to the simulator kernel) may schedule new events for any simulation object. For any new event

$(st_{new}, prio_{new}, obj_{new}, etype_{new})$

it must hold that $\mathbf{st_{new}} > \mathbf{st_{trigset}}$ or in the special case of $\mathbf{st_{new}} = \mathbf{st_{trigset}}$, the priority class of $\mathbf{prio_{new}}$ must be greater than the priority class of $\mathbf{prio_{trigset}}$.

By performing the simulation action, the kernel removes the just scheduled events $e_{trigset}$ from $\mathbf{ecoll_t}$, it subsequently triggers the simulation steps defined by $e_{trigset}$, and adds every new event scheduled by some simulation object to the actual collection of events (resulting in a new event collection $\mathbf{ecoll_{tp}}$):

$ecoll_{act} := ecoll_t - e_{trigset}$
triggering of simulation steps for all events $e_{trig} \in e_{trigset}$
$ecoll_{act} := ecoll_{act} + \{\text{new events}\}$

In case of periodic events

$e_{periodic} = (st_{trigset}, prio_{trigset}, obj_{periodic}, etype_{periodic}, frequency_{periodic}) \in e_{trigset}$,

new events of the form

$(st_{trigset}+frequency_{periodic}, prio_{trigset}, obj_{periodic}, etype_{periodic}, frequency_{periodic})$

are added:

$ecoll_{tp} := ecoll_{act} + \{\text{periodic events}\}$.

For each scheduling action, the sets {new events} and {periodic events} may be empty. The kernel proceeds to the next event set stored in $\mathbf{ecoll_{tp}}$ according to the partial ordering of the simulation time stamps when it is guaranteed that no further event is created by any simulation object (termination of the actual simulation step triggered by $\mathbf{e_{trig}}$). This termination can only be detected by the simulation objects themselves. If the set $e_{trigset}$ contains more than one events, the kernel will wait for the completion of all these execution steps—possibly performed in parallel—at one place (this requires, that the kernel is informed about the termination of every single simulation step).

Priority classes determine an order for the execution of events with the same simulation time stamp. According to the semantics of commands such as setting and getting values of state variables, or stopping the simulator, for a consistent behaviour of the simulator several priority classes (with low and high limits) are defined. Priority classes and the priorities available within a class are ordered. Of interest for the simulation objects is only the class “simulation priority class” with at least “min_class_elements” values available for the scheduling of simulation events. The lower and upper bound of this class must be available at implementation time of simulation objects. The main purpose for the visibility of event priorities is to enable the efficient transformation of the continuous or discrete simulation paradigm to the event-driven paradigm.

The simulator may—when requested—try to maintain a given real-time binding within predefined timing factor (acc) and slack as long as the execution times of simulation actions in the application-defined simulation objects allow for this. Given that the initial simulation time stamp st_0 happens at real-time rt_0, every subsequent event with simulation time stamp st will be executed at real-time rt:

$$(st - st_0) * acc + rt_0 \leq rt \leq (st - st_0) * acc + rt_0 + slack$$

This relation is tested before the start of each simulation action. If the upper bound of a real-time binding is exceeded, the simulator issues an error message and stops the execution of any further events.

1.3.1 The simulator interface

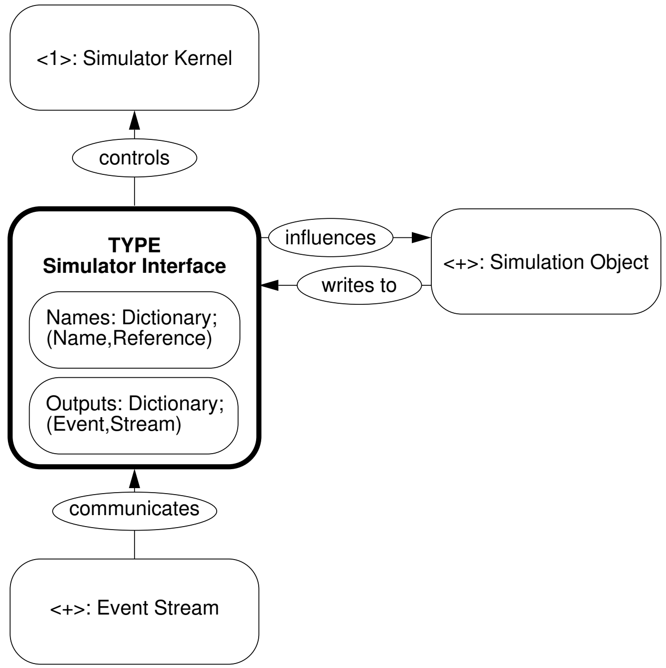


Fig. 2: The interface type

The primary purpose of the simulator interface—from a logical point of view—is to distribute the events of the various event streams either to the simulator kernel in order to influence the execution of a simulation, to simulation objects in order to change some of the object’s values, or to forward output events of simulation objects to a particular event stream. It is implementation dependent whether these tasks are performed just with the objects involved or with the kernel’s assistance. Each event stream consists of a sequence of interface events and an associated—not necessarily unique—priority:

$$estream_i := \{ Priority, \{ event_1, ..., event_{ni} \} \}$$

An interface event may be a *value event* e_v = (st_v, $object\_name_v$, sv_name_v, $value_v$) or a *command event* e_c = (st_c, $command_c$ [, $paramters_c$]). A value event as input defines a new setting $\mathbf{value_v}$ for a state variable $\mathbf{sv\_name_v}$ of object $\mathbf{object\_name_v}$. The change is performed at simulation time $\mathbf{st_v}$. A value event as output informs about the $\mathbf{value_v}$ of a state variable $\mathbf{sv\_name_v}$ of object $\mathbf{object\_name_v}$ at simulation time $\mathbf{st_v}$. Output values might be requested asynchronously by issuing a poll command or synchronously (periodically or in case of value changes). Command events as inputs control the behaviour and the execution of the simulator. An input command takes place at the specified simulation time $\mathbf{st_c}$. A command event as output tells that $\mathbf{command_c}$ has been executed by the simulator at simulation time $\mathbf{st_c}$ (output command events may be used for replay purposes). The priority of event streams defines an order for colliding events. A input collision occurs if at least two events have to be scheduled to set one state variable in one specific object at the same simulation time with different values. In case of such a collision, the event with the highest priority is taken and an error message is produced, the other colliding events are discarded. Input events may have no simulation time stamp attached to them. In this case, the simulator tries to perform these events as early as possible by assigning the next possible time stamp.

The interface contains a dictionary for the translation of object names to object references in order to deliver input value events to the specified instance of a simulation object. To fill this dictionary, the simulation objects are required to register their names and the names of their state variables (all state-sensitive variables, thus variables visible to the environment of the simulator: input variables, output variables, and parameters as well as internal state variables) at start-up time of the simulator. The dictionary provides a hierarchical name space such as

/object_name$_1$/object_name$_2$/.../object_name$_k$/state_variable_name.

This enables simulation objects to establish a certain hierarchical name space in the registration phase. As already mentioned, part of this dictionary are also "internal state variables" (state variables of objects not visible to the environment of the simulator) to enable double buffering and dumping of the simulator's complete state. The names $\mathbf{object\_name_v}$ in value events always refer to visible state variables and are required to be absolute starting from the top-level hierarchy.

Event streams are input streams and output streams. Input and output streams might be attached to files. They may also represent connections to user interfaces, real control systems, or even other simulators (e.g. a simulator for the control system). In case of input files, it can be assumed that all the events (event$_1$, ..., event$_{ni}$) are available a priori, but that they are ordered by simulation time stamps. In case of dynamic streams, these streams are strongly related to the actual simulation time of the simulator at a given real time t. This requires that such a stream never issues an event with a simulation time stamp earlier than the actual simulation time of the simulator. It also implies an increasing ordering of the events by simulation time stamps. As already mentioned, there may be events with no time stamp assigned to them, thus giving the simulator the freedom to schedule these events as early as possible.

Outputs of simulation objects or of the simulator kernel (output command events) are forwarded to event streams according to the dispatch information in the dictionary Outputs. The content of this dictionary together with the connection information about input streams has to be defined in a configuration file to be read by the simulator at start-up time. The simulator also enables the change of this configuration dynamically at run-time by issuing special command events over some input stream.

1.3.2 The interface to the simulation objects

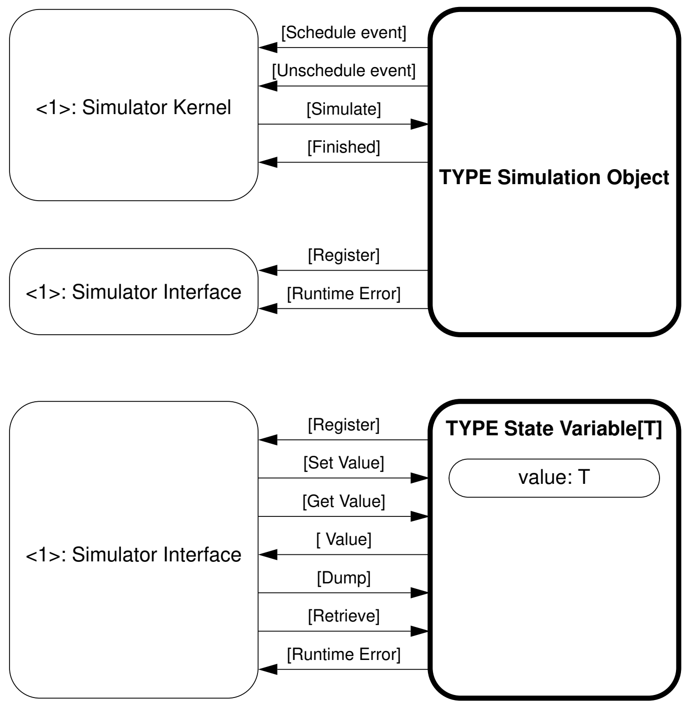


Fig. 3: The object types required for the development of domain-specific simulation objects

The "interface" to the simulation objects is on the level of types. It consists of a type "Simulation object" that defines the minimal interface each simulation object has to provide in order to be managed correctly by the simulation kernel and the simulation interface. The second type "State Variable" is required to enable the interface to deliver input value events to the specified state variable instance, to retrieve the value of state variables and to forward them to a specified output event stream, and to coordinate the dump of the complete simulator state (requested by a command event) as well as double buffering. Depending on the implementation language chosen, this type may be parametrized in order to provide all the functionality for any application-specific type.

Every simulation object provides a function *Register* to store the object's name in the dictionary Names of the interface. A simulation object may also add new events (*Schedule*) to the collection $\mathbf{ecoll_t}$ stored in the kernel to be scheduled at a given time in the future of simulation time. Simulation objects are allowed to schedule events for any other simulation object and to remove them by calling *Unschedule* (Since the events triggering a simulation step are removed from ecoll_t before the simulation step actually happens, these events cannot be unscheduled any more). Normally, every simulation object is passive (except in the registration and initialization phase). A simulation object becomes active if the kernel issues a call to *Simulate* for the certain object or if another simulation object is called transitively by an object that received such a Simulate message. This call triggers a simulation step in the specified object at the actual simulation time of the overall simulator. The simulator kernel guarantees that subsequent calls to Simulate for this object have monotone increasing simulation time stamps. Only active simulation objects may schedule new events. Due to application-specific dependencies between the simulation objects the scheduling of new events might be performed by other simulation objects than the object which received the call to Simulate primarily. This propagation of activity between different simulation objects is not controlled by the simulation kernel. It is only required that exactly one simulation object sends a *Finished* to the kernel, signalling that every simulation object has reached its passive state again (thus no further scheduling of new events happens). Logically, the kernel proceeds with the scheduling of the next event iff it receives the Finished message. In reality the kernel may execute several simulation steps in parallel, if they have the same simulation time stamp and priority. But the requirement remains, that the kernel is informed about the termination of each of these simulation steps by a corresponding finish message.

The interface between state variables and the simulator interface consists of a *Register* call to register the name in the dictionary Names. The *Set* call defines a new value for the state variable. This value change is performed without the notification of the simulation object containing the state variable. If the semantics for the simulation objects requires such a notification, it is up to the developer of application-specific simulation objects to specialize the type "State Variable" and to define a derived call *Set* which offers some notification mechanism. Setting a state variable by issuing a set command has no side-effects under any circumstances which change the state of any simulation objects to active (e.g. scheduling new events because of a set operation), otherwise a Finished message to be issued by some simulation object must be introduced in this case too. By issuing a *Get*, the interface requests the actual value of the state variable. This value is returned via *Value*. The last two calls are required to dump the actual value of state variable on file or to retrieve a dumped state back from file.

1.4 Specialization Hierarchy

The only relevant specialization hierarchy defines the integration of application-specific simulation objects into the framework of the simulator itself. The framework is defined by the two types "Simulation Object" and "State Variable" as described in section 1.3.2.

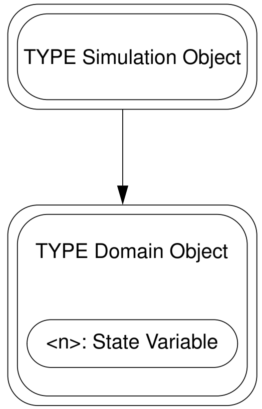


Fig. 4: Specialization of an domain-specific simulation object

The domain objects are specializations of the type "Simulation Object". Typically, they define new semantics for the *Simulate* call. Every variable of a domain object that is part of the essential state of the object must be instantiated as a state variable or a derivation of it.

1.5 Context

The context is defined in figure 5.

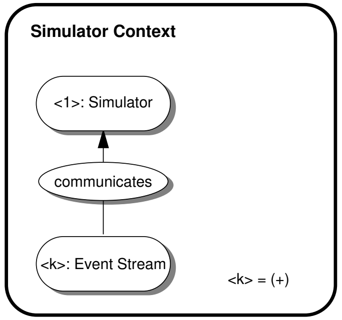


Fig. 5: Simulator Context Diagram

The information which is exchanged between the different streams and the simulator has to be of the following format.

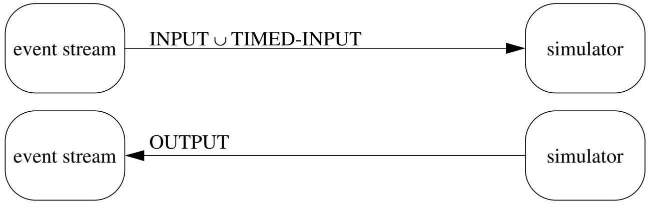
INPUTS:

1. TIMED-INPUT = TIME x INPUT
2. TIME = Set of all possible timestamps
3. INPUT = VALUE ∪ COMMAND ∪ SIM_INIT ∪ DUMP ∪ DUMP_OBJ (all possible inputs)
4. VALUE = OBJ_NAME x VAR_NAME x VALUE
5. OBJ_NAME = Set of all possible names for objects
6. VAR_NAME = Set of all possible names of state variables
7. VALUE = Set of all possible values for any state variable
8. COMMAND = INIT ∪ RECONFIGURE ∪ START ∪ RETRIEVE ∪ POLL-VALUE ∪ POLL-TIME ∪ POLL-DUMP
9. INIT = {“init”, “endinit”}
10. RECONFIGURE = {“reconfigure”} x SIM-INIT
11. START = {“start”, “stop”}
12. RETRIEVE = {“retrieve”} x IN-STREAM x FLAGS
13. FLAGS = {“all”, “objects”}
14. POLL-VALUE = {“poll”} x {“value”} x OBJ_NAME x VAR_NAME x OUT-STREAM
15. POLL-TIME = {“poll”} x {“time”} x OUT-STREAM
16. POLL-DUMP = {“poll”} x {“dump”} x OUT-STREAM
17. SIM-INIT = {time-start, acc-factor, slack, input-def, period, state-change}, time-start ∈ TIME (starting time of the simulation), slack ∈ DELAY (time deviation), acc-factor ∈ ACC-FACTOR (acceleration factor comparing real time to simulation time), input-def ∈ INPUT-DEF, period ∈ DELAY x OBJ_NAME x VAR_NAME x OUT-STREAM (definition of output of periodical changes), state-change ∈ OBJ_NAME x VAR_NAME x OUT_STREAM (definition of output of state changes)
18. ACC-FACTOR = $\mathbb{R}^+$
19. DELAY = Set of all difference Times (delay ∈ DELAY iff delay = time1 - time2; time1, time2 ∈ TIME and delay > 0)
20. INPUT-DEF = IN-STREAM x PRIORITY
21. IN-STREAM= Set of all allowed names for Input-Streams
22. PIRORITY = $\mathbb{N}$
23. OUT-STREAM = Set of all allowed names for Output-Streams
24. DUMP = Image of the state of the simulator
25. DUMP_OBJ = Image of the objects state

OUTPUTS:

1. OUTPUT = TIMED-VALUE ∪ TIME ∪ ERROR-MESSAGES ∪ DUMP ∪ ACK
2. TIMED-VALUE = TIME x VALUE
3. TIME: same as INPUT
4. ERROR-MESSAGES: Set of all error messages
5. DUMP: same as INPUT
6. ACK = {“ack”} (message to tell that the required action was successful)

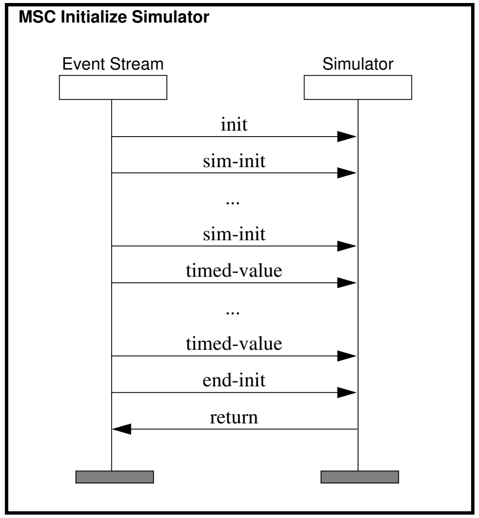
Channels between event stream and simulator:



1.6 Message Sequence Charts

In the following MSC's there is only one event stream. It should always be kept in mind that each input can be send by a separate stream and each output can be sent to a different stream. Normally results, acknowledgements and error messages will be sent to different streams.All input values and commands can have a time stamp or not.

Initialization of the simulator

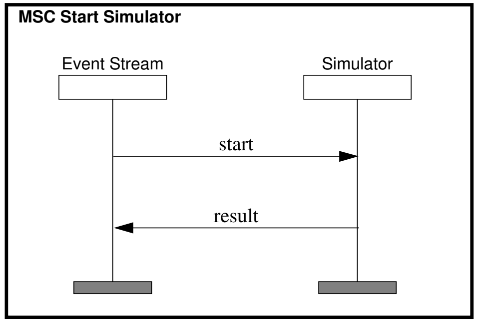


init = “init”, endinit = “endinit”, sim-init $\in$ SIM-INIT, value $\in$ VALUE, return $\in$ ACK $\cup$ ERROR-MESSAGES

After issuing the command “init” zero ore more initialization values for the simulator can be entered. Further, zero ore more values can be entered in order to overwrite the values of the state variables of the objects. The order in which those initialization values are entered is arbitrary. sim-inits and values can be entered in any order. After entering all initialization data the command “endinit” is issued. The simulator either gives an acknowledge that the action was successful or an error-message is sent.

The initialization process should be repeated till a positive acknowledge of the simulator is returned as only then the simulation can be started.

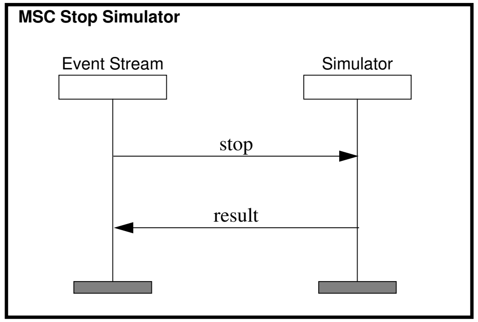
Start of the simulator



start = “start”, result ∈ ACK ∪ ERROR-MESSAGES

The simulator is started by issuing the command “start”. A positive acknowledge or an error message is returned by the simulator.

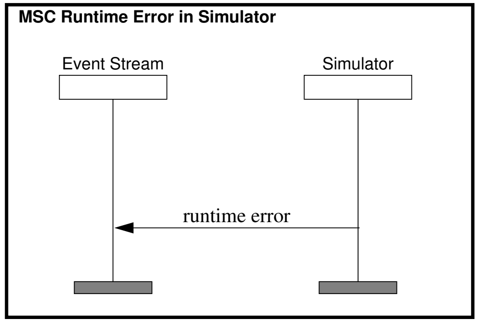
Stop of the simulator



stop =“stop”, result ∈ ACK ∪ ERROR-MESSAGES

The simulator is stopped by issuing the command “stop”. A positive acknowledge or an error-message is returned by the simulator.

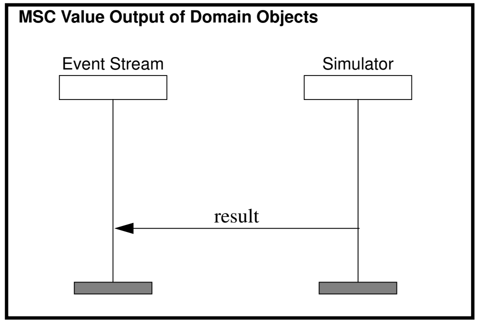
Runtime error return



runtime error ∈ ERROR-MESSAGES

A runtime error of the simulator can occur.

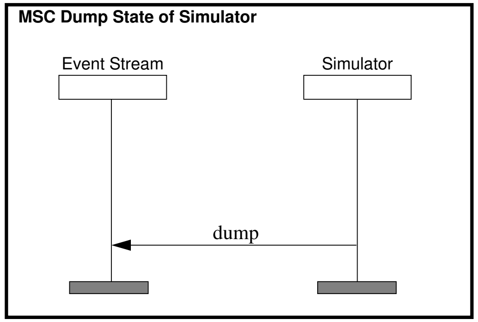
Result return



result ∈ TIMED-VALUE

A result can be returned regularly or when the state variable of an object changes.

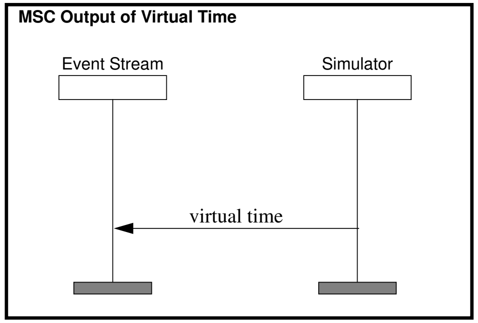
Dump return



dump ∈ DUMP

A dump is a complete image of the current state of the simulator consisting of the simulators context state and the simulators objects state.

Virtual time return

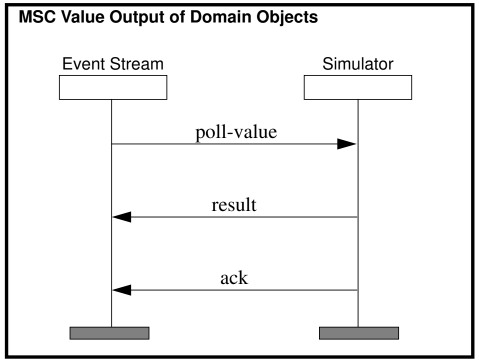


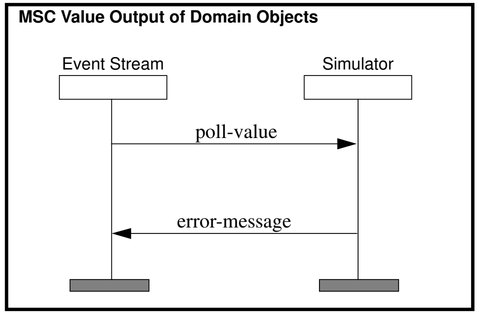
virtual time ∈ TIME

The virtual time can be returned regularly.

A second series of Message Sequence charts differs from the previous ones (result return, dump return, virtual time return) only in that first the command “poll” is issued by the event stream. This command has a parameter which corresponds to the requested item (result, dump, virtual time).

Poll Result

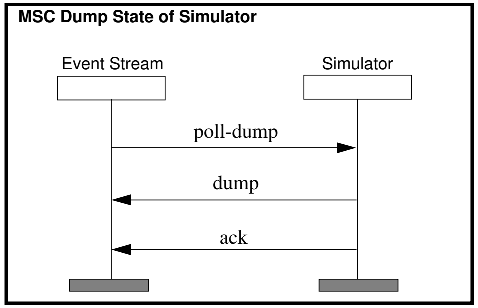


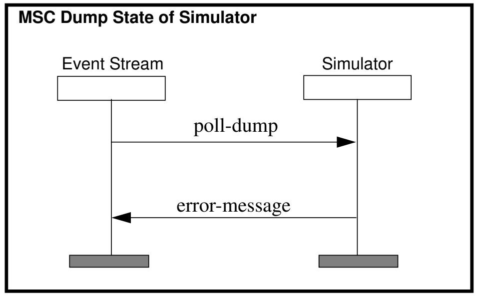


poll-value ∈ POLL-VALUE ∪ TIME x POLL-VALUE, result ∈ TIMED-VALUE, ack ∈ ACK, error-message ∈ ERROR-MESSAGES

A value of an objects state variable at a specific time can be polled. The polled value is written on a stream. A positive acknowledge is returned. If the transaction fails, no value is written and an error message is issued

Poll dump

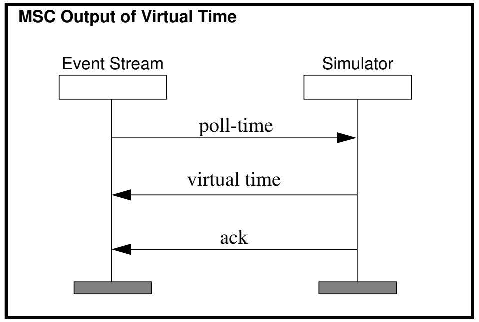


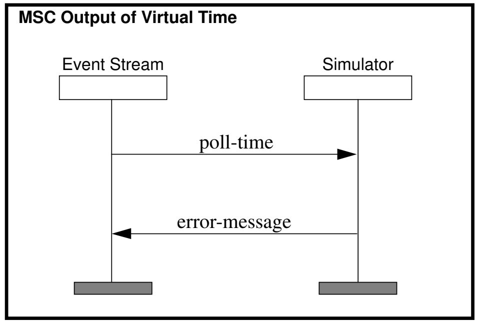


poll-dump $\in$ POLL-DUMP $\cup$ TIME x POLL-DUMP, dump $\in$ DUMP, ack $\in$ ACK, error-message $\in$ ERROR-MESSAGES

A dump can be initiated at a specific time by the command poll-dump. If the dump is successful, a positive acknowledge is given else an error-message is sent. The dump consists of the simulators state.

Poll Virtual Time

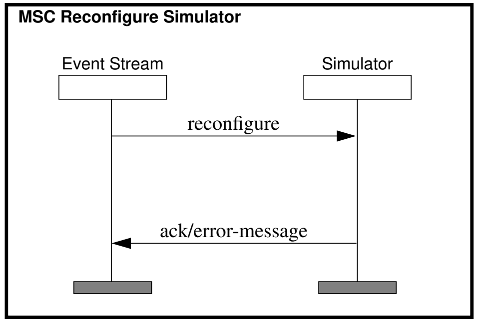




poll-time $\in$ POLL-TIME $\cup$ TIME x POLL-TIME, virtual time $\in$ TIME, ack $\in$ ACK, error-message $\in$ ERROR-MESSAGES

The virtual time can be requested and is then returned. A positive acknowledge is sent when the time can be returned else an error-message is sent.

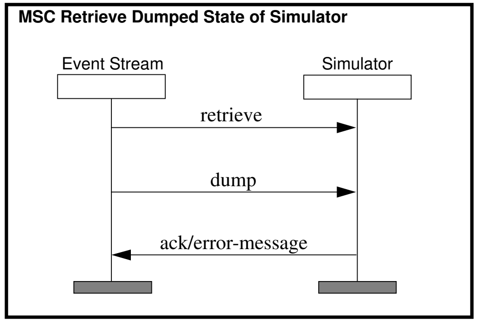
Reconfigure the Simulator



reconfigure ∈ RECONFIGURE ∪ TIME x RECONFIGURE, ack ∈ ACK, error-message ∈ ERROR-MESSAGES

The command reconfigure changes an init value. An acknowledge is sent if the change was accepted else an error-message is sent.

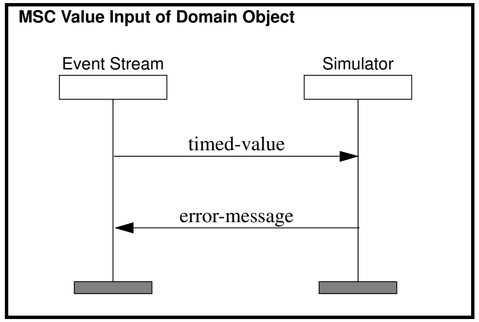
Retrieve dump



retrieve ∈ RETRIEVE ∪ TIME x RETRIEVE, dump ∈ DUMP or dump ∈ DUMP_OBJ, ack ∈ ACK, error-message ∈ ERROR-MESSAGES

The retrieve command tells the simulator to load a dump. The simulator-state or only the state of the objects can be retrieved. An acknowledge is sent if the dump was successfully retrieved else an error-message is sent.

Change state variable of an object



timed-value ∈ TIMED-VALUE, error-message ∈ ACK ∪ ERROR-MESSAGES.

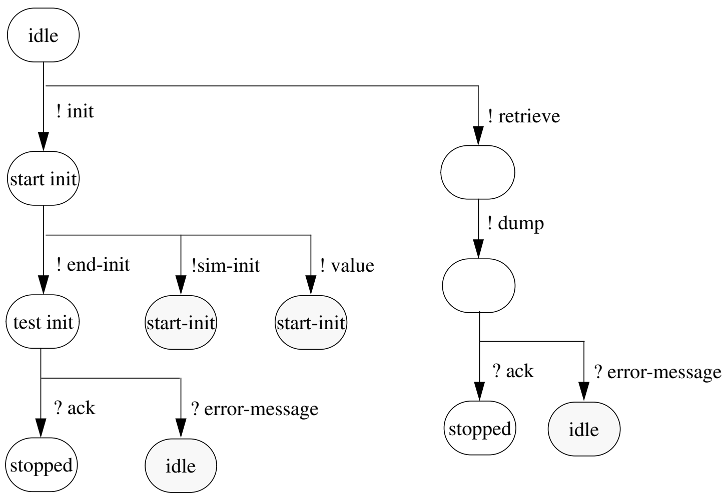
A timed value can be consumed by the simulator. If an error occurs an error-message is sent out.

1.7 Role Behaviour between event stream and Simulator

The following figures describe the role behaviour of the simulator as seen by the event stream. It shows how the MSCs which are described in the previous section can be combined to get sample event stream-simulator interaction.

All names of states are only because they are referred to. They don't imply any meaning!

Transitions from state idle

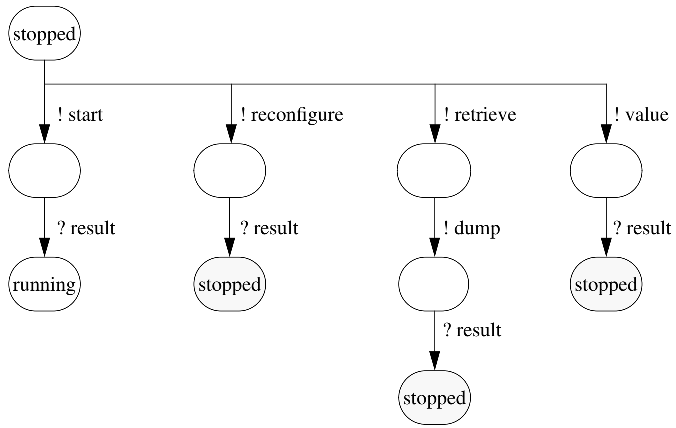


init = “init”, end-init = “endinit”, sim-init $\in$ SIM-INIT, value $\in$ VALUE,

ack $\in$ ACK, error-message $\in$ ERROR-MESSAGES,

retrieve $\in$ RETRIEVE, dump $\in$ DUMP.

Transitions from state stopped

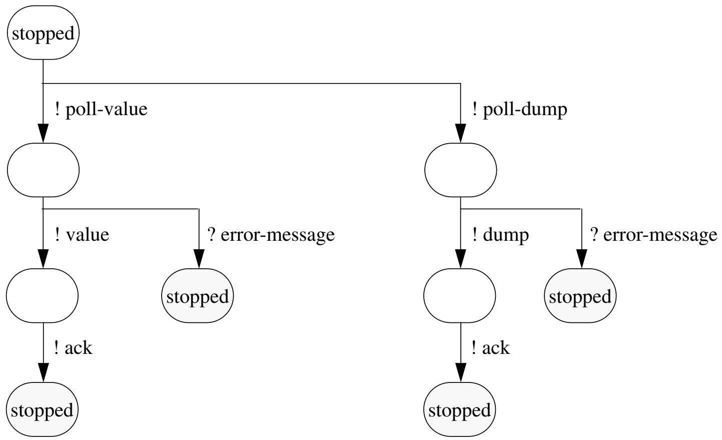


start = “start”, result $\in$ ACK $\cup$ ERROR-MESSAGES,

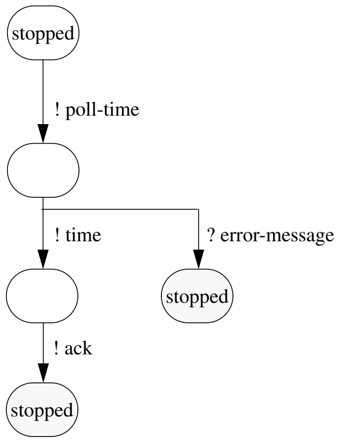
reconfigure = RECONFIGURE $\cup$ TIME x RECONFIGURE,

retrieve = RETRIEVE $\cup$ TIME x RETRIEVE, dump $\in$ DUMP $\cup$ DUMP_OBJ,

value $\in$ VALUE $\cup$ TIMED-VALUE.

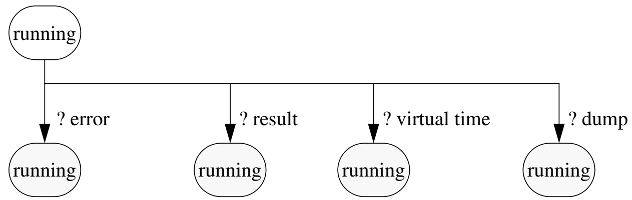


poll-value ∈ POLL-VALUE ∪ TIME x POLL-VALUE, result ∈ TIMED-VALUE,
poll-dump ∈ POLL-DUMP ∪ TIME x POLL-DUMP, dump ∈ DUMP,
ack ∈ ACK, error-message ∈ ERROR-MESSAGES.

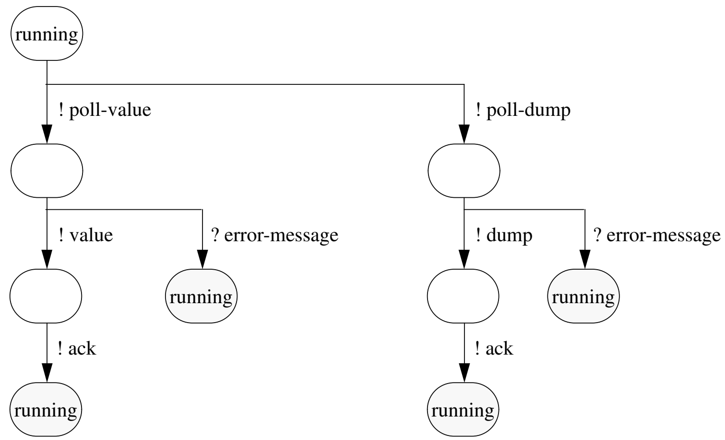


poll-time ∈ POLL-TIME ∪ TIME x POLL-TIME, time∈ TIME,
ack ∈ ACK, error-message ∈ ERROR-MESSAGES.

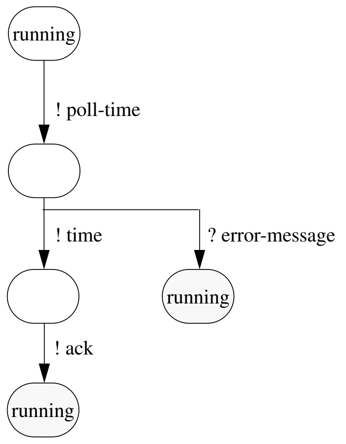
Transitions from state running



error ∈ ERROR-MESSAGES,
result ∈ TIMED-VALUE,
virtual time ∈ TIME,
dump ∈ DUMP.

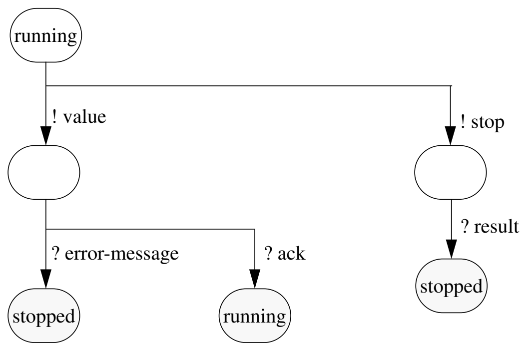


poll-value ∈ POLL-VALUE ∪ TIME x POLL-VALUE, value ∈ TIMED-VALUE,
poll-dump ∈ POLL-DUMP ∪ TIME x POLL-DUMP, dump ∈ DUMP,
ack ∈ ACK, error-message ∈ ERROR-MESSAGES.



poll-time $\in$ POLL-TIME $\cup$ TIME x POLL-TIME, time$\in$ TIME,

ack $\in$ ACK, error-message $\in$ ERROR-MESSAGES.

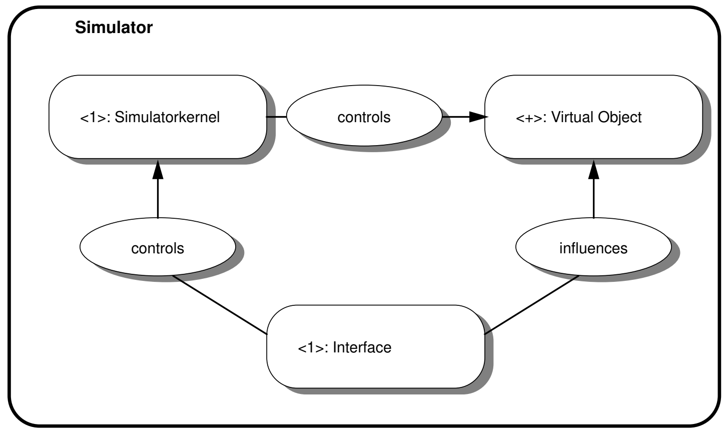


value $\in$ VALUE $\cup$ TIMED-VALUE,

error-message $\in$ ERROR-MESSAGES, ack $\in$ ACK,

stop = “stop”, result $\in$ ACK $\cup$ ERROR-MESSAGES.

1.8 System Structure



II. Non-functional Requirements

The non-functional requirements include the following aspects:

- object-oriented design
- encapsulation of the physical aspects within the objects
- 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 realtime as possible.
- The simulator should run in at least real time.
- Simulation time has to be as close to realtime as possible. This is especially important for outputs.
- It is not allowed to create internal feedback loops without delay of simulation time of outputs to inputs of the simulator

Change Log

03/14/95. Message sequence charts changed. because the original ones were not detailed enough

03/14/95. Problem description revised. because original description was to detailed (contains too much domain knowledge which is encapsulated in the objects) and it did not separate between the desired behaviour of the simulator and the realization (event-driven).
T. Deiß and B. Geppert

03/14/95 Dictionary added. because it was missing.
T.Deiß and B. Geppert

03/20/95 Refinement of 1.5-1.7. because thought necessary.
D. Zeckzer

04/20/95 Addition of new 1.3-1.4, Dictionary revised. because 1.3 and 1.4 needed refinement, dictionary changed according to new items added
P. Sturm

04/20/95 Revision of document. due to changes resulting from the design process.
D. Zeckzer

05/11/95 Revision of document. Besides minor changes, the paragraphs describing the scheduling action (section 1.3) are changed to a set-oriented view. Also the dictionary now distinguishs between kernel events and interface events.
P. Sturm