Project:	D1.Project1
Name:	Building Simulator Simple1
Document:	The Simulator Kernel
Responsible:	Deiß, Geppert, Kamsties, Sturm
File-Pathname	irz1:~sfb501/simulator_kernel/sim_kernel/v2.2

Type of change	Date	Version	Person
Start	21/3/95	1.0	T.Deiß
Formalization	24/3/95	1.1	T.Deiß
Rework	29/3/95	2	T.Deiß
Completion	31/3/95	2.1	T.Deiß
Rework	18/4/95	2.2	B.Geppert
Rework	20/4/95	2.3	B. Geppert
Rework	27/4/95	2.4	P. Sturm
Rework	15/5/95	2.5	P. Sturm

The Simulator Kernel

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:

The simulator kernel has the task to manage a set of events and to trigger objects at the simulation time as described by the collected events. The simulation time shall be related to the realtime (compressed by the timing factor). The simulation time need not to be in complete synchronization with the interface.

1.2 Dictionary

The following items are used in addition to those in the dictionary of the requirement specification

event type. the events can be distinguished by their event type: stop, simulate, set, get, dump, halt.

priority class. Priority classes offer a set of individual priority values to be used for ordering purposes (e.g. performance reasons). Actually, only the class PSimulate has a size greater than 1. The values of all priority classes are mapped to a flat value space, thus defining an overall odering within a class and between classes.

priority of an event (event priority) . This is a concept different to the priority on the input streams. It is used to serialize some of the events. E.g. Each simulate event of one simulation step must be finished before any output event of this step is triggered. ~~The priority can be seen as a tuple of positive natural numbers. The first component is stronly related to the event type of the event. The second component may be used to serialize events of the same type.~~
Let pstop, psimulate, pset, pget, pdump, phalt be the priorities (elements of the approrpiate priority classes) of the stop events, simulate events, input events, output events, dump events, resp. halt events. Then they will be ordered—by default—as follows pstop < pset1 < psimulate < pset2 < pget1 < pdump < pget2 < phalt, where smaller in this ordering means a high priority. This ordering may be changed by the customer before the simulator is compiled. The priority psimulate is treated as a single value, although it maps to an intervall of priority values to enable the transformation of the continuous and discrete simulation paradigm into the event-driven model.
At the moment, these priorites are not visible or ajustable for users via the interface. The simulation objects can influence the second component of the simulate events.

event collection: set of all events processed by the kernel. The events are sorted according to their timestamps and priorities.

slack. allowed difference between simulation time and realtime. This indicates by which amount the simulation time may be behind or ahead the realtime to be considered useful.

messages: used to communicate with the objects and the interface, e.g. error messages, schedule/unschedule event, finished, ack , init, end init, ...

1.3 Concept Model

The toplevel concepts of the simulator kernel are depicted in figure 1. In this figure the focus is on the simulation kernel, therefore the interface is seen as one monolithic block, which behaves in some sense as a domain object: It can be triggered and as a result it may insert new events in the event collection. It may also schedule new events asynchronously.

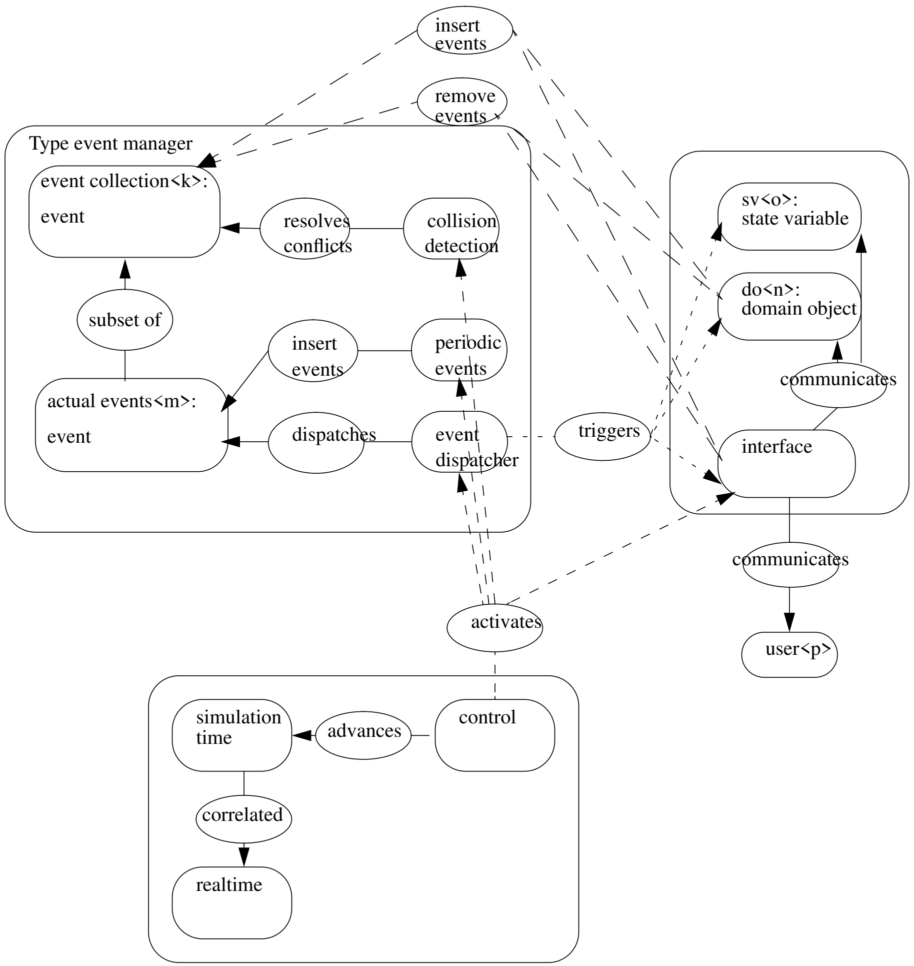


Fig. 1: Concept Model

The concept of an event seems to be unclear and must therefore be clarified. At first an event is something that actually happens, e.g. the change of the value of a state variable, an object performs its simulation, or the simulation time advances. The other notion is that of an event stored in the event collection, which can be seen as a description of a task which has to happen in the future. These events are desribed in figure 2. The timestamp describes the simulation time, at which this event has to happen.

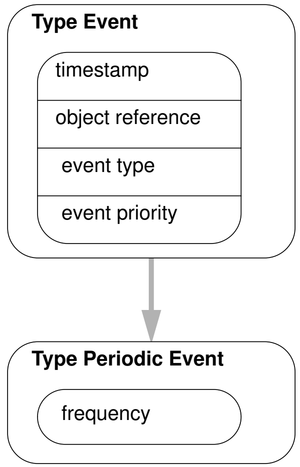


Fig. 2: Type definition of event

The object reference identifies the object which has to perform this event (or task), the event type describes what shall happen. The event priority is used to serialize the execution of some of the events. In case of periodic events, the frequency defines in which intervalls the events has to be scheduled automatically by the kernel.

Input events needn‘t have a timestamp, e.g. if they come from an interactive stream. The kernel is responsible to stamp these events before inserting them in the event collection. The events in the kernels event collection can be distinguished by their types (stop, simulate, set, get, dump, halt). The event collection is sorted according to the timestamps and the event priorities (s. requirement_specification: concept model). The actual events are a subset of the event collection consisting of those events which shall be dispatched in the actual step. It‘s possible to extend the actual events during the actual step as long as the priority of the new event is higher than that of the actually dispatched event.

Collisions such as ‘perform all set value of variable before simulation of an object’ or ‘settings of the same variable at the same simulation time with different values‘ are resolved in the event dispatcher using priorities.

The simulation time can be seen as the clock, which is used in the simulation. It ticks with a fixed interval, e.g. 1 tick corresponds to 1 second simulation time and it has a definite start time set before the start of a simulation run. It is coupled with realtime by the timing factor. It shall be possible to recognize that the simulation time is no longer well synchronized with the (compressed) realtime and it shall be possible to react in a useful way by issuing warnings or by aborting the simulation. Therefore there must be appropriate parameters indicating an allowed slack between simulation and realtime. The simulation time may never be behind realtime by more than the defined slack, as least as the simulation time can be seen from the users.

1.4 Specialization Hierarchies

1.4.1 Specialization of event

The input, output, and command events are specializations of the types event or/and periodic event. They can be distinguished by the the possible event types and the possible destination objects.

1.5 Context

1.5.1 Context Diagram

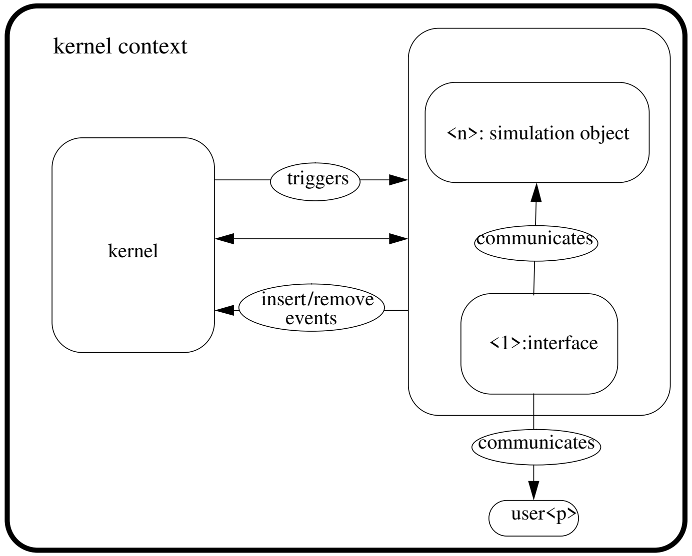
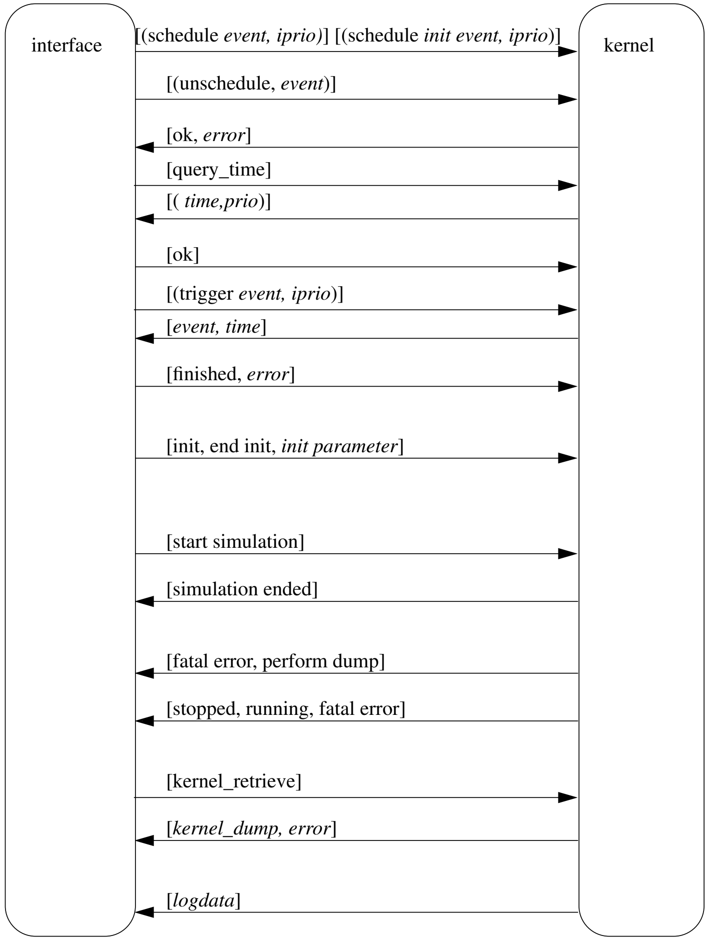


Fig. 3: Kernel Context

1.5.2 Input and Output sets between Interface and Kernel

The following figure describes all possible messages between interface and kernel.



The following definition have been used:

Which events can be sent, scheduled, resp. unscheduled:

init event := (event type, prio, obj)
event := (event type, time, prio, obj) v (event type, prio, obj)

event type := {set, get, various application-defined simulate event types, poll-kernel_dump, halt, stop}
time := timestamp (the timestamp of an event may be 0, indicating an event to be stamped by the kernel).
iprio : input priority to solve input collisions
prio := event priority to serialize the events
obj : internal representation of an object (object reference)

Errors: Together with an error message it is indicated, how severe the error is.

error := *string* x *gravity*
gravity := {light, fatal}

The initialization parameters:

init parameter := {*simulation interval, slack, timing factor, logging flag*}
simulation interval := (*time, time* U infinity). infinity means that there is no a priori end of the simulation.
slack := absolute limit the simulation time might be ahead or behind the corresponding realtime.
timing factor := *N+*
logging flag := *boolean.* Indicates whether all events dispatched should be logged.

State to retrieve and dump:

kernel_dump:= an representation of the kernels state, including the parameters of the simulation time, the event collection, and the actual events.

data for logging the kernels activities

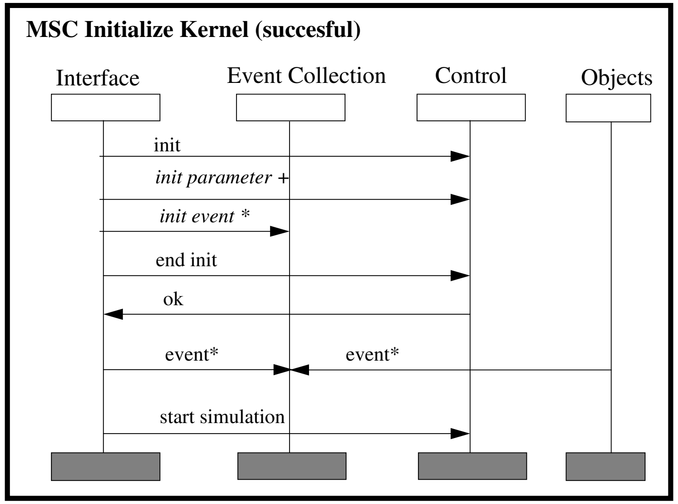
logdata := events together with a realtime timestamp (precise format has to be clarified)

1.6 Message Sequence Charts

To describe the dynamic behavior of the kernel we have to show what happens during initializition and what constitutes one step in the simulation.

1.6.1 Initialization of the the kernel

The kernel has to be initialized before each simulation run. The following *parameters* can be initialized. *Starting and ending time* of the simulation run, the allowed *slack* between simulation, and realtime and the *synchronization flag*. It is mandatory to set the starting time, the other parameters are optional. The start time must be known before the first event is sent to the kernel collection. During the initialization the interface may insert *init events* in the event collection which have to be dispatched before the actual simulation starts, i.e. before the simulation time starts to tick. After the kernel initialization it is checked whether the parameters and init events are useful and whether all necessary data has been set. At the end of a sucessful kernel initialization the event collection is empty.
The kernel is explicitly informed by the interface when the simulation has to be started. At this point in time it is assumed that the interface and the objects have all their first events in the kernel collection and then the first step of the simulation is performed. The simulation time of this first step is the initialized start time. It is a task of the interface (because the kernel does not know the objects) to ensure that the kernel is not started as long as objects (and the interface) are inserting events.



If the initialization phase of the kernel is not correct a negative acknowledge (including an error message) is sent to the interface and no events are accepted in the event collection of the kernel, and the kernel refuses to accept the start simulation message. The initialization of the kernel has to be repeated. It is sufficient to retransmit the problematic parameters (these are indicated in the error message)

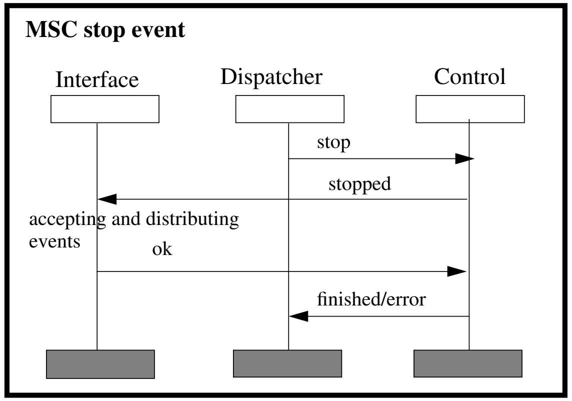
1.6.2 One simulation step

A typical simulation step can be seen as a sequence of different actions or phases. These phases are separated by the priority of the events. It is sufficient to describe the behavior of the kernel when simulating the events of one priority.

Let us assume, that the acual simulation time is t. It may now proceed immediately. From now on the kernel collection will only accept new events with timestamps in the future with respect to the current simulation time or with the same timestamp, but lower priority (event priority with a greater value). Here, only the first component of the priority is used, that is no new events of the same type are accepted, except they have to happen in the future.

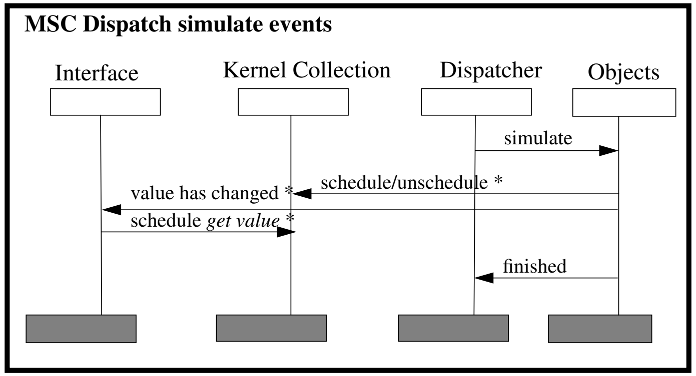
After starting the simulation the kernel may immediately proceed to dispatch the events of the actual group. Thereby it respects the priority of the events in the group. As long as it is guaranteed by the triggered objects that there are no possible collisions, several events with the same priority may be scheduled in parallel. In the case of simulate events an unique message finished is required for each triggered simulation step to guarantee that all objects activated directly or transitively are passive again. This phase ends if all finished messages are received. Then the kernel may immediately proceed to the next group of events in its collection.

1.6.2.a Stop and Start events. The intended meaning of a stop command is to enable an user to insert events interactively. A second possibility is to insert a retrieve command, see the appropriate subsection below. After receiving the stop command, the control waits for the termination of triggered simulation steps (e.g. waiting for one or more finished messages in case of simulation events). With the termination of all simulation activities (i.e. all simulation objects are passive again) the control sends a stopped message back to the interface.The interface may now accept new events and distribute it to the appropriately. After the interface has received a start command it sends a message ok to the control, which itself sends a finished to the dispatcher. If synchrony is lost, then instead of the finished a corre-

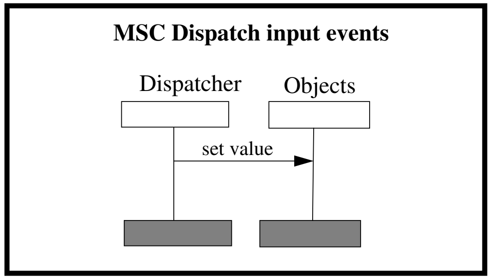


sponding error message is sent back.

1.6.2.b Simulate events. Simulation of an object may lead to the insertion of new events into or the removal of events from the event collection. In the following diagram it is depicted what may happen when one object is simulated. Several events with the same priority may be simulated in parallel. In this case, the message to trigger the simulation has in addition an unique identifier, which is required to be returned together with the finished message, to distinguish, which of the simulations have already come to end and which ones are still running. If a state variable is changed by the simulation and the change of this variable has to be propagated to some user, then a message is sent from the variable to the interface that it has changed its value. In turn, the interface will insert a corresponding event into the event collection to process outputs (but defering the processing to the end of the simulation for the actual time stamp). Thus, the timestamp will be the actual simulation time but lower priority. The following diagram does not want to imply an order, between the times, when the schedule/unschedule, ‘value has changed’, and process inputs events are sent resp. received. But it must be guaranteed, that all simulation objects are passive again, before the finished message is sent back to the dispatcher. As usual, an object may sent an error message instead of its finished. Be aware, that while dispatching simulate events of priority (p1,p2), it is not allowed to insert events with the actual simulation time as timestamp and priority (p1,p3), even if p3 is lower than p2.

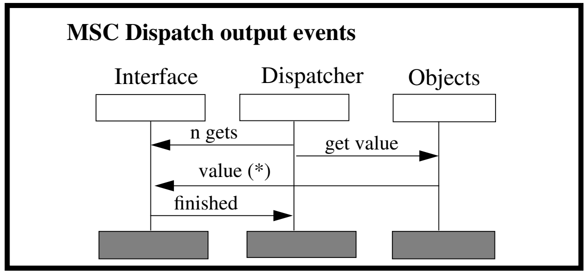


1.6.2.c Input events.



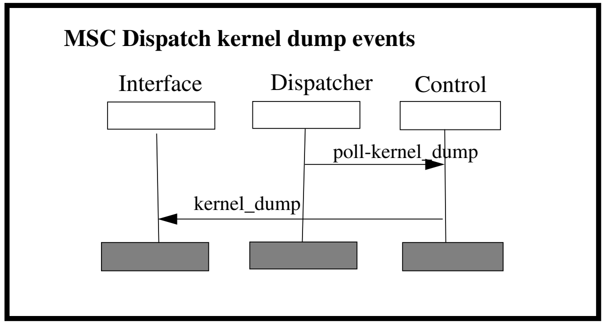
It has to checked whether setting a state variable may have side-effects possibly causing the scheduling of new events. In this case the finished mechanism has to be used as in the case of simulate events to guarantee consistency.

1.6.2.d Output events.

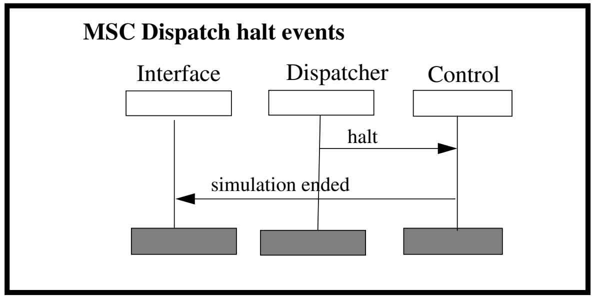


1.6.2.e Dump events. The interface may map a dump into several *get events* and one *kernel_dump event*. The internal state of the kernel consists of the *kernel collection*, and all *parameters of the time* (including the actual priority)).

The aspect of dumping has to be discussed with the customer again. The actual specification is too detailled and limits implementation. Especially, efficient dumping is prohibited.

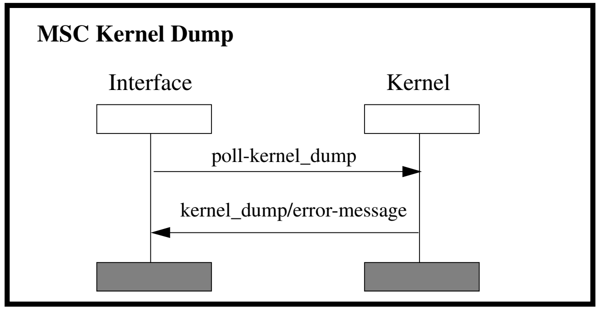


1.6.2.f Halt events. This means, that the current simulation run has ended. The state of the kernel will be preserved up to the next initialization or a ‘shutdown’ of the complete simulator. The control sends a message ‘simulation ended’ to the interface. From then on it will not dispatch new events. This enables to dump a state which can be consistently continued still after the halt command. It will not be possible to restart this simulation run, except via a retrieve command with the ending state as starting state of another simulation run.



1.6.3 Dumping and Retrieving

If the interface is performing a dump, it will request the internal state of the kernel. This internal state comprises the event collection including the actual events and the parameters of the clock including the actual priority. The state returned by the simulator kernel is guaranteed to be consistent (no simulation object has been active at the time of the dump and no other event was elected to be triggered). If the dumping of the simulation object’s state is performed immediately after the receipt of the kernel dump state, this overall state of the simulation objects is also consistent (as long as the retrievel of this state is performed correctly by the simulation objects).



The retrieval of a dump will only be possible if the simulator is stopped by a stopped command, or before starting a simulation run. In case, that the state of the kernel is not retrieved, this can be seen as a set of new variable settings of the state variables of the simulation objects. This may cause new schedule/unschedule messages from the objects as usual. If the state of the kernel is to be retrieved too, this has to happen before the retrieval of the object part of the dump. Otherwise, sending schedule/unschedule messages to the kernel is useless, as this state will be overwritten by the dumped state of the queue. The sequence of messages is trivial, therefore, no MSC will be drawn. An error message is sent to the interface if the retrieved data cannot be used to rebuild a consistent internal state of the kernel. If this happens, the kernel will not proceed or start a simulation run.

The precise format of the dumped data still has to be cleared together with the interface group.

Again, dumping has to be discussed with customer.

1.6.4 Maintaining realtime binding

It is an ongoing task of the control to check whether the simulation time is within the allowed slack with realtime. Activating this realtime binding is optional and is defined by the user of the simulator as a parameter. This check is done with the termination of each single simulation step. In case the upper limit of slack is exceeded, the kernel stops the processing of any further scheduled events and issues an error message.

1.6.5 General error handling

Similar mechanisms are used to handle all other kinds of fatal errors. The corresponding error messages will arrive at the dispatcher, which will then inform the control and this will propagate the occurence of a fatal error to the interface. In case of a fatal error the kernel will dump its state on request of the interface.

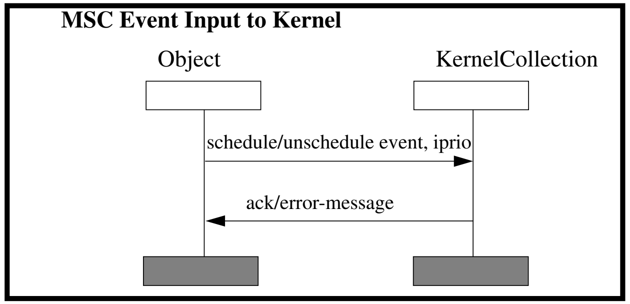
If it is possible to continue the simulation run though an error occured. The dispatcher will propagate the error message to the interface which has to present it to some user.

Some examples of errors:

- severe internal error of an object while simulation (e.g.numerical stability is lost): fatal
- trying to unschedule a nonexisting event: warning
- the kernel collection refuses to schedule an event of an object: warning/fatal
- the kernel collection is full: fatal
- insertion or removal of ‘old’ events: warning
- events with inappropriate priorities: warning
- lose of synchrony: fatal
- duplicated finish messages: warning

1.6.6 Inserting new events in the kernel collection

Each time, an object (or the interface) sends a schedule or unschedule message to the kernel collection, there will be an acknowledge. A positive one, if the collections accepts to store the event (this does not mean that it is actually stored. In case of a fixed deadline, the collection is allowed to discard events beyond this time). A negative acknowledge is sent, if the collection refuses to store the event. This is the case, if the timestamp and/or the event priority is not in the future with respect to the current simulation time, or if there is no space to store this event (These two errors should have a different gravity). Another possible error is an input collision which can be solved by input priorities *iprio* and could cause a warning. It is also considered an error to unschedule a nonexistent event. It should be noted that the interface is allowed to insert events without an timestamp. In this case the kernel is responsible to stamp the event with the next possible timestamp before inserting it into the collection.



1.6.7 Logging the kernels activities

To analyze one simulation run it might be necessary to repeat the run or to collect a lot of information, while the run is going on. In case of logging, the kernel behaves as a simulation object—with respect to the interface—requesting output.

This information will be sent asynchronously to the interface without any coupling to the realtime. The interface should convert internal names to external ones to allow that the informatin can be understood by some user. No acknowledge messages of the interface are expected. The detailed format of the messages is not clear up to now.

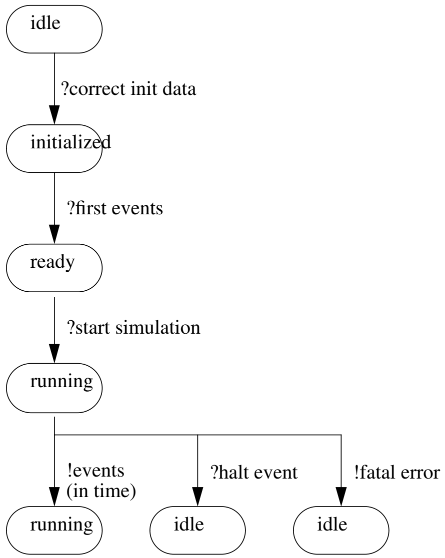
This functionality can be switched on by an appropriate parameter.

1.7 Role Behaviour between event stream and Simulator

The Kernel will be in one of the following states:

- Uninitialized,
- initialized,
- performing simulation steps

The behavior of the kernel can be seen in the following transition chart. This chart is on a very abstract level and does not show what happens in one simulation step.,



1.8 System Structure

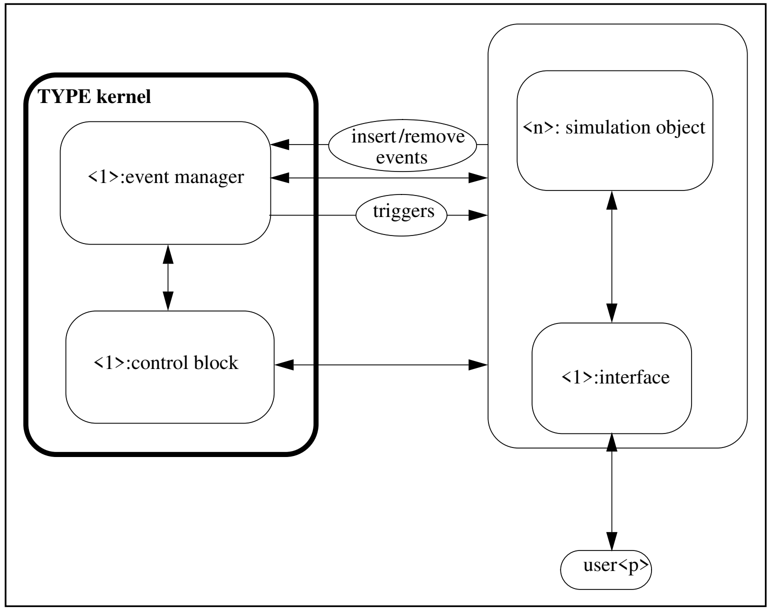


Fig. 4: System Structure: Kernel Type

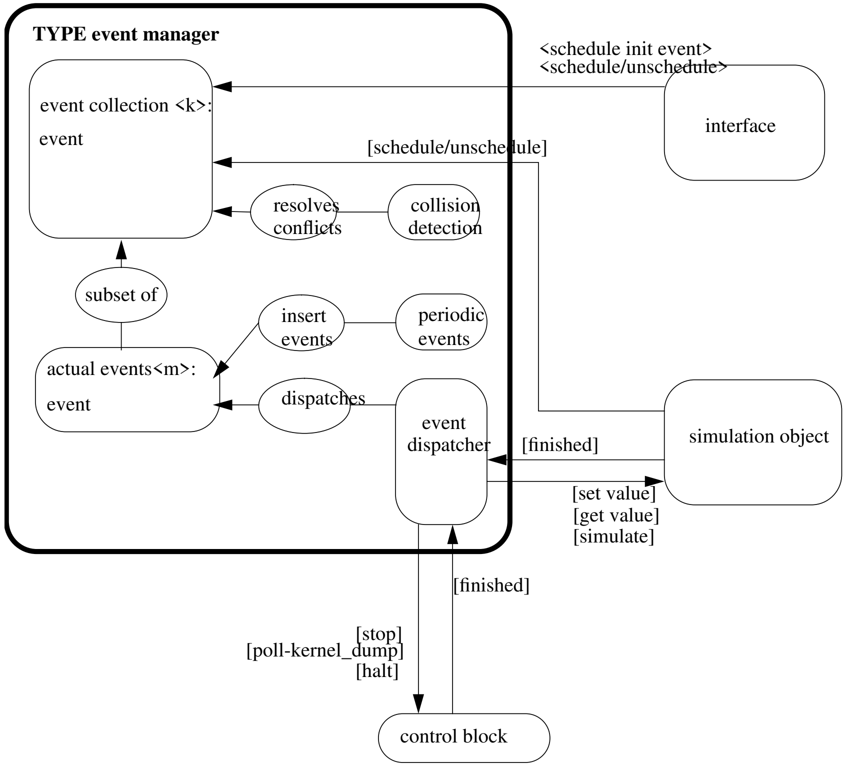


Fig. 5: System Structure: event manager

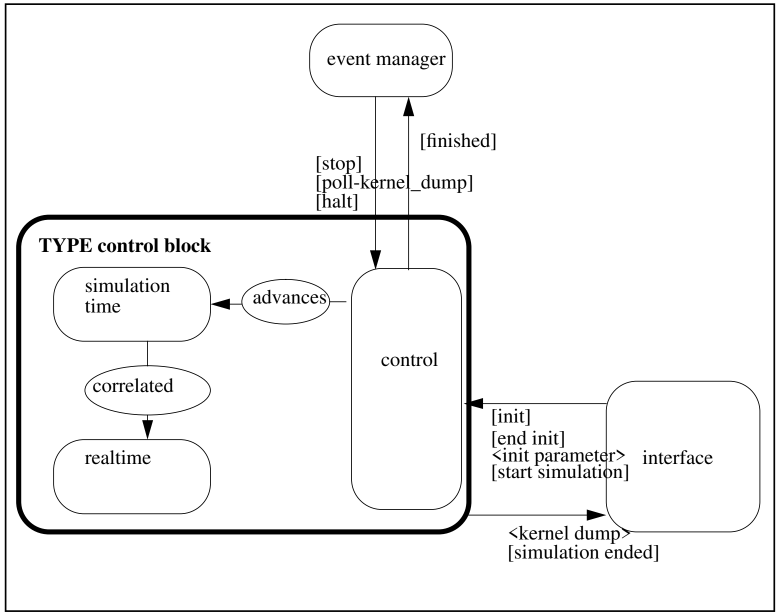


Fig. 6: System Structure: control block

II. Non-functional Requirements

The non-functional requirements include the following aspects:

- the accuracy should not be higher than necessary to test control algorithms and provide resonable 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.

Change Log

24.3 some formalization has been added to be more precise. A lot remains to be done.

T. Deiß

29.3 Due to team review (together with interface team) internal priorities for the dispatcher are added. The synchronization between interface and kernel can be weakened (adjustable by parameter). T.Deiß

31.3 Due to misunderstandings, the notion of slack and how it is measured has been defined new. T.Deiß

31.3. Missing transition chart entered. T.Deiß

18.4 Due to team review (kernel + interface team) :

- all events are collected and managed in the kernel.
  Exception: *dump* is an interface-event, i.d. the interface maps a *dump* into several *gets* and one *poll-kernel_dump.*
- incoming events needn‘t have a timestamp; kernel is responsible to stamp these events
- context diagram and system structure added

B. Geppert

26/27.4. Review and various minor changes and deletions.

P. Sturm

15.5. All corrections and deletions—as discussed in the group—performed.

P. Sturm