Discrete Event Modelling and Simulation

CS522 Fall Term 2001

Hans Vangheluwe

For a class of formalisms labelled *discrete-event*, system models are described at an abstraction level where the time base is continuous ($\mathbb{R}$), but during a bounded time-span, only a *finite number* of relevant *events* occurs. These events can cause the state of the system to change. In between events, the state of the system does *not* change. This is unlike *continuous* models in which the state of the system may change continuously over time.

Discrete-event formalisms are clearly at a high level of abstraction. This abstraction is often appropriate for realistic representation of a system's behaviour. Furthermore, as in between events, the state of the system does not change, a discrete-event simulator need not explicitly represent the state of the system at non-event times. This allows for highly *efficient simulation* as compared to continuous simulation, where in principle, state information must be represented at each point in continuous time.
The high level of abstraction may however introduce simulation *artifacts* which do not pertain to real-world behaviour. In particular, event *simultaneity* whereby multiple distinct events occur at exactly the same time may be due to an insufficiently detailed discrete-event model. The DEVS formalism and its derivatives rigourously describe the semantics of such event *collisions*. A detailed presentation of the semantics of *pinnacles* and *mythical states*, which occur when events are used for respectively *time-scale abstraction* and *parameter abstraction* of continuous phenomena in hybrid models, is given by Mosterman and Biswas in [MB01].

The simple example system depicted in Figure 1 will be used to illustrate relevant concepts. At the physical level, the system consists of a cashier serving arriving customers, one at a time. Customers queue if the cashier is not available (serving another customer). Here, the state of the system consists of the state of the queue and that of the cashier. The queueing discipline is First In First Out (FIFO) and individual customers are assumed not to have any distinguishing features (such as age, or number of items bought). Thus, it is meaningful to model the state of the queue by means of the queue length, a natural number. The cashier can be in either the Idle or the Busy state. The dynamics of the system is determined by:

- the arrival pattern of customers characterized by their Inter Arrival Time (IAT) distribution,
- the time required by the cashier to serve a customer characterized by the Service Time (ST) distribution,
- the logical sequence of customers progressing through the system under different conditions (queue empty/not empty, cashier Busy/Idle).

In Figure 2, an example behaviour of the cashier/queue system, its reaction to a particular input segment of customer arrivals, starting from an initial state, is depicted.

1 Definitions

The following (due to Nance [Nan81]) enable a correct understanding of different types of discrete-event simulation models.

- An *instant* is a value of system time at which the value of at least one attribute of an object can be assigned.
- An *interval* is the duration between two successive instants.
- A time *span* is the contiguous succession of one or more intervals.
- The *state* of an object at a particular instant is the enumeration of all attribute values of that object at that instant (mathematically a tuple, element of the product set of all attribute value sets). The state consists of all the object states at a particular instant.

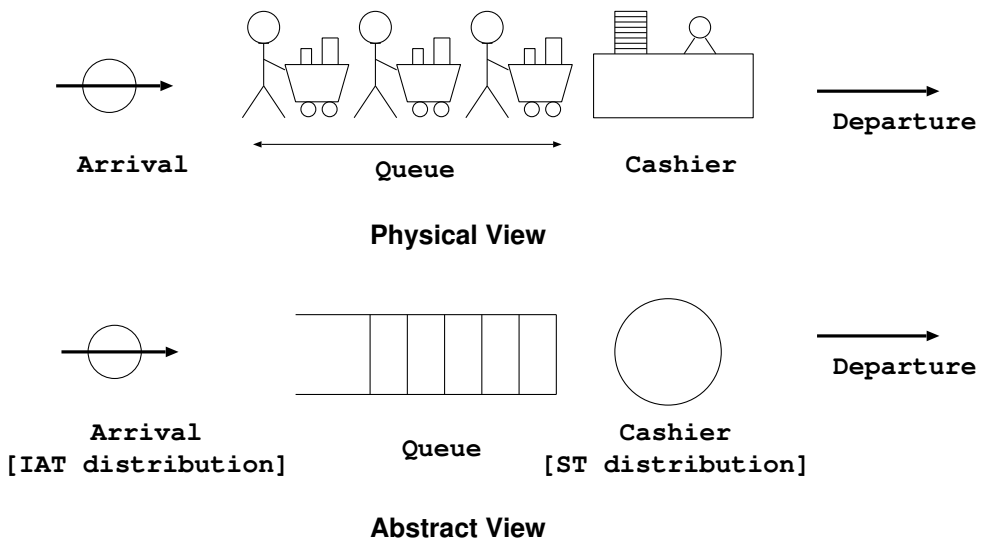

Figure 1: Single server queuing system

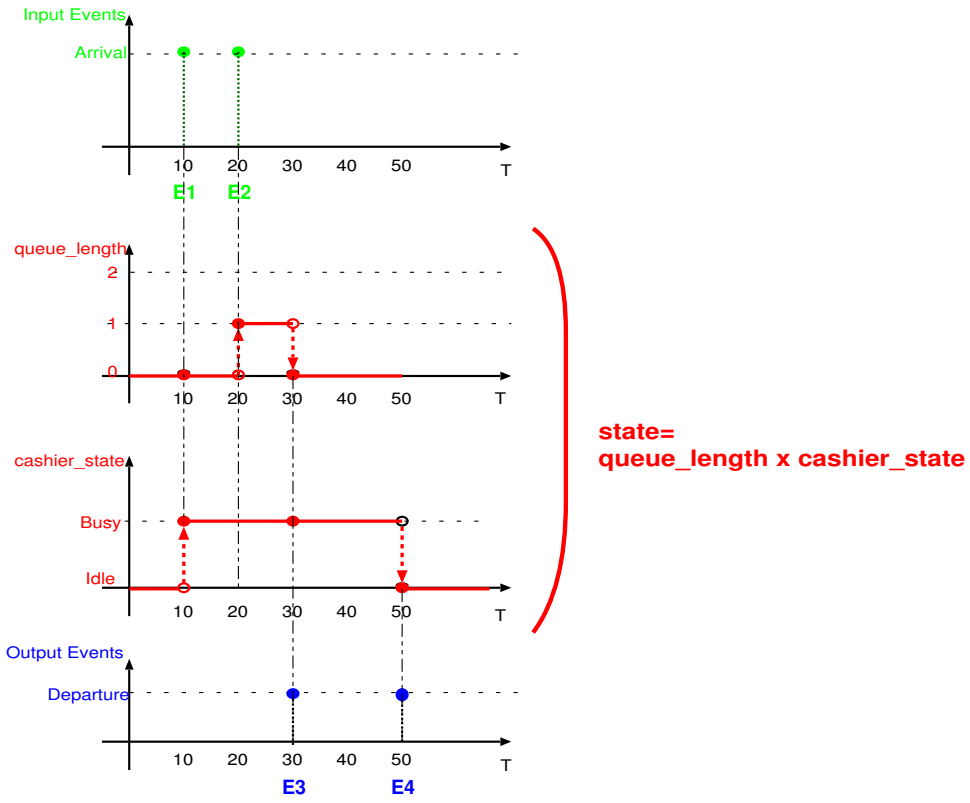

Figure 2: Queueing system state trajectory

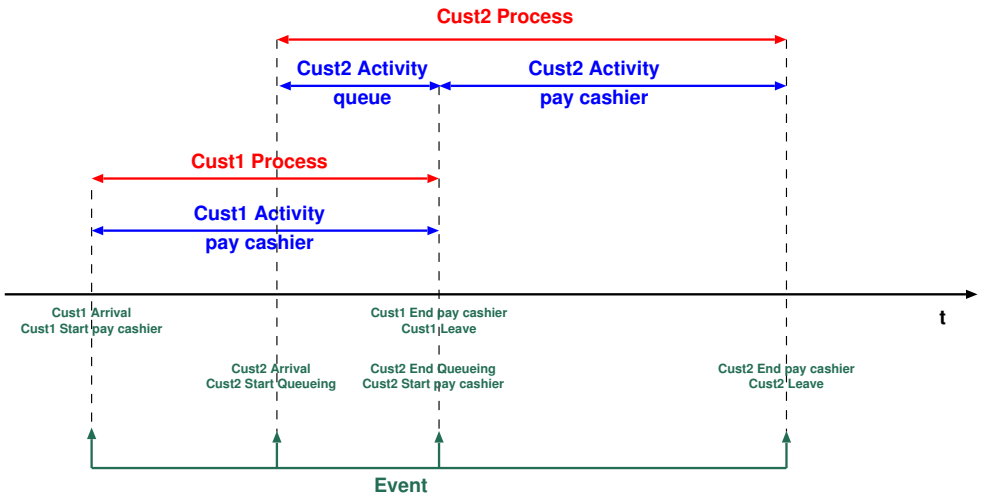

Figure 3: Event/Object Activity/Process

A simulation model has a *static structure* and a *dynamic structure*. The static structure specifies the possible states of the model. The dynamic structure specifies how the state changes over time. The static structure is usually described as a collection of objects and their attributes [CS92]. There are different approaches, known as *world views*, to representing the dynamic structure of a model. The following concepts are at the basis of the different world views:

- An *activity* is the state of an object over an interval.
- An *event* is a change of object state, occuring at an instant, that initiates an activity precluded prior to that instant. An event is *determined* if the condition for event occurence depends exclusively on system time. In hybrid simulation modelling this is called a *time event*. Otherwise, the event is *contingent* (dependent on system conditions). In hybrid modelling, this is called a *state event*.
- An *object activity* is the state of an object between two events describing successive state changes *for that object*. Other events may occur, related to state changes of other objects.
- A *process* is the succession of states of an object over a time span. This is equivalent to the contiguous succession of one or more object activities.

Events, activities and processes for the cashier/queue example are depicted in Figure 3. For a given problem, the following steps are followed to determine what the events are:

1. Identify objects and their attributes.
2. Identify attributes of the system.
3. Define what causes changes in attribute value as an *event*.

Often extra state variables are added to allow calculation of *performance metrics* such as counters, minima and maxima, averages, and frequency distributions of relevant variables. In discrete event simulation, one is mostly interested in the values of performance metrics such as average queue length and utilization of resources. This is in contrast with continuous simulation, where one is mostly interested in the explicit state trajectory. The performance metrics are output at the end of a simulation run.

In the following sections, the different world views are presented by means of an *operational* definition of their simulation kernels.

2 From Untimed to Timed formalisms

Previously, *untimed* formalisms such as State Automata, State Charts, and Petri Nets were introduced. In these formalisms, only the *order* in which events occur is represented (in the form of a time-base $\mathbb{N}$), not the explicit (time-base

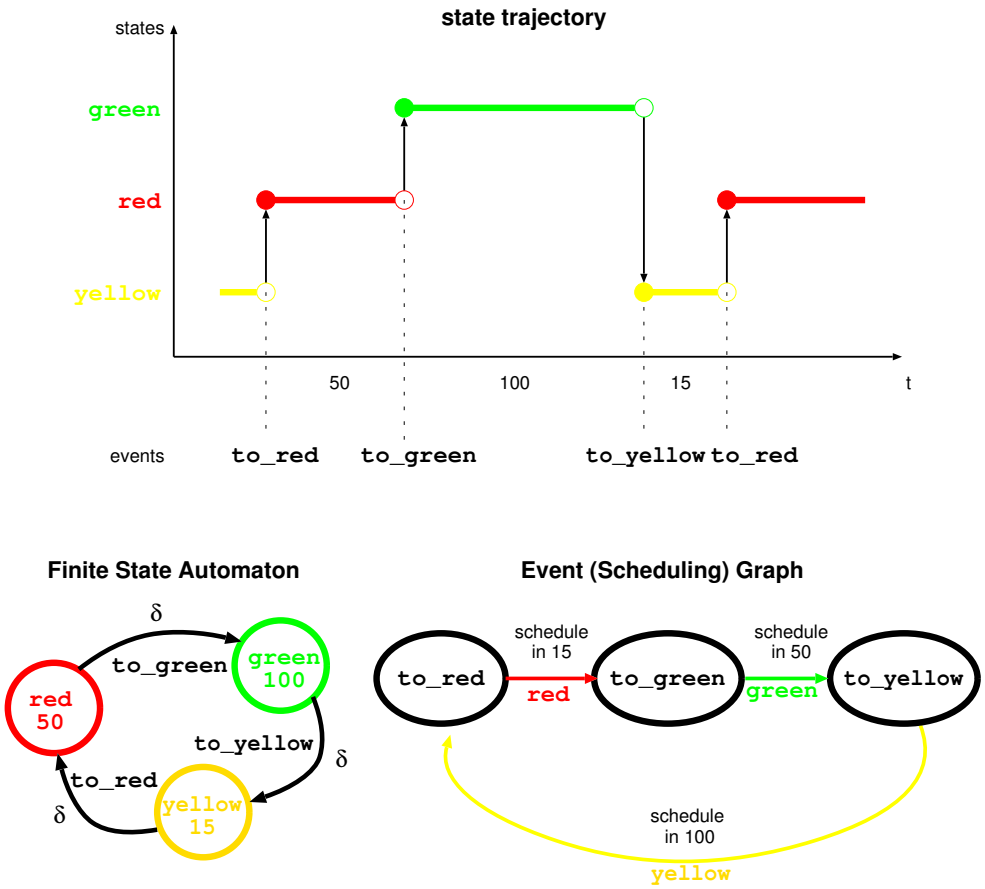

Figure 4: Finite State Automaton and Event Graph models

$\mathbb{R}$) time at which they occur. On the one hand, this implies a loss of information. On the other hand, it allows one to describe *classes* behaviour and *prove* properties of these. All the above formalisms can be extended to include explicit timing information. In particular, State Automata can be extended to include the time the system stays in a particular state before making a transition to the next state. As shown in Figure 4 (the state trajectory of a traffic light) it is always possible to construct an Event Graph which has as nodes, the transitions, and as edges, the time interval after which the next transition is scheduled to occur. This demonstrates the link with the event scheduling discrete event world view introduced in the next section.

3 The Event Scheduling world view

In the *Event Scheduling* world view, a model describes, for each of the events, the event's effect

- on the state,
- on the future behaviour of the system. This is achieved by *scheduling* new events into the future.

An event scheduling model for the single queue, single server example is given below:

```
declare variables:
 t              : Time
 queue_length   : PosInt
 cashier_state  : {Idle, Busy}

declare events:
 start, arrival, departure, end

define events:

 start event:
  /* scheduled first automatically by simulator */

  /* initializations */
  queue_length   = 0
  cashier_state  = Idle

  /* schedule end of simulation */
  schedule end absolute end_time

  /* schedule first arrival */
  schedule arrival relative 0

 arrival event:
  schedule arrival relative Random(IATmean, IATspread)
  if (queue_length == 0)
   if (cashier_state == Idle)
    cashier_state = Busy
    schedule departure relative Random(SERVmean, SERVspread)
   else
    queue_length++
  else /* queue_length != 0 */
   queue_length++

 departure event:
  if (queue_length == 0)
   cashier_state = Idle
  else /* queue_length != 0 */
   queue_length--
   schedule departure relative Random(SERVmean, SERVspread)

 end event:
  /* terminates simulation */
  /* process/output performance metrics */
  print time, queue_length /* current */
  print average_queue_length
```

As shown in Figure 5, an event scheduling simulation kernel uses two (global) data structures. One contains the state variables declared in the model. The other contains scheduled event notices in an event list, ordered by increasing time and decreasing priority. When scheduled, events are always added from the rear. Priorities are used to choose between events occurring at the same time (collisions). The state variables may be augmented by additional performance variables for calculation of minima, maxima, mean, standard deviation*, etc.* of state variables and combinations of them. An event scheduling kernel operates by ordering (according to increasing time) scheduled events in the event list and iteratively removing and processing the head of that list until the list becomes empty. The event time of the event notice is used to advance the simulation time. Depending on the event type of the event notice, the appropriate event notice is invoked. This routine may modify the system's state and schedule new events into the future by placing event notices in the event list. As an example, part of the evolution of the state and event list during a typical event scheduling simulation of the Cashier/Queue model is shown in Figure 6.

In the spirit of the Event Scheduling formalism,

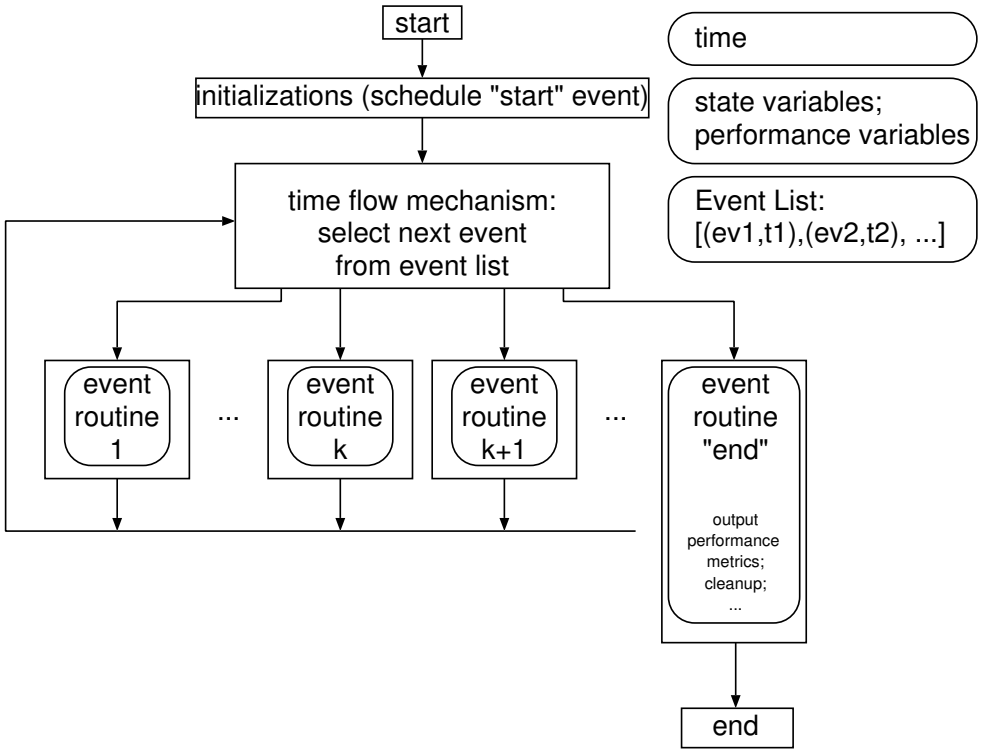

Figure 5: Event Scheduling simulation kernel

- initialization of the system state as well as pre-scheduling of events may be put in a "start" event. This event is automatically put in the event list and subsequently processed (first) using the same procedure as for any other event.
- halting the simulation at a certain simulation time can be achieved by scheduling a special "end" event which is recognized by the simulation procedure as the last event to be processed (even if more event notices are present on the event list). This event may contain terminal processing instructions, mainly generating output of performance measures and other gathered statistics. Caveat: it may be necessary to re-schedule the "end" event or to give it the lowest priority to avoid missing an event occurring exactly at the time of the "end" event.

The above informally describes an event scheduling model in terms of a system state, events, an event list, and how an event influences the system state and event list (schedule new events in future). An event scheduling model is simulated by a simulation procedure which iteratively advances simulation time, updates the event list, as well as the system state.

The model representation as well as the simulation procedure are presented here in mathematical form, to facilitate the description (in the next section) of the mapping onto the DEVS formalism.

For the sake of simplicitly, we currently ignore

- output of the model, as this can easily be added and does not change the essence of the formalism;
- external events interrupting the autonomous behaviour of the system, as this is not normally part of the event scheduling formalism. This implies that hierarchies of event scheduling models can not be described. External events and hierarchy can be added easily after mapping onto DEVS.

The structure of an event scheduling model ES is

$$ES = \langle T, E, S, EL, \delta_t, \delta_\eta, \delta_S \rangle.$$

In this structure, T is the Time Base

$$T = \mathbb{R}.$$

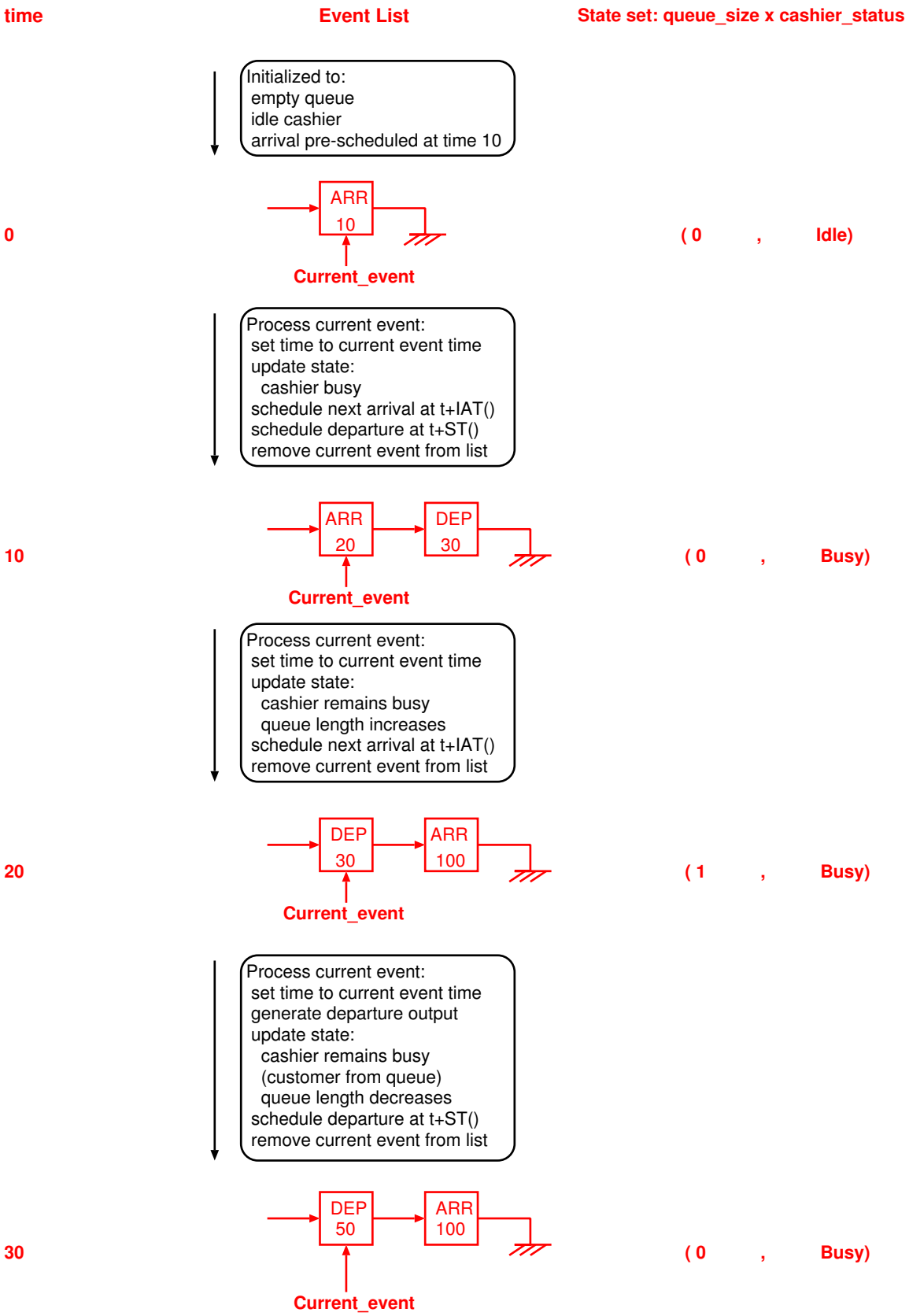

Figure 6: Event Scheduling simulator at work

The finite set E contains unique *event types* η such as "arrival of customer 1" and "departure of customer 5". Note that an actual *event* occurrence is characterized by a tuple (η, t) including the event type and the event instant. When present on an event list, this tuple is called an *event notice*. Events may be divided into "classes" (C) such as arrivals and departures (of different entities):

$$E = \bigcup_{i \in C} E_i.$$

It may be necessary to define an order relationship $\leq$ over E: $(E, \leq)$ to "encode" *priorities*. For example, $arrival_{customer} < arrival_{manager} < departure_{customer} < departure_{manager}$ means *arrival* events have lower priority than *departure* events and within these two event classes, managers have higher priority than customers. This is a common approach to encode event selection when it is necessary to choose between multiple events occurring simultaneously (a collision) such as $\{(\eta_1, t), (\eta_2, t), \ldots\}$ on the event list. If priorities do *not* resolve a collision to one single (η, t), the selection becomes implementation dependent. This is not portable across implementations and leads to different simulation results on different platforms. This means simulation experiments are not *repeatable*. If this situation occurs, more detail will typically be added to the priorities ordering $(<)$.

The event list EL is a possibly empty set (or even a bag if the same event is allowed to occur multiple times at the same time) of event notices

$$EL \in 2^{E \times T}.$$

For example,

$$EL = \{\underbrace{(\eta_1, t_1)}_{ev_1}, \underbrace{(\eta_2, t_2)}_{ev_2}, \cdots, \underbrace{(\eta_n, t_n)}_{ev_n}\}.$$

Note how, in an implementation-oriented description of ES, an event list would be described as an ordered list. In a more denotational fashion, not insisting on a particular implementation data structure, we use a set, with the order imposed by a $select\_first()$ operator. This leaves room for efficient, possibly parallel, implementation.

$$\begin{array}{rlll} select\_first: & 2^{E \times T} \setminus \emptyset & \rightarrow & E \times T, \\ & EL & \rightarrow & ev^* = (\eta^*, t^*). \end{array}$$

where

$$\begin{array}{rcl} t^* & = & \min_{(T, \leq)} \{t \mid (\eta, t) \in EL\}; \\ \eta^* & = & select\{(\eta \mid (\eta, t^*) \in EL\}. \end{array}$$

In the above, it is assumed $EL \neq \emptyset$. Simulation halts when the event list becomes empty and thus $select\_first$ will never be applied to an empty event list (as specified in the simulation procedure Algorithm 1).

The *select* tie-breaking function is needed to select between simultaneously occuring events. As mentioned before, *select* is typically implemented based on an ordering relationship $\leq$ over E:

$$\begin{array}{rlll} select: & 2^E & \rightarrow & E, \\ & \{\eta_1, \cdots, \eta_n\} & \rightarrow & \min_{(E, \leq)} \{\eta_1, \cdots, \eta_n\}. \end{array}$$

Applying *select* should yield a unique result. This will only be the case if there is a *strict* ordering (no equalities) over the set of events E.

The state set S is modified at event times by *event handlers*. S may obviously be a product set: $S = \times_i S_i$.

For each $\eta \in E$, an event handler is a structure

$$(\delta_t, \delta_S, \delta_\eta).$$

This allows one to specify the effects of handling an event:

1. modification of the system state $\in S$,
2. scheduling a new event η in the future.

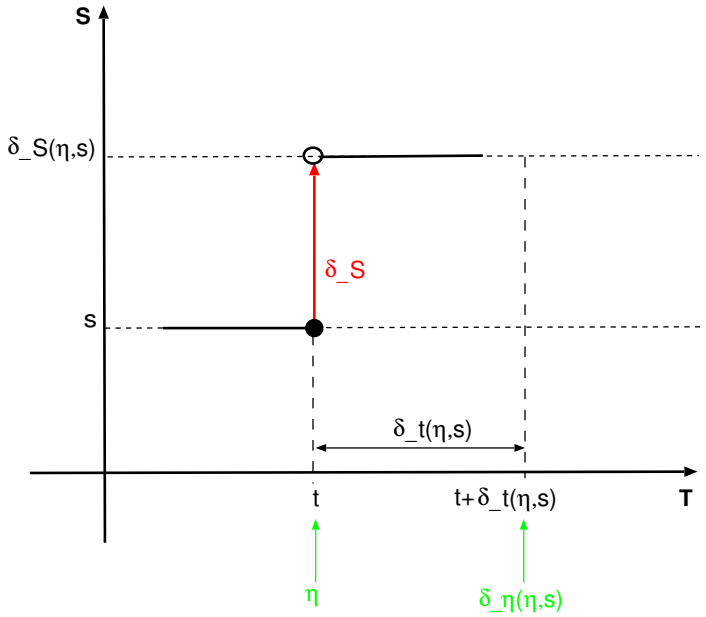

Figure 7: Event handler $= (\delta_t, \delta_S, \delta_\eta)$

In the Event Scheduling formalism as described here, there is only a single modification of the system state, as well as only one event scheduled. This can be done without loss of generality as multiple state changes and future events scheduled are all done at the same instant of time. Multiple state changes and events scheduled can be emulated (modelled) by a sequence of events, each only performing one state change and one scheduling. Note how this is only true if no output is generated for intermediate state changes.

Alternately, the formalism could easily be adopted to lump a sequence of state changes into one resultant state change, and to schedule a series of events in the future.

As shown in Figure 7, the event handler structure specifies:

1. At what time in the future to schedule a new event. The time delay δ_t between the current time and the time the new event is scheduled at is based on the current event and system state:

$$\delta_t : E \times S \to \mathbb{R}^+_{0,+\infty}.$$

2. How to modify the system state based on the current event and system state:

$$\delta_S : E \times S \to S.$$

3. The event type to be scheduled. This is given by δ_η based on the current event and system state:

$$\delta_\eta : E \times S \to E.$$

The above concludes the *structure* of the model formalism. Algorithm 1 describes a *simulation procedure* for simulating an event scheduling model ES. To carry out the simulation, the model (event types, state set, and event handlers) is given together with intial conditions:

- the initial event list (pre-scheduled events),
- the initial state $s \in S$.

Note how lines 4 and 5 in Algorithm 1 are a generalization of what is common in list-based event scheduling implementations:

$$(\eta_{first}, t_{first}) \leftarrow \text{head}(EL),$$

Algorithm 1 Event Scheduling simulation procedure

1: $s \leftarrow$ initial state $\in S$ {initialize the state}
2: $EL \leftarrow$ initial event list {initialize the event list (pre-scheduled events)}
3: **while** $(EL \neq \emptyset)$ **do**
4: $t_{first} \leftarrow \min_{(T,\leq)}\{t \mid (\eta, t) \in EL\}$
5: $\eta_{first} \leftarrow select(\{\eta \mid (\eta, t_{first}) \in EL)$
6: $t \leftarrow t_{first}$ {advance current time to t_{first}}
7: $\eta \leftarrow \eta_{first}$ {event type currently processed}
8: $\Delta t \leftarrow \delta_t(\eta, s)$
9: $\eta^*_{new} \leftarrow \delta_\eta(\eta, s)$
10: $EL \leftarrow (EL \setminus (\eta, t)) \cup \{(\eta', t + \Delta t)\}$ {$\setminus$: remove the current event, $\cup$: add a scheduled event}
11: $s \leftarrow \delta_S(\eta, s)$ {update state}
12: **end while**

where "head" is the standard ordered list operator which selects the first element of the list.

Similarly, the approach on line 10 is a generalization of updating of the event list in list-based event scheduling implementations:

$$EL \leftarrow \text{insert}_{((T,\leq),(E,\leq))}((\eta', t + \Delta t), \text{tail}(EL)),$$

with "insert" and "tail", the standard ordered list operators. The position where an event notice is "insert"ed is determined by the its timestamp as well as its priority. If both are equal, notices are added from the rear. "tail" produces the remainder of EL after removing its "head".

4 The Activity Scanning world view

In the *Activity Scanning* world view, a model describes *conditions* which will activate *activities*. This representation (and its semantics described below) resembles that used in declarative AI languages such as Prolog [CM87].

An activity scanning model for the single queue, single server example is given below.

```
declare (and initialize) variables:
 t             : Time
 queue_length  : PosInt = 0
 cashier_state : {Idle, Busy} = Idle
 t_arrival     : Time = 0
 t_depart      : Time = plusInf

declare activities:
 queue_pay, depart, end

queue_pay activity
condition: t >= t_arrival
actions:
 if (queue_length == 0)
  if (cashier_state == Idle)
   keep queue_length == 0
   cashier_state = Busy
   t_depart = t + Random(SERVmean, SERVspread) /* service time */
  else
   queue_length++
 else /* queue_length != 0 */
  queue_length++, keep cashier_state == Busy
 t_arrival = t + Random(IATmean, IATspread)   /* inter arrival time */

depart activity
condition: t >= t_departure
actions:
 if (queue_length == 0)
  cashier_state = Idle
 else /* queue_length != 0 */
  queue_length--, keep cashier_state == Busy
  t_depart = t + Random(SERVmean, SERVspread) /* service time */

end activity
condition: t >= t_end
actions:
 print t, queue_length /* current */
 print avg_queue_length /* performance metric */
```

As shown in Figure 8, an activity scanning simulation kernel uses a discrete time step to advance time. During the activity scan phase, the solver checks for an activity whose condition (a boolean function of the time variable and the state variables) is true and processes it. This scan is continued as long as some activity condition evaluates to true. If none of the activities is enabled, the time flow phase is executed again, advancing time. In the spirit of the Activity Scanning formalism, a "start" and "end" activity may be defined with semantics similar to their Event Scheduling counterparts.

5 The Three Phase Approach world view

As Activity Scanning uses a fixed time step, it is not efficient. On the one hand, the time step needs to be chosen as small as the smallest time interval possible between two events to correctly model behaviour. On the other hand, some events may be extremely far apart in time (many times the smallest time between events). For such long time intervals, an activity scanning simulator will unnecessarily check all conditions at each point in time despite the fact that the conditions do not change.

In the *Three Phase Approach* world view, Activity Scanning is combined with Event Scheduling. Activities may be scheduled explicitly into the future as in the Event Scheduling world view. In addition, at event times, all activity conditions are checked as in the Activity Scanning world view. Two types of activities are represented:

- "bound to occur activities" (B): are scheduled in an Event Scheduling fashion and describe the effect of unconditional state changes on the current state and on the future (by scheduling new B activities into the future).
- "conditional activities" (C): are invoked at event times if their condition evaluates to true. Describe the effect of unconditional state changes on the current state and on the future (by scheduling new B activities into the future).

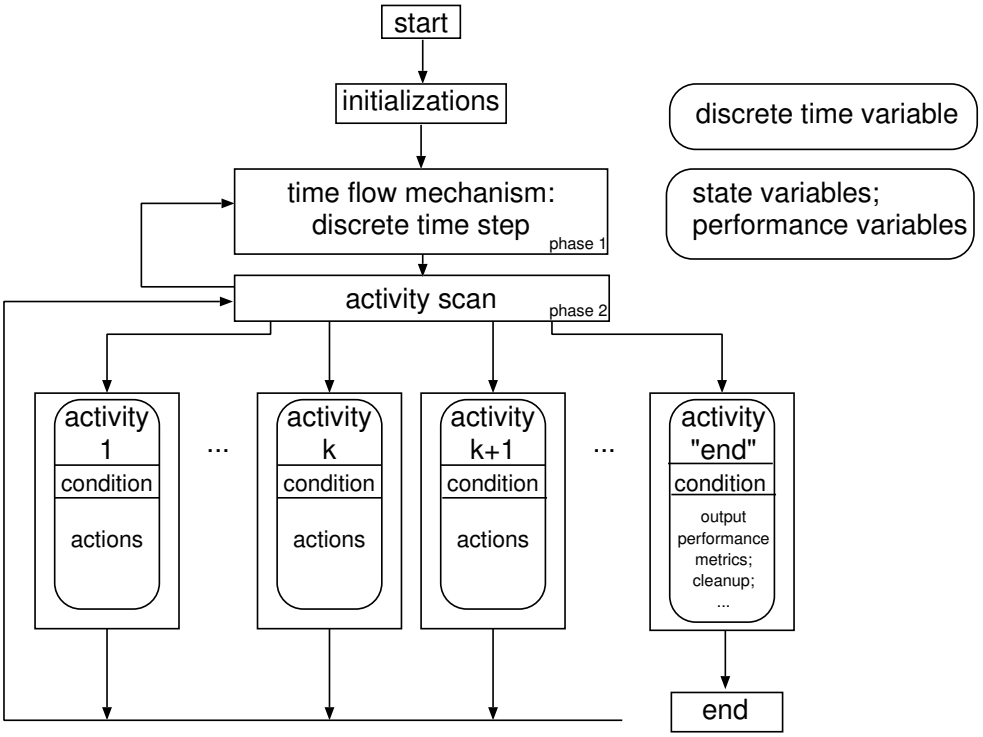

Figure 8: Activity Scanning simulation kernel

As shown in Figure 9, a three phase approach simulation kernel combines the scheduling of B activities of the Event Scheduling world view (with its associated time flow mechanism of advancing time to the time of the first event on the Event List) with the invocation of conditionional C activities of the Activity Scanning world view. Again "start" and "end" activites may be defined. It will be noticed that these and all other activities may be described as B *or* as C activities. This shows the conceptual flaw in the Three Phase Approach: mixing different world views in a single model makes the model hard to understand and maintain.

6 The Process Interaction world view

At the highest level of abstraction, in the *Process Interaction* world view, a template is given for the life of *transactions* or *processes* as they progress through a number of *activities* or *blocks*. In Figure 10, a process interaction model for the single queue, single server example is given in the General Purpose Simulation System (GPSS) language and its corresponding graphical notation [Gor96, Sch74, BCIN98, LK91]. The arrival of transactions (customers) is modelled in the `GENERATE` block. The inter-arrival time of customers is uniformly distributed over the interval 10 +/- 5 time units. The `QUEUE/DEPART` block combination collects queueing statistics of the queue formed by customers waiting for the capacity 1 resource "cashier" modelled by the `SEIZE/RELEASE` block combination. Once the cashier facility is seized, a customer is served for a time sampled from a uniform distribution over the interval 5 +/- 3 time units. This is modelled by an `ADVANCE` block. At the `TERMINATE` block, the life of transaction ends.

As shown in Figure 11, a process interaction simulation kernel employs three main data structures: the Future Event List (FEL) and the Current Event List (CEL) (the "chains" FEC and CEC in GPSS terminology) are internal to the simulator whereas the third one represents the Process Interaction model. A transaction is always present in exactly one of the two lists. A transaction data structure contains

- a unique identifier,
- a priority,
- a move-time, the time at which the transaction is scheduled (by an `ADVANCE` block for example) to the next block in the model.
- a number of transaction attributes described by the modeller ("parameters" in GPSS terminology).

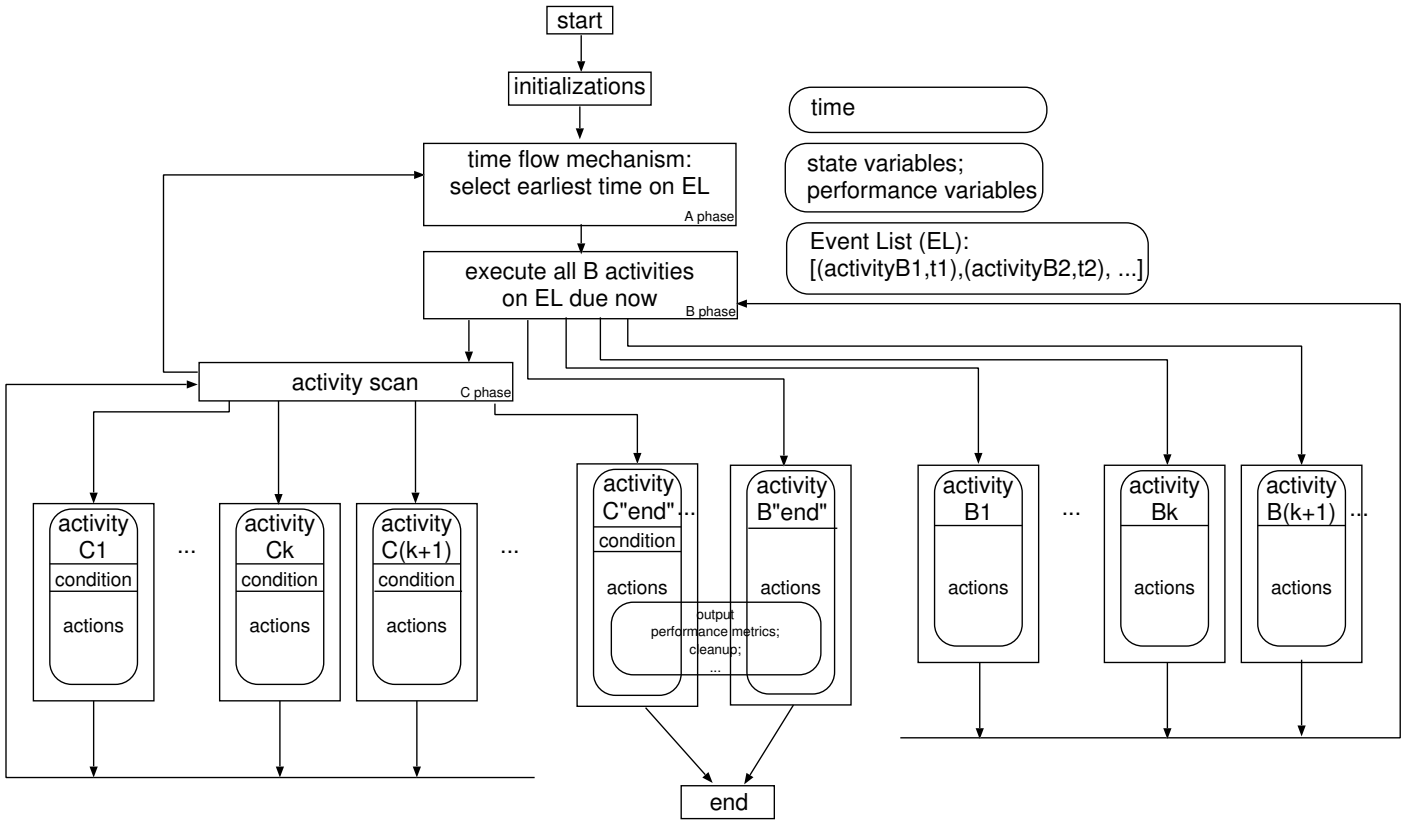

Figure 9: Three Phase Approach simulation kernel

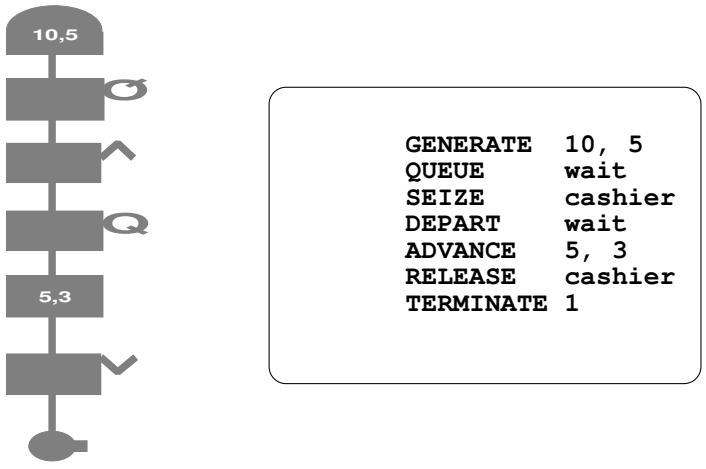

Figure 10: Process Interaction (GPSS) model of a cashier/queue system

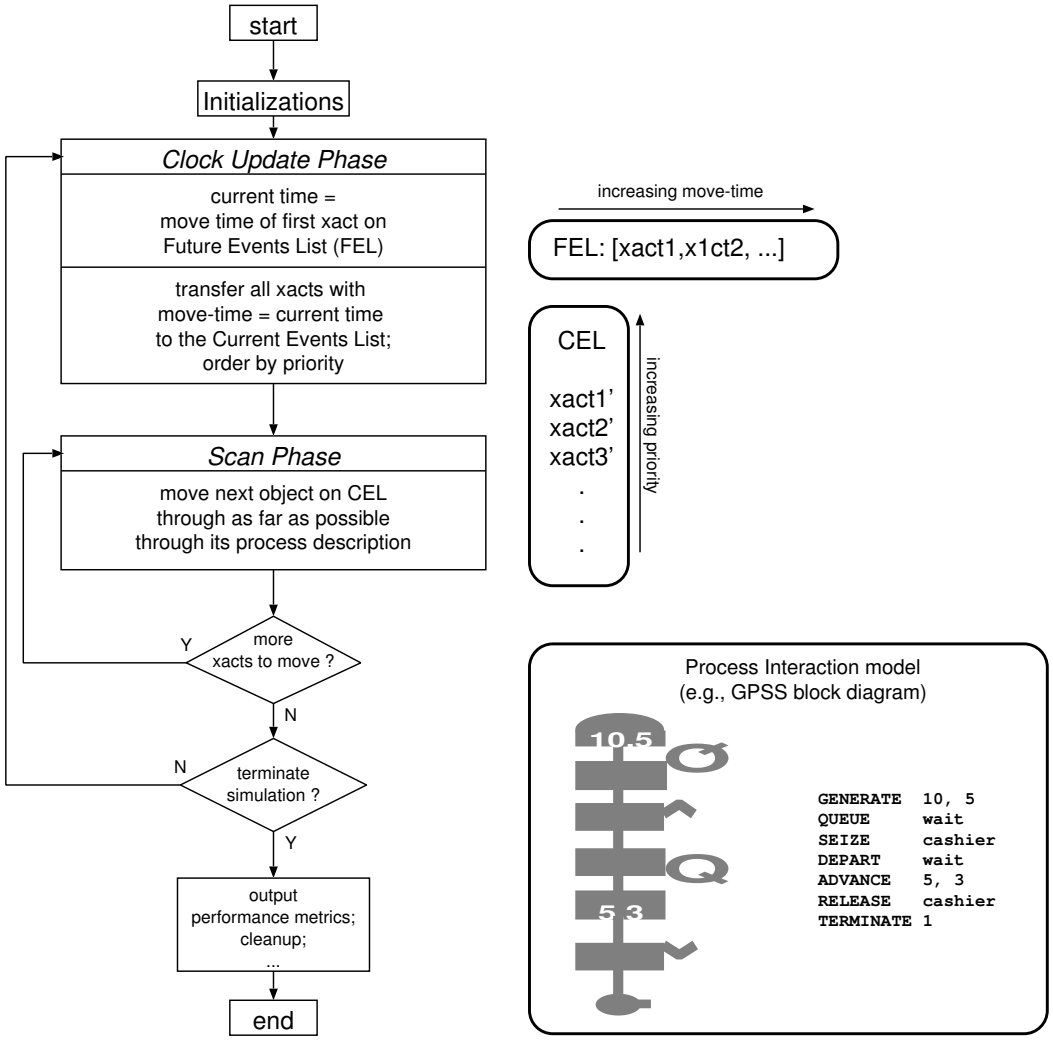

Figure 11: Process Interaction simulation kernel

The FEL is a list of transactions ordered by increasing move-time (starting from the head). For equal move-times, the order is not specified. The CEL is a list of transactions ordered by decreasing priority (starting from the head). After an initialization phase, the simulation proceeds iteratively through two phases:

1. During the *clock update phase*, the current simulation time is advanced to the move time of the first transaction on the FEL. Subsequently, all transactions on the FEL with move-time equal to the current time are moved to the CEL. On the CEL, transactions are ordered by priority.

2. During the *scan phase*, the CEL is searched from beginning (high priority transactions) till end (low priority transactions) for transactions which can be moved through the model. If a transaction is found, it is moved *as far as possible* through the model. A transaction may become *blocked* when it enters an `ADVANCE` for example. Then, the simulator schedules it to leave the block at a later time by putting it on the FEL. When a transaction reaches a `TERMINATE` block, it is destroyed (removed from the CEL). The scan phase is repeated as long as it is possible to move transactions through the model. When no more transactions can be moved, and the simulation's termination condition does not yet evaluate to true, the clock update phase is invoked again. The termination condition is often encoded as a global Termination Counter being decremented to zero or below. It is obvious that blocked transactions may reside on the CEL well beyond their move-time, waiting for some system condition.

Before completely terminating a simulation, performance metrics are output and data structures are cleaned up.

Figure 12 depicts the life of a single transaction from its creation, over its repeated migration to and from the Current

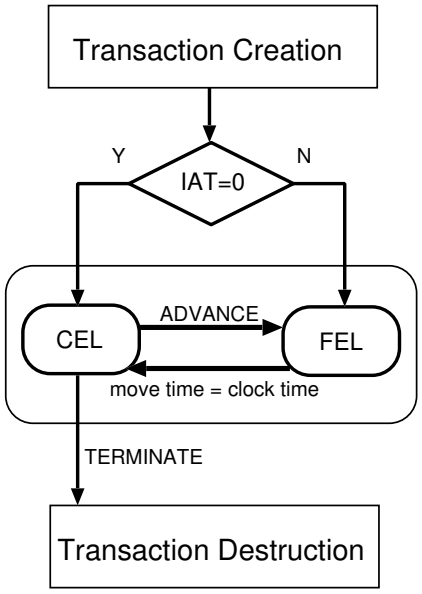

Figure 12: A transaction's life during a Process Interaction simulation

Event List and the Future Event List, until its final destruction in a `TERMINATE` block.

7 Relationships between discrete event world views

Figure 13 gives an overview of the relationships between "different" discrete formalisms. On the left hand side, formalisms are shown whose time flow mechanism is a fixed time advance. On the right hand side, formalisms with a "discrete event" time flow (clock advances to event times only) are shown. It is noted that the Activity Scanning world view really belongs under Discrete Time formalisms though it is always erroneously included with Discrete Event formalisms.

All discrete event formalisms presented are *non-modular*. Model components such as event handlers (Event Scheduling world view), activities (Activity Scanning world view) and process blocks (Process Interaction world view) are not encapsulated entities, only interacting with their environment through interfaces. Rather, they directly influence global state variables as well as other components.

The dashed arrow lines in Figure 13 denote transformation of a model described in a source formalism (start of arrow line) into that same model described in the target formalism (end of the arrow line). Original and transformed model are considered "equivalent" when they produce the same state trajectory when simulated from identical intial conditions. In the figure, all transformations are towards the DEVS formalism which is described in the next section. There, it is also shown how an Event Scheduling model may be transformed into an "equivalent" DEVS model. None of the other transformations in Figure 13 is described. A rigourous treatment of these transformations (including equivalence proofs) is future work. The essence of all these transformations is the construction of a *modular* Coupled DEVS model from the non-modular specifications. This is achieved by explicitly representing *dependencies* by means of couplings between modular components. To automate such transformations, dependency analysis needs to be automated. This approach is deemed feasible thanks to the experience with dependency analysis for continuous models (Differential Algebraic Equations) as described in a later section.

References

[BCIN98] Jerry Banks, John S. Carson II, and Barry L. Nelson. *Discrete-Event System Simulation*. Prentice Hall of India, second edition, 1998.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer Verlag, third edition, 1987.

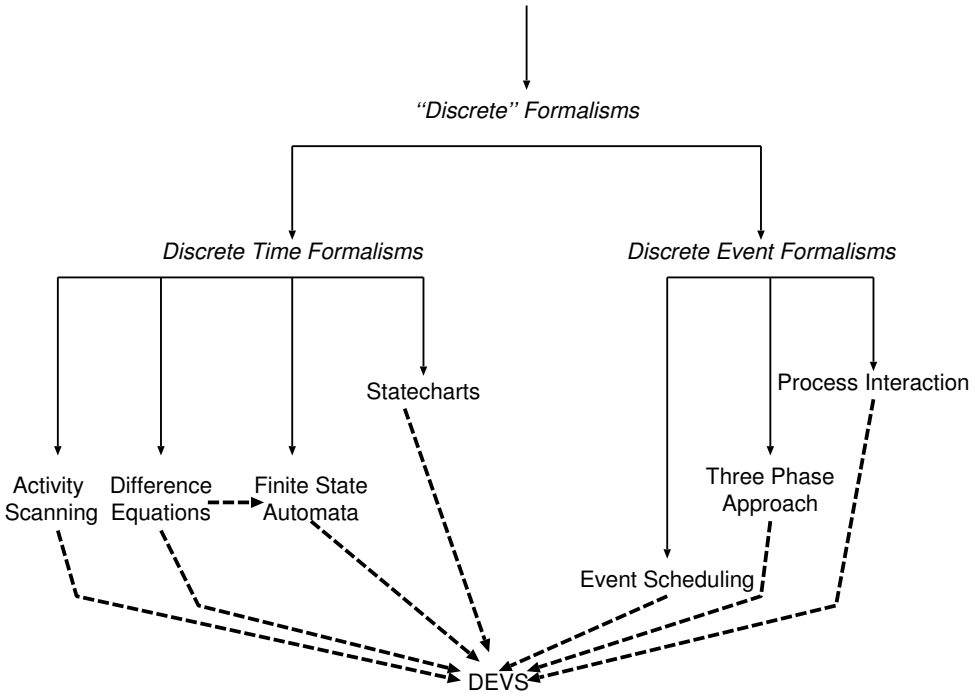

Figure 13: World Views classification

[CS92] Bruce A. Cota and Robert G. Sargent. A modification of the process interaction world view. *ACM Transactions on Modeling and Computer Simulation*, 2(2):109–129, April 1992.

[Gor96] Geoffrey Gordon. *System Simulation*. Prentice Hall of India, second edition, 1996.

[LK91] Averill M. Law and David W. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 1991.

[MB01] Pieter J. Mosterman and Gautam Biswas. A modeling and simulation methodology for hybrid dynamic physical systems. *Transactions of the Society for Computer Simulation International*, 18, 2001. (to appear).

[Nan81] Richard E. Nance. The time and state relationships in simulation modeling. *Communications of the ACM*, 24(4):173–179, April 1981.

[Sch74] Thomas J. Schriber. *Simulation Using GPSS*. Wiley, 1974.