GMD Report 26

GMD –
Forschungszentrum
Informationstechnik
GmbH

Gernot Richter

Counting Interfaces for Discrete Time Modeling

July 1998

GMD –
Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Germany
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
http://www.gmd.de

Anschrift des Verfassers/Address of the author:
Dr. Gernot Richter
Institut für Systementwurfstechnik
GMD – Forschungszentrum Informationstechnik GmbH
D-53754 Sankt Augustin
E-mail: Gernot.Richter@gmd.de

ISSN 1435-2702

Abstract

A class of interfaces is introduced that are designed to model time relations between events of a system in terms of occurrences of a reference event. The reference event generates a discrete time scale in that each occurrence produces a next graduation on the scale. It is shown how various kinds of causal connection between the "timed system" and one or several reference events or "timers" can be achieved by counting the occurrences of the reference event(s). Since reference events belong to the considered system, their scopes as timers for other events can be chosen as needed and are visible in the model. Requirements referring to a *local time* or *several independent times in the same system* can thus be specified without making assumptions which are foreign to the model and are of possibly opaque consequences for implementation. Elementary and high-level Petri nets are used for modeling counting interfaces.

Keywords: systems modeling, interfaces, time modeling, global vs local time, distributed systems, timed Petri nets

Contents

1 Introduction

Time relations between events can sometimes be specified as distances on a discrete scale generated by a reference event: each occurrence of the reference event produces a next graduation on the scale. This report proposes a quite general pattern for *counting interfaces* and their abstract implementations called *event counters* to be used for producing such scales by counting the occurrences of the reference event. An event counter models a causal connection between the reference event and the events to be related to it. Several event counters can refer to the same reference event, and a system model can contain several reference events. Since a reference event is part of the considered system, its scope as a timer for other events can be chosen as needed and made visible in the model. Requirements referring to a *local time* or *several independent times in the same system* can thus be specified without making assumptions which are foreign to the model and are of possibly opaque consequences for implementation. We use elementary and high-level Petri nets for modeling event counters, because they allow us to model concurrency and non-global time in a single conceptual framework.

The idea that "time" is basically defined by observable event occurrences is well established. In computer systems, time is in general understood as being generated by a simulator or a hardware clock. In organizational systems or in our everyday life, time scales are also defined by physical events whose regular occurrences are recorded. In system *models*, time as a system feature is not always visible. System *implementation* will usually fill the gap using proven constructs and conventions with respect to time aspects. In other cases, as for example in distributed real-time or reactive systems, the accuracy of the model can be critical since the results obtained from analyzing the formal model influence implementation decisions. In particular, a careful modeling is absolutely necessary if time relationships between events are critical and can be within the order of magnitude of the time units.

Event counters aim above all at modeling discrete time by means of causal structures. That in most time modeling problems *time dependence amounts to causal relationships*, is the central thesis of this work. Therefore, the main objective is to identify abstract conditions and events that in some way form part of any specific system with time measurements and time dependencies or with other usage of event counting. Although we use elementary and high-level Petri nets with the usual condition/event interpretation, that is, without any time extension, we do not regard time modeling as a "Petri net problem", but as a general systems modeling problem which due to its very nature can best be dealt with in terms of causal dependence and independence. Since these are the fundamental concepts of Petri nets, we chose to pursue this research on the basis of this conceptual framework.

Originally, this research was stimulated by the commonly held opinion that Petri nets cannot deal with time. Several authors considered extensions of Petri nets to be necessary to overcome their restriction to "pure" causal relationships even if this affected fundamental rules of system behavior. The following discusses in brief some approaches towards extending Petri nets by description means for modeling time relationships. It is argued that such extensions are not required or that, if suitably interpreted, they can be regarded as abbreviations with a standard Petri net semantics and not necessarily as extensions in the sense of modifications.

1.1 Related work

The earliest approaches towards time modeling in Petri nets were developed more than twenty years ago: *time Petri nets* (Merlin, 1974, Merlin and Farber, 1976) and *timed Petri nets* (Ramchandani, 1974). They are outlined, for example, in [5, 12, 4, 30, 2]. In *time Petri nets*, the transitions are assigned time intervals (minimum time which must elapse / maximum time which can elapse until the transition fires). As soon as all input tokens of a transition are available,

the countdown towards the beginning of the time interval begins on the basis of a global time scale. Within this interval the transition must occur unless it is *disabled* before the end of the interval. The occurrence itself takes no time ("the time needed to fire a transition is equal to 0"). *Time Petri nets* are defined in [3] and are used for verifying time-dependent systems.

In *timed Petri nets*, the transitions are assigned a firing duration ("every transition takes a bounded, non-zero amount of time to fire") during which the tokens are no longer available at the *input places* and not yet at the *output places*. A variant (Sifakis, 1980) is to label the places of a net with a dwell time for tokens. In [10], where this variant is called *place latency nets*, it is shown that such timed Petri nets can be interpreted in terms of elementary net systems.

A modified Petri net model for the specification of timing requirements is proposed in [14, 15, 5]. The model called *environment/relationship nets* (ER nets) is based on *coloured Petri nets* with a single global time scale. It allows one to include passing of time into the model but not to refer to global points in time. An *environment* (high-level token) can contain a standard variable called *time* (or *chronos*) whose value indicates the time at which the environment appeared at the place (timestamp of the token). In the transition expression, passing of time can be formulated by means of the variable *time* in the *environments* of the input places and output places. A *firing of a transition* is always done with duration 0.

Time modeling with *ER nets* introduces a global course of time for the entire model although the authors deny the existence of a global clock: "*Within ER nets, time is therefore a distributed concept: each environment has its local private time. There is no clock-on-the-wall whose time shows the instant reached by the global evolution of the net. The only link among the time values within the environments is that they refer to the same time scale, i.e. the same origin and the same time units; it is clear that such link is by no means a global unique time for the whole net.*" [5, p. 267]. But what else is to be understood by a global time? To count the (assumed) global time pulses individually for each transition is not a *local private time*, but rather a *local private counting* (such as the "local time" of a time zone).

Another time modeling approach for high-level nets is proposed in [32]. The *Interval Timed Coloured Petri Nets* (ITCPN) are coloured Petri nets where transitions are assigned *firing delays* (intervals) and tokens carry *timestamps*. When a transition fires it produces tokens with timestamps. The value of the timestamp is the firing time plus an arbitrary value from the firing delay (interval) of the transition. The *firing time* is given by the current enabling situation: The enabling time of a transition is defined as the maximum timestamp of the tokens to be consumed. The earliest enabling time under the current marking defines the global model time. This is the time taken as the firing time of a transition. A transition always fires at its enabling time *if* it fires. If several transitions have the same enabling time, they fire in arbitrary order.

Coloured Petri Nets (CPN) [19] support time modeling based on a global clock. In a CPN model with time simulation, a global model time is counted and recorded, and the tokens (of a time-dependent *colour set*) show a *time stamp* affixed by the simulator. *Transition delays* and *arc delays* can be defined to assign the production of tokens an individual duration by postdating the time of their availability for subsequent transitions. To enable an *occurrence element* to occur it must be both *color enabled* on account of the marking and *time enabled* on account of the current model time. Time-independent tokens are immediately available when they appear in a place. The occurrence of an occurrence element takes no time itself. As in other time modeling approaches, untimed transitions are not really time-free, but always occur immediately (zero time) when they become enabled.

In order to describe synchronization schemes for multimedia systems, *Time stream Petri nets* are proposed in [8]. Their input arcs (outgoing from places) are labeled with time intervals referring to a global time. This entails a set of new, complex firing rules based on timers local to each outgoing arc of the places.

There are still other efforts to come to terms with time in net-based system models. For instance, in [31] a relationship between time and processes is established in that time information is added to the places of the occurrence net. *Continuous Petri nets* are introduced in [6]. They associate *firing speeds* with transitions which are obtained from a timed discrete Petri net.

These and many other proposals of bringing time into Petri nets give evidence that there is a need not only to include time aspects into net models, but also to find out their time-related properties through formal analysis. Where a global timer can be assumed to underly the model, a clear semantics for the combination of net model and timer seems rather straightforward. However, in these cases it is not always clear whether it is the simulation environment or the real system that is being modeled. In other words, the question arises as to which extent the model reflects reality and how model and reality are interconnected.

A way of integrating the course of time into a system model without extending Petri net theory by an additional time concept and, in particular, without modifying the *firing rule* for Petri nets, is proposed in [25]. Via functional units called *clocks*, any part of a system model can be linked to the time pulse of a timer, other parts can remain free from any time dependence. If required, a global system time can be introduced (alone or together with local times). In the present report, this approach is taken further by providing uniform and more general interfaces along with considerably simplified and elaborated net models. In order to emphasize that time modeling is not the only, albeit the most obvious area of application we chose the term *event counters* instead of *clocks*.

1.2 Structure of the report

This report presents a quite general pattern for *counting interfaces* with application-specific protocols and a first set of self-contained and executable models that can be applied to the analysis and specification of timed systems. Section 2 describes the basic concepts and principles of this approach. Sections 3 and 4 contain elaborated models of event counters for use in typical applications. They can be used as components in system models either immediately or after some adaptation to the problem at hand. This is shown in Section 5 with examples of frequently occurring time modeling problems. Section 6 presents some conclusions.

While the basic principle of event counting can best be demonstrated with elementary Petri nets for a constant built-in counting distance (see Section 3), high-level Petri nets are preferred for the practical use of event counters in systems modeling since they make possible compact modeling of interfaces and counter readings (see Section 4). What is more, high-level Petri nets allow different variants of event counters to be modeled in a largely uniform way[1].

2 Interface design

Where parts of a given system, in the following referred to as *subsystems*, depend on each other they either interact directly or relate their behavior to occurrences of a distinguished event that is observable by all of them (e.g. the pulse of a clock, the beat of a metronome, the sunrise at a given place). Each subsystem counts the occurrences of the reference event and links its behavior with the sequence of consecutive occurrences. The term 'subsystem' is used without definition in an intuitive way, which seems acceptable in the present context.

[1] We use well-known basic concepts of Petri nets: *condition* refers to an elementary local state, *event* to an elementary local state transition, i.e., a coincident change of conditions. Conditions are modeled with S-elements or places, represented with circles. Events are modeled with T-elements that are connected with one or more places. They are represented with boxes and arrows. The marking of a place models whether the condition holds or not and, in the case of high-level nets, which conditions hold.

2.1 Clock-like interfaces

Any event can serve as reference event. From a mere modeling point of view, it is irrelevant what is taken as a reference event. This is in general not true for a technical or organizational implementation of the modeled system. There, a natural or artificial device produces the occurrences that are regarded as "ticks" for counting purposes. Two conditions which alternatingly begin and cease to hold suffice: one condition holds always after an occurrence of the reference event, the other holds after counting the occurrence and before the reference event occurs again. We want to refer to each individual occurrence of the reference event as a *pulse* (e.g., a clock tick). The two conditions are therefore called AP (*after pulse*) and BP (*before pulse*). A pulse is thus a unique, unrepeatable transition from marked to unmarked BP and from unmarked to marked AP.

If one or several subsystems are connected to the common *pulse generator* (the "implementation" of the reference event), two conditions AP_i and BP_i constitute the interface between the pulse generator and subsystem i. When the reference event occurs, all conditions BP_i ceased to hold and all conditions AP_i begin to hold. When a subsystem i completes the counting of this pulse, condition AP_i ceased to hold and condition BP_i begins to hold. Depending on the context, the interface of an individual subsystem or that of all subsystems taken together is referred to as the *pulse interface*. The left part of Figure 1 shows a Petri net that illustrates the position of the pulse interface.

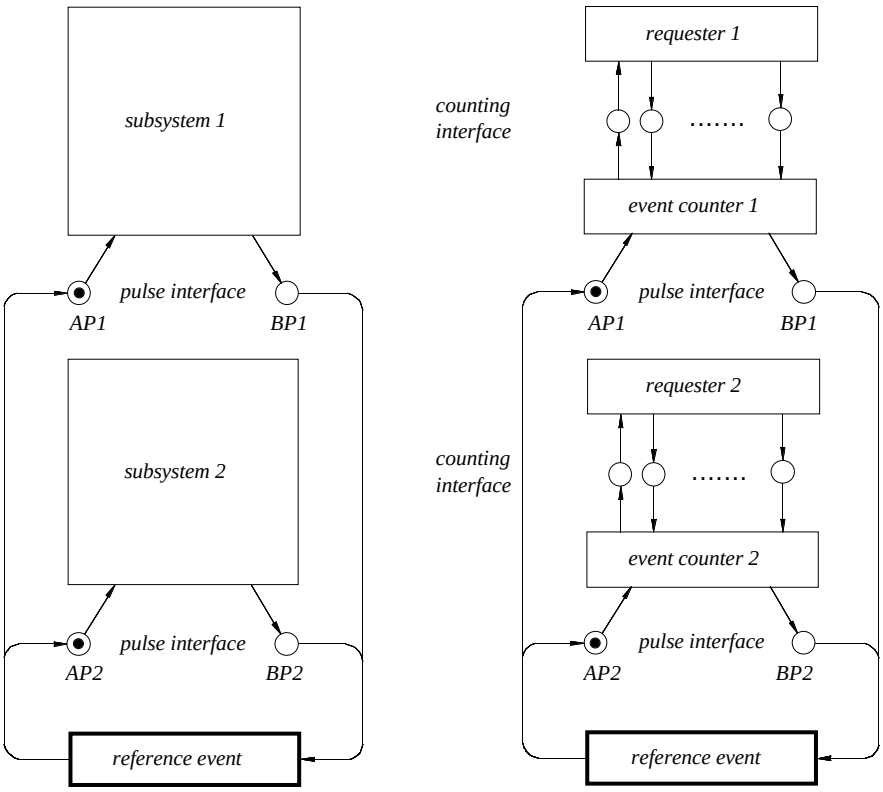

Figure 1: Interfaces for coordination via a reference event

Aiming at a systematic description of various applications and types of event counting it is useful to further distinguish the actual counting and recording from the other functions of the subsystem. A functional unit specialized in counting shall be referred to as an *event counter*. Its interface to the remaining subsystem is the *counting interface*. This subdivision is shown with the right part of Figure 1.

Thus, an event counter has two interfaces: the *pulse interface* where the interaction with the pulse generator occurs and the *counting interface* where the pulses, e.g. in a time modeling application, define "points in time" and "time distances". The remaining subsystem can be regarded as the *requester* of the counter that submits a *counting request* to the counter as the *provider*, e.g., to deliver a signal after 5 pulses from receipt of the counting request. The introduction of two interfaces has an obvious analogy in a usual clock: the swing system (oscillator, pendulum etc.) has a pulse interface to the clockwork and the latter has a display as a counting interface to the user or to a technical device whose behavior relates to the clock.

The following terminology (illustrated in Figure 2) makes it easier to talk about event counting. A seemingly obvious concept is the *interval*. It is obvious if we consider the sequence of pulses: Each pulse coincidently terminates the current interval and begins the next one. We call each individual state between two *consecutive* pulses one *pulse interval*.

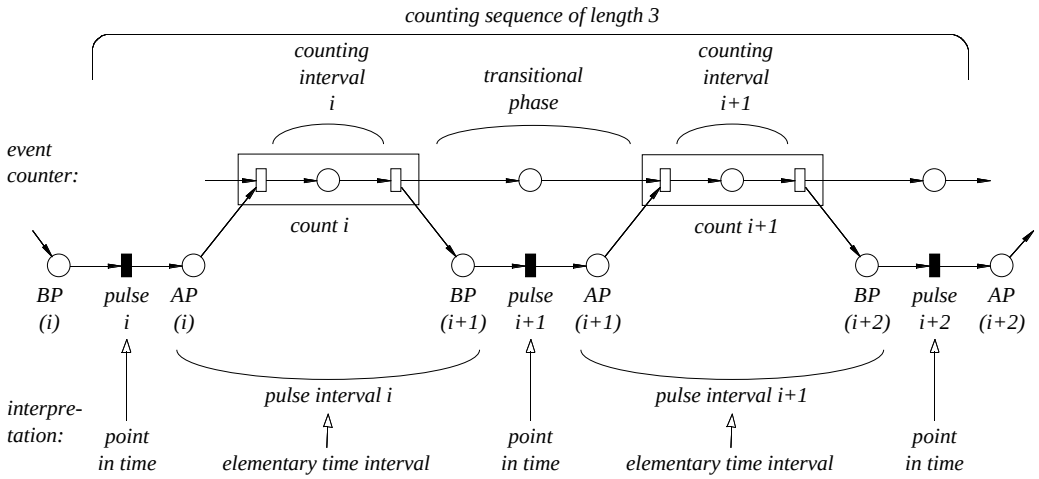

Figure 2: An occurrence graph illustrating terminology

In order for the pulses to become a means of relating different subsystems they need to be monitored by a counter. Thus it is not the pulses themselves but their being realized and enabled that is visible at the counting interface. Here, we need a notion of interval that allows us to say that something happens in the same interval as something else, or that something happens in every interval, in the current interval, etc. In these cases we refer to a *counting interval* rather than to a pulse interval.

A counting interval begins when AP ceases to hold and ends when BP begins to hold. With this definition we introduce an additional *transitional phase* between counting intervals where the "current counting interval" is undefined (an alternative view is discussed in the Conclusion). Since counting intervals are included in pulse intervals, we can sometimes ignore the difference and use the word 'interval' alone. The process running in a counter during a pulse interval is referred to as a *count*.

We adopt the following numbering rules. Pulse i begins pulse interval i and precedes counting interval i. Pulse $i+1$ succeeds counting interval i and ends pulse interval i. A *counting sequence* of length d is a sequence of d consecutive pulses recorded by the counter or, what is the same, of $d-1$ consecutive intervals. This sequence defines what is called a *counting distance* d. The pulses of a counting sequence are numbered in a way that depends on the counting mode of the event counter (see 2.2).

An obvious application of event counting is to interpret the occurrences of the reference event as the advancement of time (or of *a* local time if there are several reference events). This enables

time dependence to be modeled such as: delay by e units of time (in the sense of time intervals or, alternatively, of points in time), waking up after e units of time (alarm) unless a specific event occurs before, measuring a period of time (stopwatch) or delivering a signal regularly after e time units (chime). Faced with the variety of possible counting requests, this report can only propose some few basic constructions for event counters. If required, further counters can be derived from them or larger functional units can be composed by combining them.

A counting request normally entails to keep count of the reference event, starting with the receipt of the request, and to signal fulfilment of the counting request when a particular number of pulses has occurred. The number can be given as a distance from the counting request submission ("as soon as the reference event has occurred d-times from now on") or explicitly as a target ("as soon as pulse number t has occurred").

2.2 Counting requests

We distinguish between counting that is done for each request individually, i.e., *relatively* to the receipt of the counting request (like a stopwatch), and counting that begins from the system start, i.e., *absolutely* in a single, consecutive counting sequence (like ordinary time). In both cases, we can distinguish between *forward* counting and *backward* counting.

With relative counting, the next pulse after having received the submitted request is given the number 1 (forward counting) or d (backward counting). Counting is done individually for the request until d pulses are recorded relative to request submission. Absolute counting makes all requesters share a common scale. After each pulse, the pulse number is increased (forward counting) or decreased (backward counting) by 1. Upon starting the counter, 1 or the maximum pulse number x, respectively, is taken as the initial value of the serial pulse number.

Another distinction refers to whether one counts linearly or cyclically. When the limit of *linear counting* is reached (after pulse x or 1), the counter stops counting and (depending on the type of counter) gets ready for new requests or stops completely (no next pulse will occur or be counted, depending on the problem). For *cyclic counting*, x is at the same time the length of the cycle (individual cycle lengths are not considered). Upon transition from one cycle to the next, the pulse number changes from x to 1 or from 1 to x.

Finally, it is useful to distinguish between a counting distance and a counting target given with the counting request. A *counting distance* (e.g., a time distance) is specified as a number d. To carry out the request means to set a condition or to pass a signal to the requester as soon as d consecutive pulses are recorded. A *counting target* (e.g., a point in time) is specified as a serial number t. To carry out the request means to set a condition or to pass a signal to the requester as soon as a pulse with number t (*target pulse*) is recorded. Since pulse numbers and counting distances have limits in any real counter implementation, an upper bound x (*maximum pulse number*) is defined as a model parameter for each counter.

2.3 Interaction requirements

Where event counting is used for time modeling an issue arises that is often considered difficult or even confusing and against intuition. The reader may have noticed that there is no priority among the events: all events are regarded as of "equal rank". Even the reference event in its role as a "pacemaker" is dependent on the other events, since its occurrence must have been recorded before it occurs again. If it were not guaranteed that the event counter kept pace with the pulse sequence, the reference event could not be used for determining (measuring) or defining "occurrence distances" (e.g. a duration).

At first sight, this *mutual* dependence might seem implausible and provoke contradiction,

in particular if the reference event means the advancement of real time. However, we should realize here that a model itself is merely the *representation* of a perceived or imagined reality. It is only within a pragmatic context, that the model either becomes a documentation of observed behavior or a specification of desired or required behavior. This is also known as *descriptive* modeling (the observed reality behaves like this) versus *prescriptive* modeling (the reality to be created should or must behave like this).

To put it in concrete terms: Each piece of reality which is to fulfill a system specification with a reference event must be constructed such that the count be always completed before the next occurrence of the reference event or, equally, such that the reference event occurs only after the completion of the count. Especially for the modeling of time with event counters, the equal treatment of the events from a causal viewpoint can lead to apparent paradoxies ("time cannot be stopped"). Answers to some questions which might be asked in this context are given in [9].

In a similar way, the interaction between requester and provider (event counter) requires a "guaranteed mutual attention". If the counter can continue without being noticed by the requester, a reliable coordination between subsystems is not guaranteed in specific cases. There are situations where the requester is required to monitor every signal of the counter and to produce an explicit signal even after a "mere reading" in order to say that in this moment (i) a first count is explicitly *not enabled* or (ii) the next count is explicitly *not disabled* or (iii) the last count is explicitly *not acknowledged* (see Section 2.4). At such a moment, the requester is aware of an opportunity to place a request, to cancel a running counting process or to acknowledge execution of a request, but explicitly causes the counter to proceed with the current activity.

2.4 Elements of counting interfaces

It turns out that a general counting interface can be defined by means of nine conditions. In the diagrams, the interface elements are arranged in three groups as shown in Figure 3:
the group *F-...* ("**F**irst count ...") for request submission,
the group *N-...* ("**N**ext count ...") for request cancelation,
the group *L-...* ("**L**ast count ...") for acknowledgement of request fulfilment.

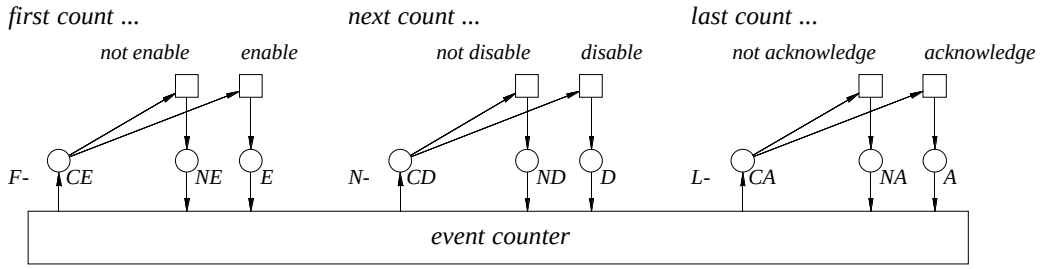

Figure 3: Basic structure of the counting interface and its use

As explained above, submission, cancelation and acknowledgement include also to explicitly *refuse* to submit, cancel or acknowledge a request. The conditions resulting from explicit *non*-submission, explicit *non*-cancelation or explicit *non*-acknowledgment have an "...-*N*..." in their names. Table 1 summarizes the nine places of a (basic) counting interface and the two places of a pulse interface.

Table 1. Interface elements of an event counter

counting interface

holding of condition ...	means ...
F-CE	**F**irst count **C**an be **E**nabled, i.e., counting process can be started
F-NE	**F**irst count was explicitly **N**ot **E**nabled, i.e., counting process was not started
F-E	**F**irst count was **E**nabled, i.e., counting process was started
N-CD	**N**ext count **C**an be **D**isabled, i.e., counting process can be discontinued
N-ND	**N**ext count was explicitly **N**ot **D**isabled, i.e., counting process was not discontinued
N-D	**N**ext count was **D**isabled, i.e., counting process was discontinued
L-CA	**L**ast count **C**an be **A**cknowledged, i.e., counting process can be finished
L-NA	**L**ast count was explicitly **N**ot **A**cknowledged, i.e., counting process was not finished
L-A	**L**ast count was **A**cknowledged, i.e., counting process was finished

pulse interface

holding of condition ...	means ...
BP	**B**efore **P**ulse, i.e., current pulse interval can be finished
AP	**A**fter **P**ulse, i.e., current pulse interval was started

The nine conditions are expected to be used as outlined in Figure 3. Complete specifications of counting interface behavior are provided in Sections 3 and 4. This is a rough description of the interface behavior: Initially, when the event counter is without a request, condition *F-CE* holds after every pulse. In this case, the requester decides whether a counting process is to be started (event *first count enable*) or not. If the requester sets condition *F-E*, a counting process (depending on the request and/or on the type of counter) starts, otherwise (condition *F-NE*) the counter returns to condition *F-CE*. In every interval the requester can decide to discontinue the execution of the request (event *next count disable*) or not. In the former case, the counter returns to condition *F-CE*. In the latter one, the counting process continues until the last pulse has been counted.

Eventually, condition *L-CA* is set and the requester decides whether to acknowledge fulfilment of the request (event *last count acknowledge*) or not. If the last count is not acknowledged, the counter sets *L-CA* after every pulse until condition *L-A* is set by the requester. Then the counter returns to *F-CE*.

Thus, a typical sequence of condition holdings at the counting interface is *F-CE,F-E,{N-CD,N-ND,}*{L-CA,L-NA,}*L-A,F-CE*. Depending on the modeling problem, a particular counting interface can differ from the outline given above. Examples of such variants (missing or merged places, additional places, other modifications) are given in the subsequent sections.

3 Elementary counting interfaces

The concept of interface includes a defined behavior, i.e., an organized interaction (protocol) between provider (event counter) and one or several requesters. The following subsections briefly describe the interface behavior of event counters for a fixed counting distance of 3 pulses that can handle a single request. They are considered the smallest illustrative examples and are modeled with *elementary* Petri nets[2]. Their functionality differs with respect to whether or not

[2]Drawing and annotation conventions are described in the Appendix.

cancelation is possible and whether or not acknowledgement can be deferred. Short descriptions of the internal S-elements (places) and T-elements of the models are listed in Table 2. The diagrams also specify (see hidden arc expr, hidden colour set expr) that all places can carry a single token only.

Table 2. S- and T-elements of elementary event counters
for 3 pulses (see Figures 4, 5, 6)

S-elements:

holding of condition ...	means ...
NDX=	*N-ND* expected in the current interval (but *N-D* possible)
WP1	waiting for pulse 1 of the counting sequence
WP2	waiting for pulse 2 of the counting sequence
WLP	waiting for the last pulse of the counting sequence
AX=	*L-A* expected in the current interval (but *L-NA* possible)
AX>	*L-A* expected in a subsequent interval (but *L-NA* possible)

T-elements:

occurrence of event ...	means ...
RNE	to react to *F-NE*
RE	to react to *F-E*
REX=	to react to *F-E* and expect confirmation in the current interval
RND=	to react to *N-ND* in the current interval
RD=	to react to *N-D* in the current interval
RP1	to react to pulse 1 of the counting sequence
RP1ND	to react to pulse 1 of the counting sequence under *N-ND*
RP1D	to react to pulse 1 of the counting sequence under *N-D*
RP2	to react to pulse 2 of the counting sequence
RP2ND	to react to pulse 2 of the counting sequence under *N-ND*
RP2D	to react to pulse 2 of the counting sequence under *N-D*
RLP	to react to the last pulse of the counting sequence
RLPND	to react to the last pulse of the counting sequence under *N-ND*
RLPD	to react to the last pulse of the counting sequence under *N-D*
RNA=	to react to *L-NA* in the current interval
RNA>	to react to *L-NA* in a subsequent interval
RA=	to react to *L-A* in the current interval
RA>	to react to *L-A* in a subsequent interval

3.1 Submitting and handling a counting request

A simple event counter is modeled in Figure 4. The marking shown in the diagram models the state immediately before a pulse and before a decision of the requester. After the first pulse and later, whenever the requester explicitly did not submit a request (transition *F-CE* → *F-NE*), the counter returns into the state *F-CE*. This is called the *idle loop* of the counter.

If the explicit non-submission of a request should not be modeled, an event that explicitly does not submit a request can be dropped at the requester, and the two places *F-CE* and *F-NE* are merged to a new place *F-CE/NE*. The resulting place models a side-condition of the event *RNE*. Thus, a conflict between counter and requester may arise in each interval—namely in the case where a request is going to be submitted. In such a case, the model does not tell us

whether the prerequisites for a request submission existing at the requester will ever lead to the submission of a request.

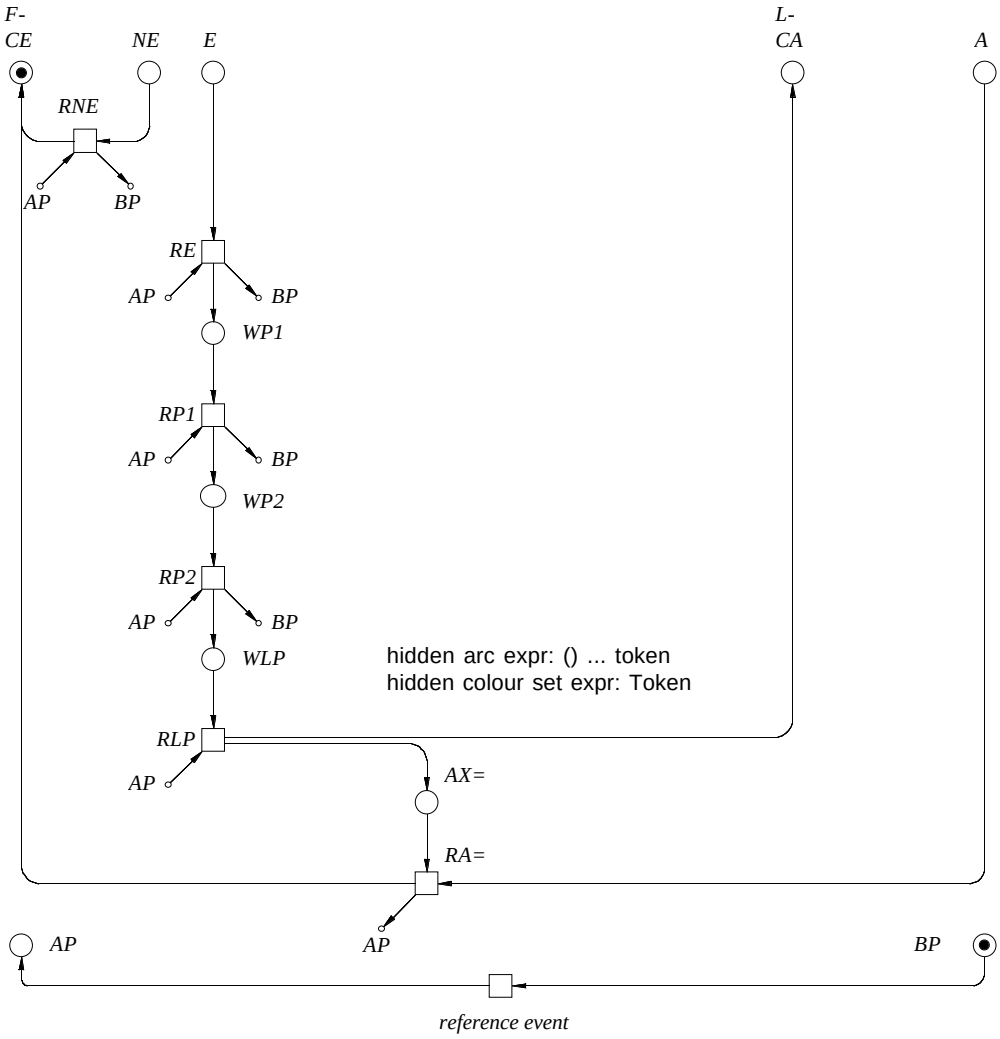

Figure 4: Event counter for 3 pulses: cancelation is not possible, acknowledgement cannot be deferred

If condition *F-CE* holds, a counting request can be submitted (transition *F-CE* $\rightarrow$ *F-E*). Notice that request submission (like non-submission) is not linked to the current state at the pulse interface. However, holding of either *F-NE* or *F-E* is required to enable the next pulse. The reaction of the counter to *F-NE* (event *RNE*) or *F-E* (event *RE*) leads to the holding of condition *BP*. If a counting request has been submitted, the first pulse in the current request processing is pending. After each pulse the counter moves on by 1 until it waits for pulse 3 as the last pulse (condition *WLP* holds).

After the last pulse, the final interaction starts via the interface elements *L-...*. The counter signals via *L-CA* that the number of pulses to be counted has occurred; the fulfilment of the request can now be acknowledged. The acknowledgment is due "immediately", i.e., in the current interval (condition *AX=* holds). Reacting to the acknowledgement (event *RA=*) the possibility of submitting another request is provided still in the current interval. For this purpose, the event counter sets *AP* and *F-CE* which is the same state as immediately after a pulse before the requester has decided between *F-NE* and *F-E*.

3.2 Canceling a counting request

If the event counter described in 3.1 is extended by the option of canceling a counting process in progress, we obtain the model shown in Figure 5. This adds a facility of discontinuing counting before the first pulse (after occurrence of event *REX*=) and "around" the subsequent pulses of a counting sequence. For this purpose, the counting interface includes an additional group of places whose names begin with "*N-*" (for *next count* ..., see Table 1). With a transition *N-CD* → *N-D* the requester cancels the current counting request, i.e., initiates the discontinuation of the counting process. The four additional events *RD*=, *RP1D*, *RP2D* and *RLPD* model the possible reactions to a cancelation[3].

The operation without submitted request (idle loop) and the submission of a request work as described in 3.1. Request handling begins with the event *REX*=. The counter reacts to *F-E* and, still in the currrent interval, requires the requester via condition *N-CD* to issue either an explicit confirmation of the counting request (*N-ND*) or a cancelation (*N-D*)—an immediate cancelation in this case. If the request is confirmed (*N-ND*), the counter proceeds (event *RND*=) by reproducing condition *N-CD* and by enabling the first pulse of the just started counting sequence (transition *AP* → *BP*). In the case of discontinuation (condition *N-D*), the counter returns (event *RD*=) to condition *F-CE*.

If an application does not require a confirmation and cancelation option before the first pulse, it suffices to discard the following components: *REX*=, *NDX*=, *RD*= and the connector from *N-ND* to *RND*=, and to add a connector from *F-E* to *RND*=, instead.

Whenever condition *N-CD* holds, the requester can confirm the continuation of request handling or require its discontinuation. Reacting to the continuation signal (*N-ND*), the counter moves on by 1 after each pulse. Even after the penultimate pulse and the counter's reaction to it (in the present example event *RP2ND*), request handling can still be discontinued (which enables event *RLPD* in the counter). Notice that the reaction to a discontinuation signal occurs independently of whether the pulse has already occurred or not.

Cancelation can be summarized as follows: A submitted request can be canceled immediately after its receipt—still before the first pulse to be counted—till immediately before its fulfilment. To model what elsewhere is called an "immediate transition" [2] one can use an event counter for 0 pulses. Obviously, for such a counter it would not make sense to allow a request to be canceled. Any cancelation allows another request to be submitted in the same interval, that is, before the first or next pulse to be originally counted for the canceled request. A cancelation is done concurrently with a pulse or, so to speak, "around" a pulse. A cancelation is responded even if request processing is just about to be terminated. That is, even then it leads to the discontinuation of request processing since the counter does not check anyhow whether the last counting pulse has already occurred or not.

Like with request submission and explicit non-submission, we can do without the explicit continuation signal (*N-ND*) at the cancelation part of the interface by merging the two places *N-CD* and *N-ND* to a single place *N-CD/ND* and thereby turning it into a side-condition of event *RND*= and the three events *R...ND*. As with condition *F-CE/NE*, a conflict between event counter and requester may arise after each pulse—namely in the case of a desired cancelation. In such a case, the model does not tell us whether an enabled cancelation at the requester will ever lead to cancelation.

[3]The subnet *REX*=, *NDX*=, *RD*= and *RND*= was developed from *RE* in Figure 4, while *RP1*, *RP2* and *RLP* of Figure 4 became *RP1ND*, *RP2ND* and *RLPND* in Figure 5.

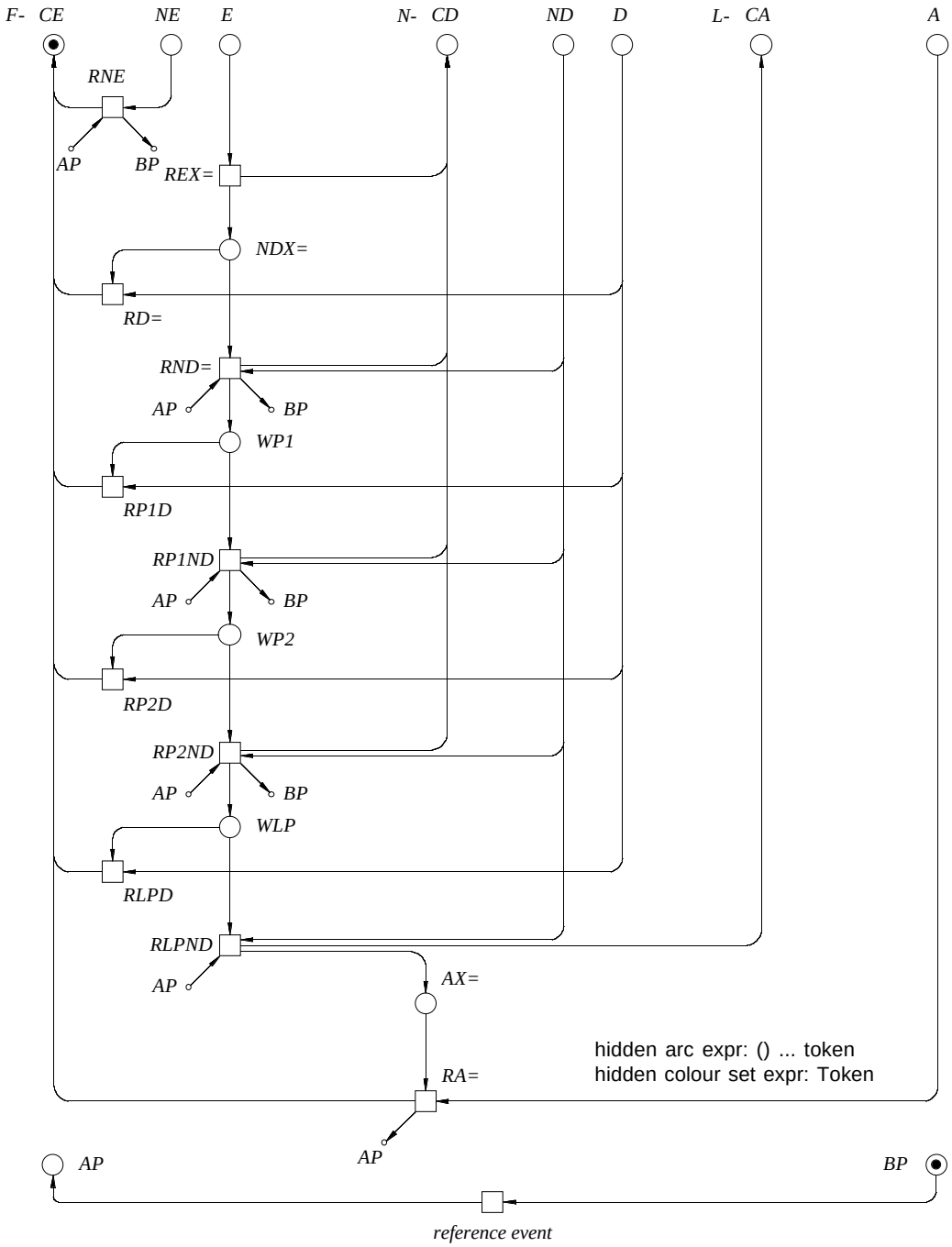

Figure 5: Event counter for 3 pulses: cancelation is possible, acknowledgement cannot be deferred

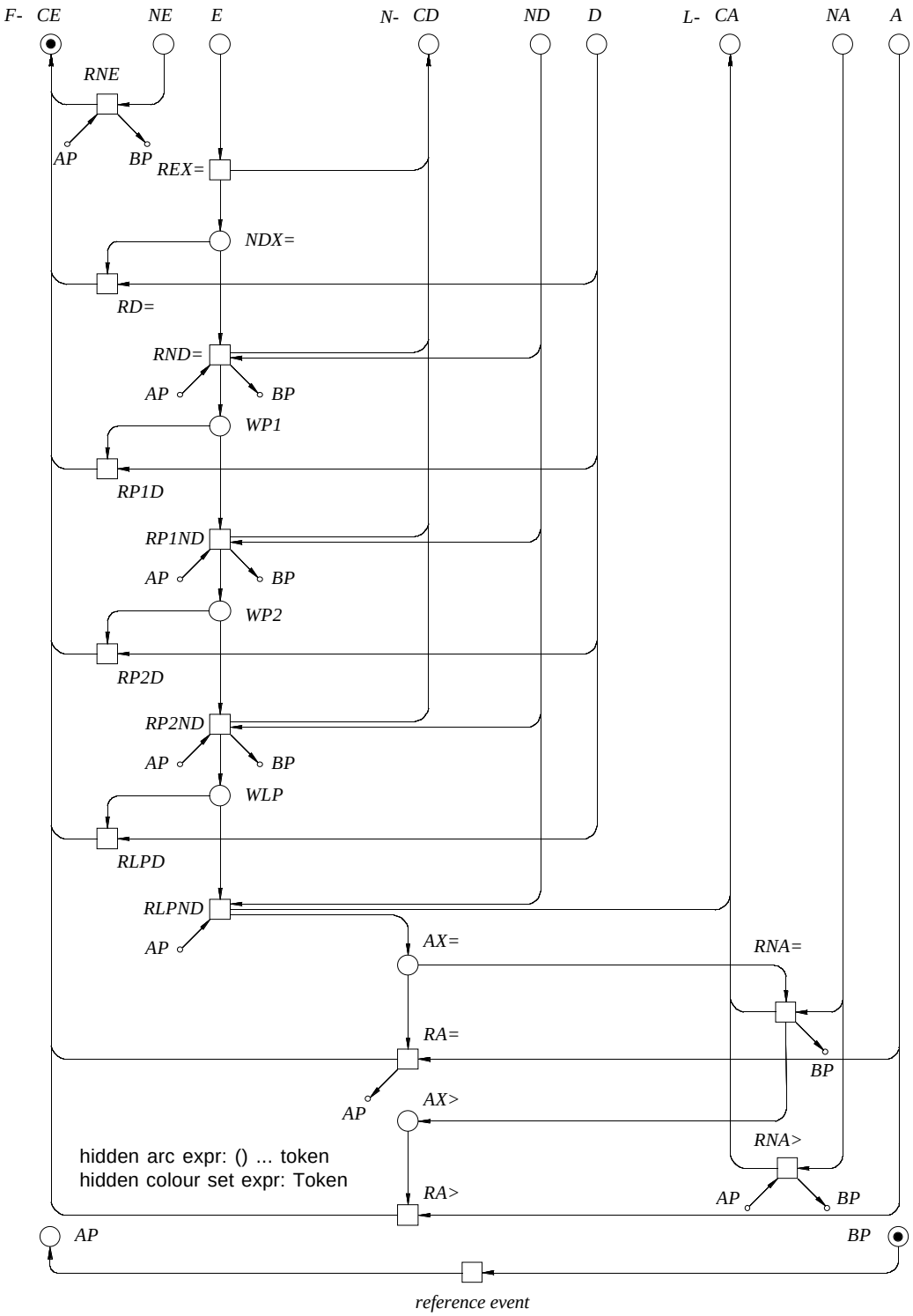

Figure 6: Event counter for 3 pulses: cancelation is possible, acknowledgement can be deferred

3.3 Deferring an acknowledgement

In specific cases ("at the earliest after ...") it might be useful to admit an acknowledgement by the requester later than immediately after the last pulse. This is achieved by the additional place *L-NA* as shown in Figure 6.

The requester is expected to refrain explicitly from giving the acknowledgment via condition *L-NA* or to give it explicitly via *L-A* in the current interval. In the latter case, event *RA=* provides the facility of another request submission still in the current interval. However, if the requester refrains explicitly from giving an acknowledgement (transition *L-CA→L-NA*), this generates a loop until acknowledgement actually occurs. Eventually, condition *L-A* is set and event *RA>* enables a further request submission. It is to be noted that, in contrast to *RA=*, it is sufficient to set only the condition *F-CE*, since the holding of condition *AP* after the last pulse has not yet been used.

If an explicit non-acknowledgment should not be modeled, like with the other groups of interface elements, the two places *L-CA* and *L-NA* can be merged to one place *L-CA/NA* and can be turned into a side-condition of events *RA=* and *RNA>* (with analog consequences as above).

For applications of event counting where cancelation or other interface elements are not used, the model can be correspondingly simplified by discarding components of the net. Such variations of the basic model are used in the examples of Section 5.

4 High-level counting interfaces

To simplify matters, we have considered so far event counters which can only handle one request with a single fixed counting distance. Practical systems modeling requires counting interfaces for handling several and different requests simultaneously. High-level Petri nets suggest themselves for modeling such counters. In terms of Petri nets, the transition from elementary event counter models to high-level ones results from a folding in two respects:

1) For each distance $d \in \{0, 1, 2, \ldots, x\}$ (x ... *maximum pulse number*), there is an elementary counter model of the same type. Folding them on top of each other, we obtain a single (strict) PrT-net for all counting distances d. Since d is now part of the request description, it appears as a parameter of the request in *F-E*.

2) Several counters folded in this way are stacked and distinguished by an identifier $q \in Q$ (q ... *request slot*). From the viewpoint of the resulting counter, q is the number of a slot for submitting a request. In the net models, the request identifier q now appears at the high-level places of the counting interface instead of the elementary places marked with a token. Q is the set of request slots, $n = |Q|$ the number of counting requests that can be handled simultaneously.

In order to keep the pulse interface in unmodified form, merging and separating the counting processes is modeled by adding an S- and a T-element before the place *BP* and after the place *AP* (see bottom of Figures 7 and 8). A condition *AC* (*after count*) models the end of the count at all request slots. As soon as all counts are terminated, an event *FANIN* sets the condition *BP*. Accordingly, when condition *AP* holds, an event *FANOUT* sets the condition *BC* (*before count*) for each request slot and thus enables each counter to continue with the next count.

Table 3 lists the internal S- and T-elements of the high-level models presented in the following sections.

Table 3. S- and T-elements of high-level event counters
for 0 to x pulses with several request slots (see Figures 7, 8, 9)

S-elements:

holding of condition ...	means ...
NDX=	*N-ND* expected in the current interval (but *N-D* possible)
WNP	waiting for the next (also last) pulse of the counting sequence
AX=	*L-A* expected in the current interval (but *L-NA* possible)
AX>	*L-A* expected in a subsequent interval (but *L-NA* possible)
AC	after count, i.e., current count ended
BC	before count, i.e., current count started
M	memory, holds the common current pulse number during a pulse

T-elements:

occurrence of event ...	means ...
RNE	to react to *F-NE*
REZ	to react to *F-E* for a distance of zero pulses
T	to transform a request with a distance into a request with a target
REGZ	to react to *F-E* for a distance of > 0 pulses
REGZX=	to react to *F-E* for a distance of > 0 pulses and expect confirmation in the current interval
REX=	to react to *F-E* and expect confirmation in the current interval
RND=	to react to *N-ND* in the current interval
RD=	to react to *N-D* in the current interval
RNP	to react to the next pulse of the counting sequence
RNPND	to react to the next pulse of the counting sequence under *N-ND*
RNPD	to react to the next (also last) pulse of the counting sequence under *N-D*
RLP	to react to the last pulse of the counting sequence
RLPND	to react to the last pulse of the counting sequence under *N-ND*
RLPD	– subsumed within *RNPD* –
RNA=	to react to *L-NA* in the current interval
RNA>	to react to *L-NA* in a subsequent interval
RA=	to react to *L-A* in the current interval
RA>	to react to *L-A* in a subsequent interval
FANIN	to enable pulse after all counts (to "fan in")
FANOUT	to enable all counts after pulse (to "fan out")

4.1 Multiple counters with complete counting interface

An event counter with $|Q|$ request slots for counting 0 to x pulses is shown in Figure 7. The model specifies a linear counter for relative forward counting with variable counting distances[4]. A request for this counter is specified by a pair (q, d): q denotes the slot occupied by the request, d is the request's counting distance ($\leq x$). While handling a request, c (*current pulse number*)

[4]The new elements (compared with the elementary counters) are as follows:
• the internal S-element *WNP* (resulting from *WP1* , *WP2* , ..., *WLP*),
• the internal T-elements *REZ* (for $d = 0$) and *REGZX=* (for $d > 0$) for request receipt (*REGZX=* corresponds to *REX=*),
• the internal T-elements *RNPD* (resulting from *RP1D*, *RP2D*, ..., *RLPD*) and *RNPND* (resulting from *RP1ND*, *RP2ND*, ..., but without *RLPND*).

The T-element *RLPND* which models the final phase of request handling (last pulse of the counting sequence has been recorded) retains its special position.

is added as a further variable, namely the request-specific current pulse number starting with 1. Two parameters appear in the place annotations: Q, the set of request slots, and P, the set of pulse numbers ($P = \{1, 2, \ldots, x\}$). The set of counting distances is $\{0\} \cup P$. A consistent initial marking is $\{AP\} \cup (\{F\text{-}NE\} \times Q)$, i.e., place *F-NE* is marked with all elements of Q (all slots idle), AP is marked (token), all other places of the counter are unmarked.

From the model shown in Figure 7, counters for absolute counting can be derived without great modifications. Figure 8 shows an example: an event counter for absolute forward cyclic counting where a central variable c for the common current pulse number is updated and used for distance or target checking (recall that each pulse is counted in the case of absolute counting even if there is no counting request at all). The place M (memory) saves the value of c from one interval to the next.

A space-saving notation had to be introduced to keep the diagram within the limits of a page: y, Y and c' should be considered as textual abbreviations for the terms (c, q), $P \times Q$ and $1 + c \cdot sign(x - c)$, respectively. The function $sign(z)$ returns $+1$ if $z > 0$, -1 if $z < 0$, 0 if $z = 0$.

The place *F-E* has been split into two places: *F-Ed* for requests with a counting distance, *F-Et* for requests with a counting target. Requests (c, q, d) with a distance are transformed into requests (c, q, t) with a target pulse number t (annotation of T had to be shifted to the right).

A consistent initial marking is $(\{F\text{-}NE\} \times \{1\} \times Q) \cup (\{BC\} \times \{1\} \times Q)$, i.e., places *F-NE* and *BC* are marked with all elements of Q (all slots idle), starting with pulse number 1. All other places of the counter are unmarked. Notice that a counting distance 0 can be specified either with $d = 0$ or with $t = c$. *REZ* includes both cases and treats them in the same way.

4.2 Multiple counters with counting interface for measurement

As an example of a variant of the basic counter (Figure 7), a high-level counter for determining (measuring) counting distances is described. Figure 9 shows such a "high-level linear stopwatch". This counter is devised for counting requests of the type "count until further notice" (the notice being the transition *N-CD*→*N-D*). The interface elements *F-CE* and *F-NE* are merged to one place *F-CE/NE*. The elements *L-CA*, *L-NA* and *L-A* are not needed.

At the place *F-E*, this model requires the start of counting *without* specifying a counting distance or counting target (simply "press the button q": transition *F-CE/NE*→*F-E*). Still in the current interval, counting is either discontinued (*N-D*) or not (*N-ND*). The requester makes this decision concurrently to each pulse. As soon as the counter reaches the maximum pulse number x, the request slot q is released again (q appears in *F-CE/NE*). Since this requires holding of *N-ND*, it is secured that the counter does not reach the value x without the requester noticing it (the pair (q, x) appears last in *N-CD* after *RNPND* occurred with $c + 1 = x$).

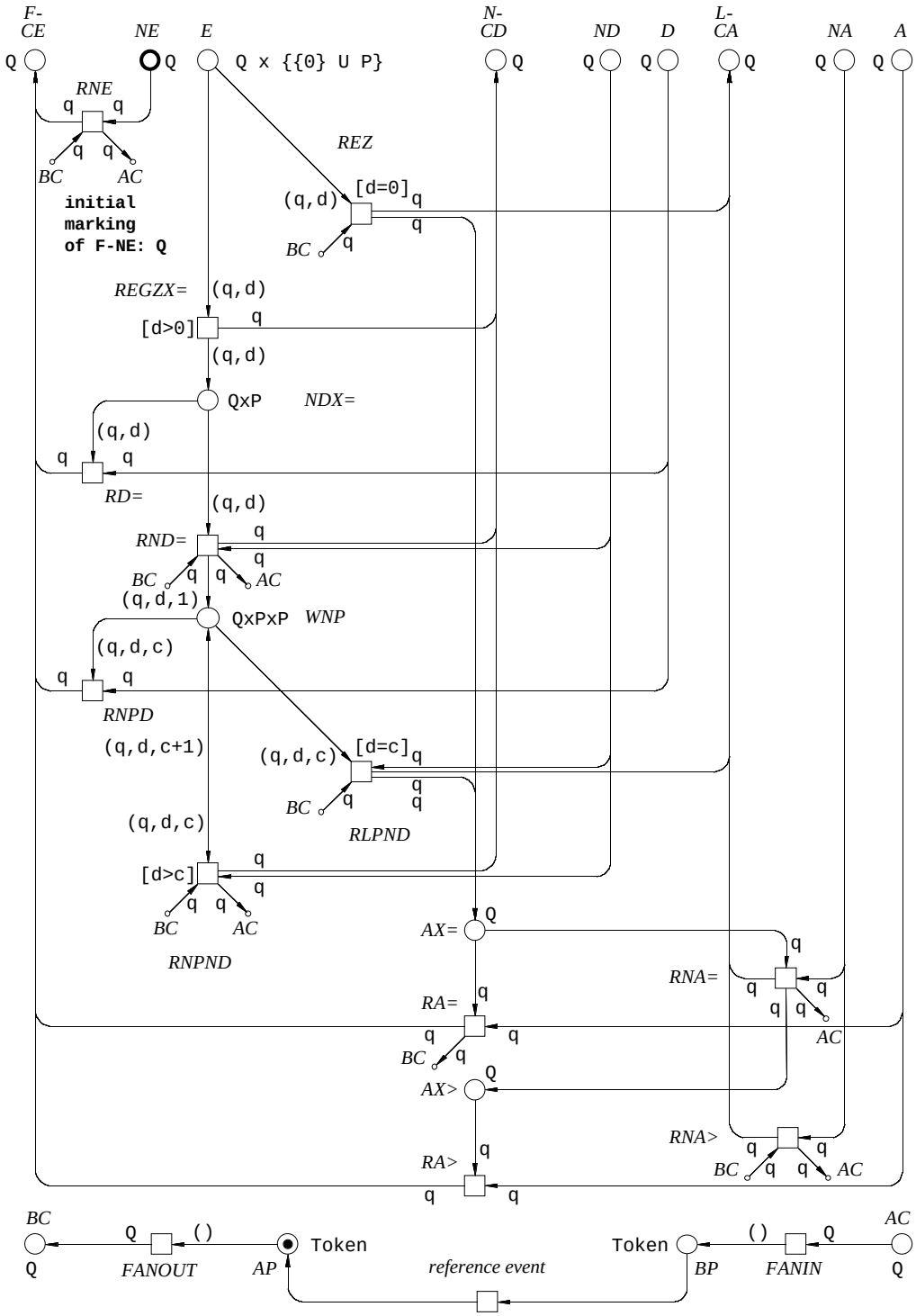

Figure 7: Event counter with $|Q|$ request slots for relative forward counting of 0 to x pulses: cancelation is possible, acknowledgement can be deferred

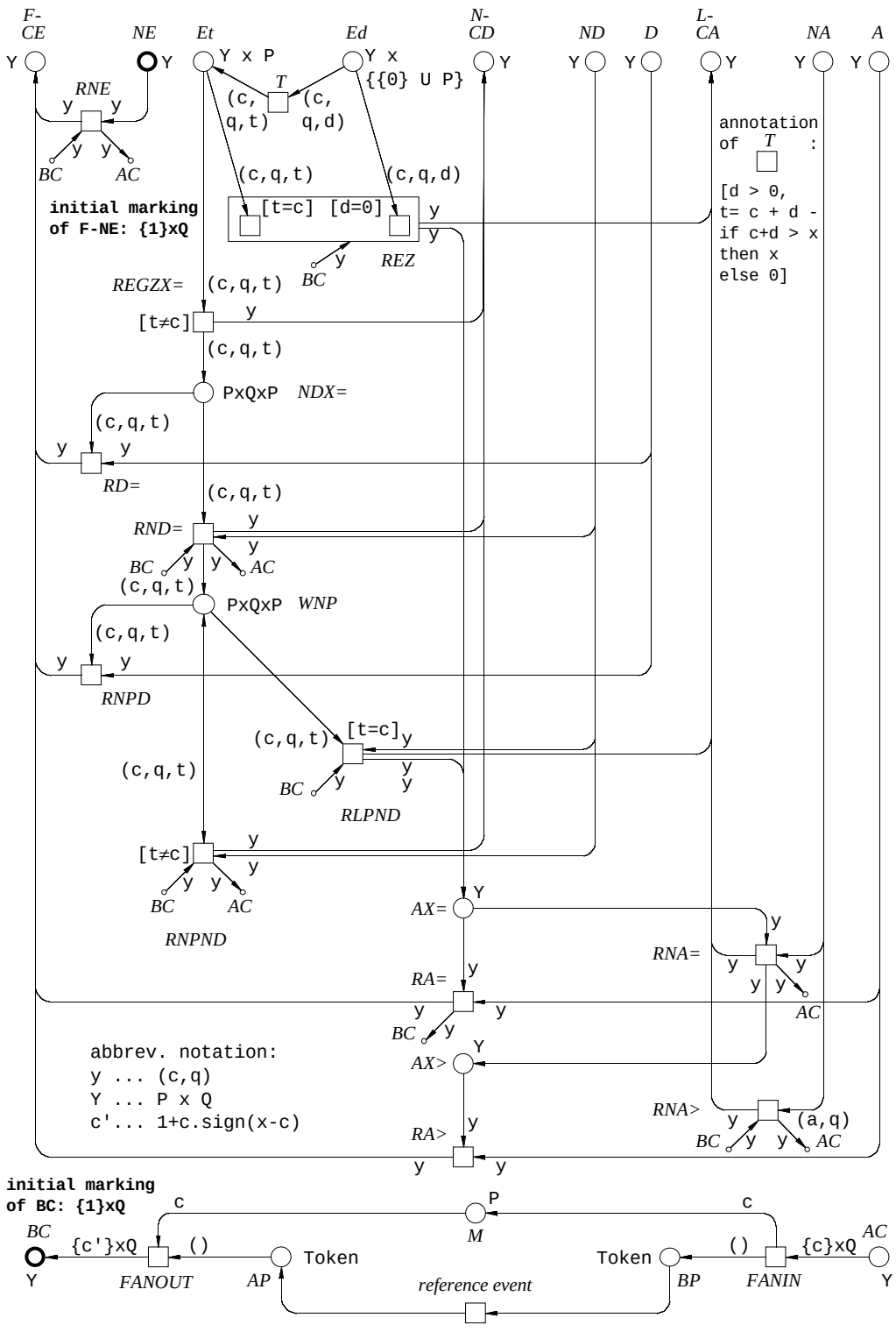

Figure 8: Event counter with $|Q|$ request slots for absolute forward cyclic counting of 0 to x pulses: cancelation is possible, acknowledgement can be deferred

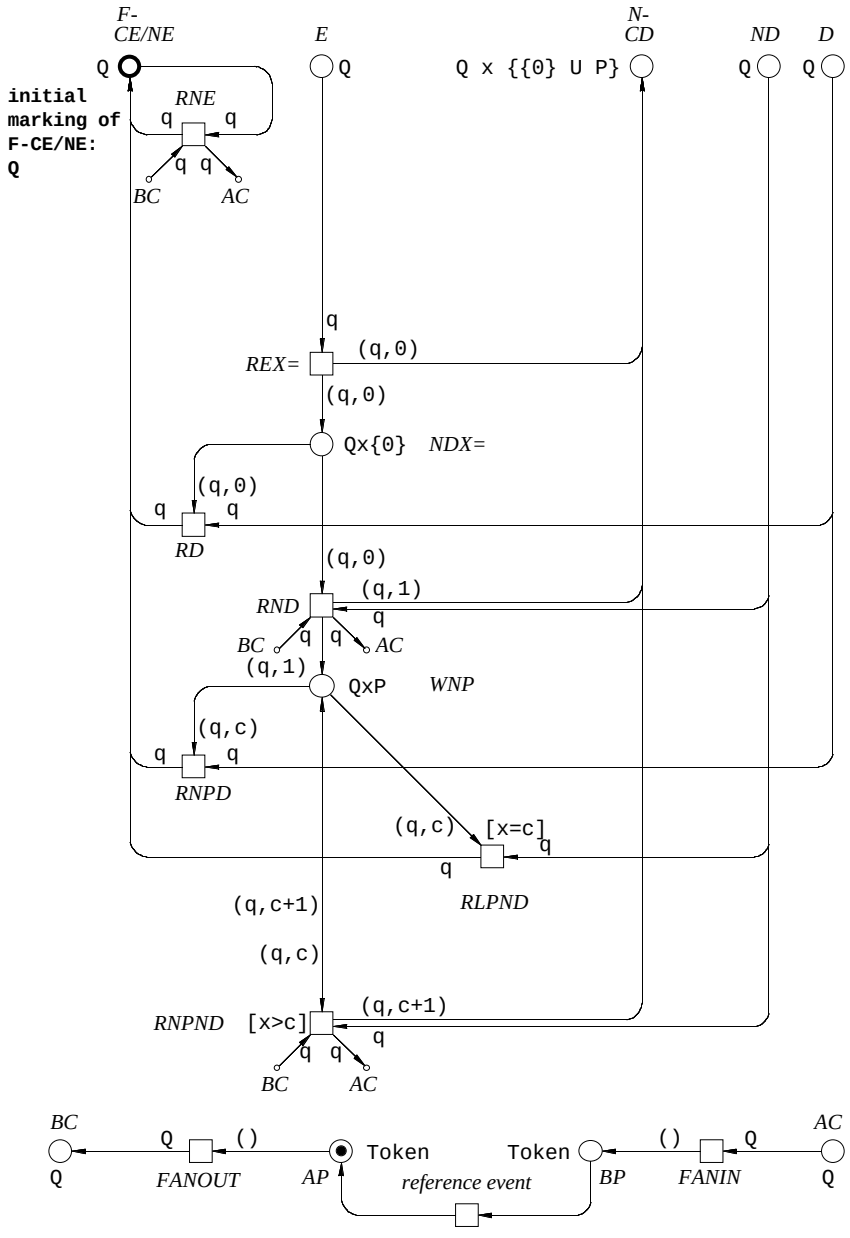

Figure 9: Event counter (stopwatch) with $|Q|$ request slots for relative forward counting of 0 to x pulses: cancelation (stopping) is expected, no acknowledgement

5 Time modeling applications

5.1 Defining the duration of a process

Figure 10 shows how to model duration of a process by means of an elementary event counter for d pulses (cf. Figure 4): exactly d pulses (i.e., $d-1$ intervals) are between the initial event A and the final event E of the considered process (requester). In this case, it is not relevant that the requesting process begins immediately after A has been enabled (therefore no such conditions are displayed in the model). What is important and thus represented in the model is that time counting starts with the beginning of the process.

After d pulses and before the $d+1$st pulse, E occurs, that is, the process terminates. In the figure, the "process" consists only of events A and E and the condition *in progress* which holds until the end of the counting distance (*delay*). In a more complex application, additional conditions and events would appear between A and E.

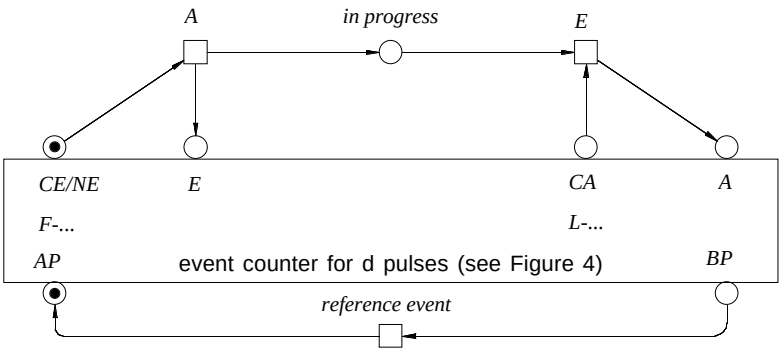

Figure 10: Process and counting terminate with lapse of time

5.2 Limiting the duration of a process

The duration of a process is limited to d pulses means that the process terminates before pulse $d+1$, counted from the initial event A. However, this does not tell everything. The following shows three different ways of specifying and monitoring deadlines.

The simplest model (Figure 11) says that pulse $d+1$ always occurs only after E. The process therefore remains within the specified period of time, there is no other possibility.

Another interpretation of period of time or deadline is shown in Figures 12 and 13. Exceeding a deadline is not excluded or, in the language of the model: E has not yet occurred (the condition *in progress* does still hold) when *L-CA* has begun to hold.

In the one model, shown in Figure 12, the above is interpreted as follows: Depending on whether E has already occurred or not, it is stated that the deadline was met or exceeded. That a conflict may arise between the latter and E, is quite natural. It can, however, not be concluded from *L-A* whether such a conflict really existed or not. In the theory of Petri nets this is called a *confusion*. The model tells that the deadline can even be met if E occurs "a long time" after *L-CA* started to hold. However, this intuitive style is illegal here: There is no short or long duration between event occurrences in the system unless with respect to the time scale defined by the pulses. Thus, it is not relevant and cannot be determined when E occurs. But it is relevant whether, after *L-CA* started to hold, *meeting* or *exceeding* is stated (in any case, pulse $d+1$ will occur only afterwards).

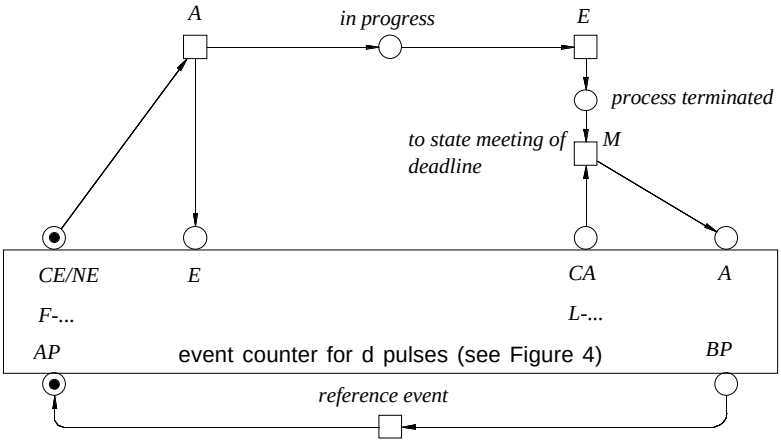

Figure 11: Process terminates with lapse of time at the latest (E), counting terminates with lapse of time, deadline cannot be exceeded

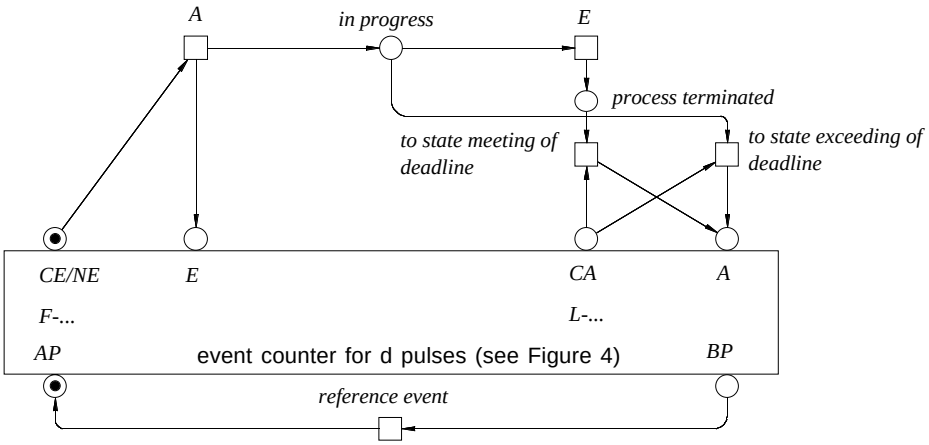

Figure 12: Process can be aborted before the end (E), counting terminates with lapse of time, deadline can be exceeded

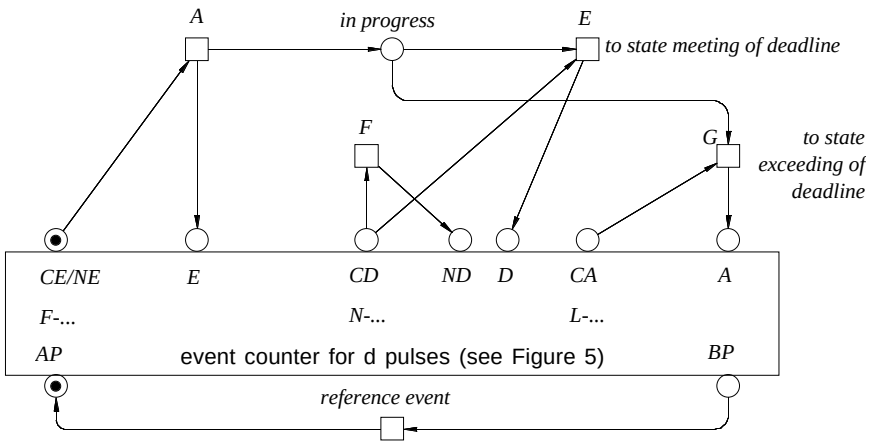

Figure 13: Process can be aborted before the end (E), counting terminates with end of process, deadline can be exceeded

The other model, shown in Figure 13, represents a variant which, like in the preceding example, does not exclude exceeding the deadline. However, if E occurs in due time, counting is not continued until the end of the period of time, but is discontinued immediately (condition *N-D*). A conflict between E and *to state exceeding of deadline* cannot arise here, since the conditions *N-CD* and *L-CA* will never hold together. With the continuation event F, counting is "kept in motion" as long as the process runs. In Figure 13, events E and F are in conflict again and again; in a real application one would expect that either E or F is enabled whenever *N-CD* holds.

In both examples, the process is terminated not later than at the deadline. A model allowing the process to terminate after having reached the deadline, though in a different way, is discussed in Section 5.4 (time-out).

5.3 Determining the duration of a process

Let an unspecified process be limited by an initial event A and a final event E. Whenever the counter is in a defined state (condition *F-CE/NE* holds), the process can begin. Like in 5.1 and 5.2, the separation of *F-CE* and *F-NE* is not important in this case since the process is analyzed from its actual beginning (event A) without considering when A was enabled.

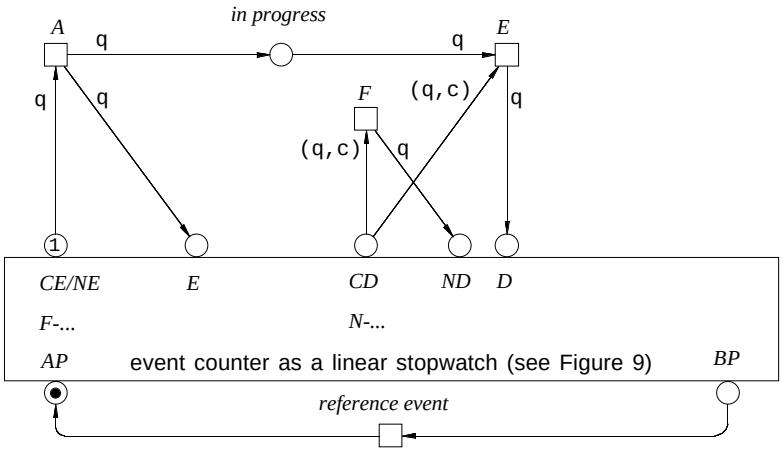

Figure 14: Counting terminates with end of process

A counter as shown in Figure 9 is used for timekeeping[5]. It counts up like a stopwatch, but linearly in this case. Unlike during the counter's idle loop, it is required during the process that the counter does not advance without the requester's involvement: the conflict between stopping (E) and going on (F) is therefore completely with the requester and not between requester and counter. Figure 14 shows the model of timekeeping with a single request slot ($q = 1$).

The counting request submitted at the beginning of the process (event A) can only be discontinued (*stopped*). The current pulse number c appears at the place *N-CD* after each pulse. It is this pulse number which specifies the duration of the process (since event A) when E occurs. Of course, process duration can only be determined if it is not larger than x. Otherwise, this system stops on the side of the requester because the counter returns to idling and consequently no pair (q, c) appears at the place *N-CD* anymore.

[5]One can also modify the counter such that the reference event begins to "tick" only after a request submission: remove *RNE* in Figure 9.

5.4 Expecting response within a period of time (time-out)

The net diagram in Figure 15 shows how to model a "time-out" problem by means of an event counter with fixed counting distance. Let the initial conditions be *AP*, *F-CE* and *idle*.

When condition *idle* holds, the system is in the basic operation (idling) in which it merely observes/monitors the course of time with the event *epidle* (*to enable pulse while idle*). By an occurrence which is not specified in detail here and which is subsumed in the event *prep*, the condition *ready* can begin to hold at any time. This enables the system to send a message to some other functional unit: event *send*, in conflict with event *epidle*, sets three conditions: *sent* (*message sent*) means that the message is on the way, *pendg* (*response pending*) means that a message has been sent to the other functional unit and that a response to it has not yet arrived, and *waitg* (*waiting for response*) means that the system is still waiting for the response in order to process it.

The behavior of the other functional unit is only represented as far as it is relevant to the considered system: It is assumed that the event *E* does not occur before the event *A*.

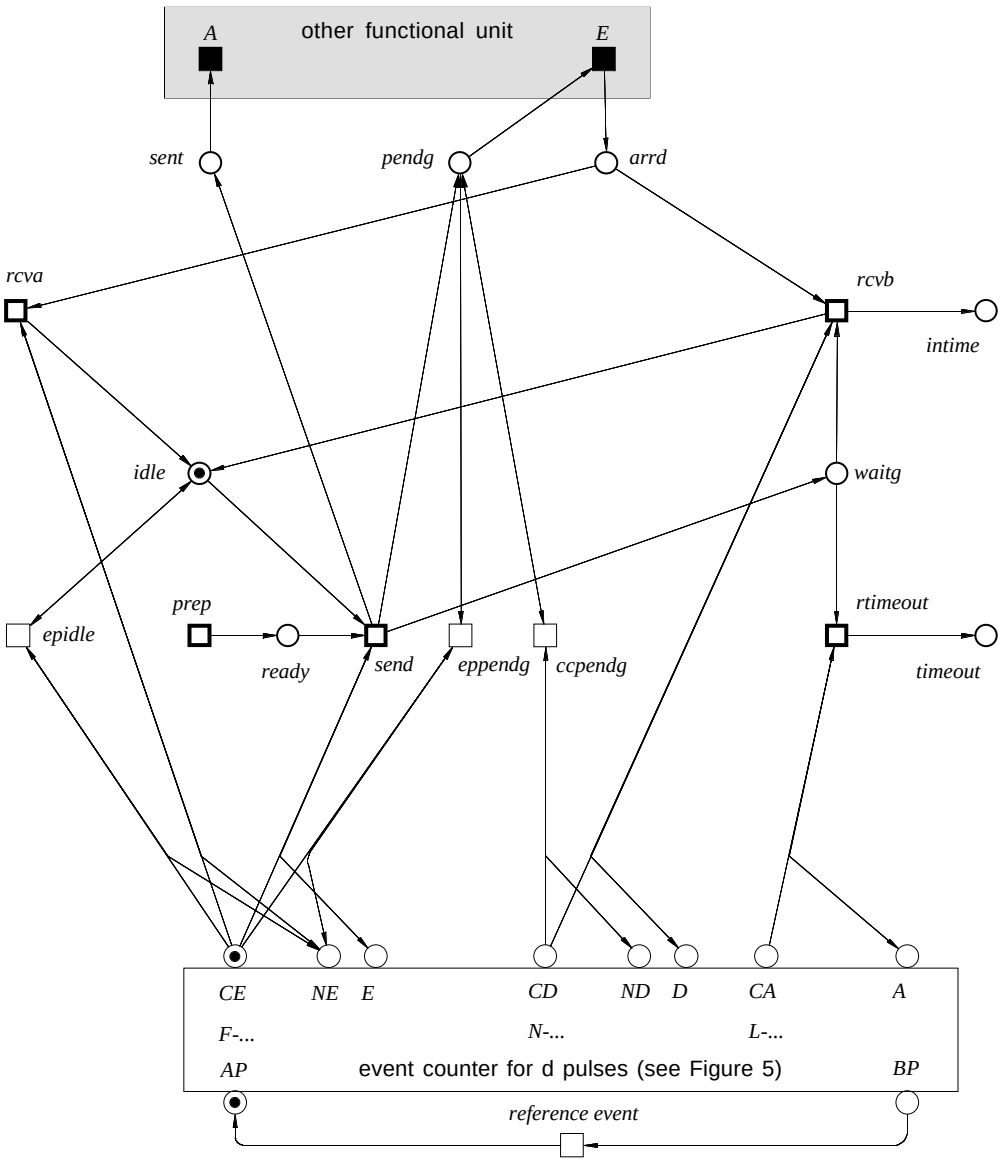

Figure 15: Waiting for response with *time-out*

Arrival of a response means that condition *arrd* (*response arrived*) begins to hold and coincidently condition *pendg* ceases to hold. How to describe in the model that the response can arrive either in time (*intime*) or too late (*timeout*)?

Through the event *send*, the system not only enters the state described above, but coincidently submits a counting request to an event counter for the counting distance d (*delay*) with cancelation option. If a response arrives before the end of the counting distance, the system states the in-time arrival of the response with event *rcvb* (*to receive response before deadline*): it cancels the counting request (*N-D*), sets the condition *intime* for some further activities of the system and returns to the basic operation (*idle*).

If the alarm rings before receiving a response (*waitg* still holding), the waking signal is confirmed immediately (*L-A*), but a later arrival of a response is still expected: *rtimeout* (*to react to time-out*) sets *timeout* as a result of exceeding the time limit, and, still in the same time interval, waiting for the late response begins with *eppendg* (*to enable pulse while response pending*). As soon as a response arrives (*pendg* ceases to hold, *arrd* begins to hold), the system reacts with the event *rcva* (*to receive response after deadline*) and restores the condition *idle*.

To realize the arrival of a response as early as possible and to identify its timeliness we must use a counter which requires a signal for continuing counting (*N-ND*) after each pulse. This "pulse release" (reflecting that there is no response in this interval either) occurs in the event *ccpendg* (*to continue counting while response pending*).

5.5 Timing with suspend/resume

As a last (and more complex) example of time depending behavior, consider the "timed PN system" depicted in Figure 16. The example is taken from [2, p. 52] (the net elements and time variables were given mnemonic names). The transitions are *timed transitions* in the terminology of [2]. A timed transition "*can be associated with a local clock or timer*". The diagram shows a left and a right subsystem. The authors describe the intended behavior as follows:

"*Transitions TL and TR ... belong to a free-choice conflict, and the firing of either of them disables the other (...). In fact, if the initial marking is that shown in the figure, the timers of TL and TR are set to their initial values, say* d_{TL} *and* d_{TR}*, with* $d_{\mathrm{TL}} < d_{\mathrm{TR}}$*; after a time* d_{TL}*, transition TL fires, and the timer of TR is stopped. Now the timer of BL is started and decremented at constant speed until it reaches the zero value. After BL fires, the conflict comprising TL and TR is enabled again. The timer of TL must be set again to an initial value (possibly, again* d_{TL}*), whereas the timer of TR can either resume from the point at which it was previously interrupted, i.e., from* $d_{\mathrm{TR}} - d_{\mathrm{TL}}$*, or be reset to a new initial value. The choice depends on the behaviour of the modelled system, ...*" Our choice is to resume counting.

The timed PN system includes the three conditions W, LX and RX. W (both subsystems waiting) holds while both timers of TL (top left) and TR (top right) are counting (down). It ceases to hold when one of the subsystems starts executing its activities, i.e., when TL or TR occurs. Execution is modeled with conditions LX (left execution) and RX (right execution). If one of them holds, the respective subsystem is executing. When time for execution has run out, transition BL (bottom left) or BR (bottom right) occurs: W begins to hold anew, the timer of TL or TR, respectively, is set to the initial value and the suspended counting process is resumed from the point at which it was previously interrupted. Notice that place W is duplicated in the diagram.

To keep the model as simple as possible, we will not consider the case where TL and TR become enabled at the same time (our model will simply stop, see below). That case would, however, not pose any new problem.

If we want to capture the time semantics of the model in a formal way, timing has to be

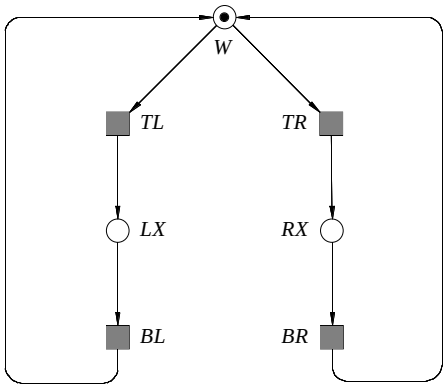

Figure 16: The timed PN system

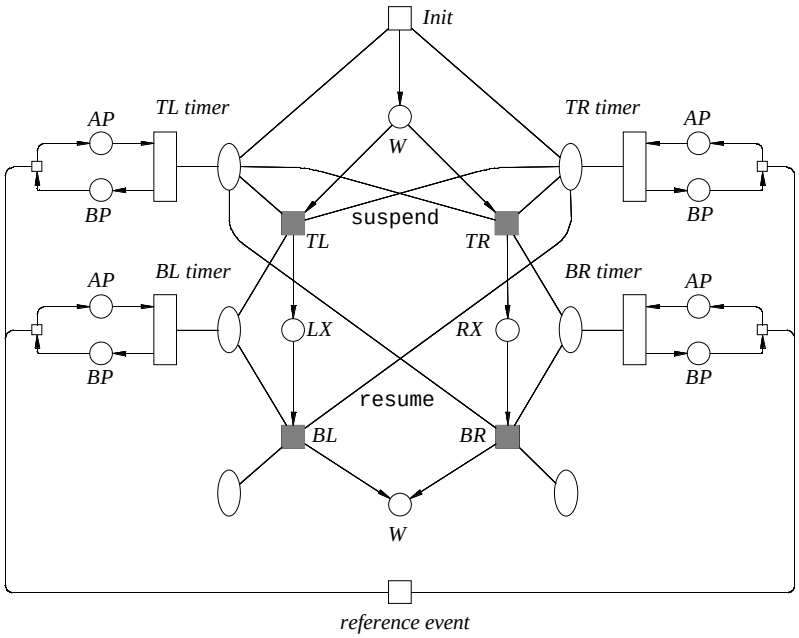

Figure 17: The timed PN system with timers and reference event (informal net, not executable)

made part of the model. With event counters this can be achieved as outlined in Figure 17. The original transitions are now connected with four event counters or *timers* that count the same pulses. Although the timers are local to the four transitions, the time is global. An oval stands for the places of a counting interface, a rectangle for an event counter. Connectors without arrow heads indicate the existence of some causal relationship in both directions. The reference event can be regarded as resulting from a *fusion* of the four transitions from BP_i to AP_i, $i = 1, 2, 3, 4$ (subscripts omitted in the diagrams). They form the component events (small squares) whose coincident occurrences make up a *pulse*. This is only a graphical notation that aims at reducing the number of lines in the diagram. An additional transition *Init* is needed to set the timers of TL and TR the very first time.

The connectors that cross the diagram indicate the interaction with the T-timer of the other subsystem:

- *suspend* ... lines between TL and the TR timer and between TR and the TL timer,
- *resume* ... lines between BL and the TR timer and between BR and the TL timer.

The exact specification of the timed behavior requires a straightforward modification of the counting interface of the *TL* and *TR* timer to allow for suspend/resume as described below.

The four timers can be modeled with event counters for fixed counting distances, starting from Figure 6. *N-D* can be omitted, there is no need for it. The timers of *BL* and *BR* need no extension, those of *TL* and *TR* receive a modified "Next count"-part (*N-...*) as shown in Figure 18. The features of the new *N*-part are listed in Table 1a.

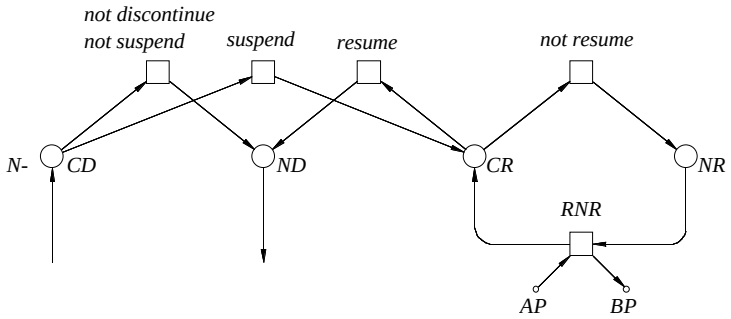

Figure 18: *N*-part of the counting interface with suspend/resume option (with additional T-element)

Table 1a. Net elements of the modified "Next count"-part for suspend/resume

holding of condition/ occurrence of event ...	means ...
	Next count, i.e., counting process ...
N-CD	... **C**an be definitely or temporarily **D**iscontinued, i.e., can be aborted or suspended
N-ND	... was explicitly **N**ot **D**iscontinued or is no longer temporarily discontinued, i.e., was resumed
N-CR	... **C**an be **R**esumed
N-NR	... was explicitly **N**ot **R**esumed
RNR	to react to *not resumed* (*N-NR*) in the current interval

The system represented in Figure 16 and described in the quoted text can now be modeled with standard constructs of Petri nets. Figure 19 shows the resulting model. Due to lack of space the timers are represented without pulse interfaces and with their counting interfaces only, prefixes *F-*, *N-* and *L-* are omitted. Several places of the counting interfaces are graphically duplicated to reveal the structure of the model: places *F-CE*, *F-NE* and *F-E* of the timers for *TL* and *TR* (two of them are triplicated), and places *F-CE* and *F-NE* of the timers for *BL* and *BR*.

To avoid too much crossing lines, the places of the two extended *N*-interfaces with suspend/resume option (Figure 18) are arranged in vertical order and on the "wrong" side: they belong to the *T*-timer of the *other* subsystem, i.e., to the timer on the opposite side of the diagram. Interface elements of the *B*-timers are also drawn separately, but on the same side of the diagram.

Summarizing, the interfaces of the timers for *TL* and *TR* comprise the following places:

F-CE, *F-NE*, *F-E*

N-CD, *N-ND*, *N-CR*, *N-NR*
L-CA, *L-A*.

The timers for *BL* and *BR* have the same *F*- and *L*-interfaces, but no suspend/resume elements (*N-CR*, *N-NR*) at their *N*-interfaces.

The counters are expected to be chosen according to the time distances $d_{\mathrm{TR}}, d_{\mathrm{TL}}, d_{\mathrm{BL}}$ and d_{BR}. What is important is that no pulse is missed by the timers and no pulse occurs without being noticed at each counting interface.

Initially, eight conditions hold: the four conditions *N-CE*$_i$ and the four conditions AP_i. *Init* is the only enabled event, all timers stand idle (no pulse occurs). The first occurrence of *Init* triggers the timed process of the two subsystems.

After several loops it will happen that *TL* and *TR* become enabled in the same time interval. The model will then stop in the following state (seven conditions hold): *W*, *L-CA*$_i$ of the *TL* and the *TR* timer, *N-CE*$_i$ and AP_i of the *BL* and the *BR* timer. Handling this situation poses no new problem and has therefore been omitted (both subsystems execute sequentially in arbitrary order, the later one of *BL* and *BR* sets *W* again).

Events *NW*, *NLX* and *NRX* result from fusing the indicated component events (coincident occurrence). They continue the four counting processes depending on the current situation in the system:

NW ... transition to the next waiting interval, *W* begins to hold,
NLX ... transition to the next execution interval of left subsystem, *LX* begins to hold,
NRX ... transition to the next execution interval of right subsystem, *RX* begins to hold.

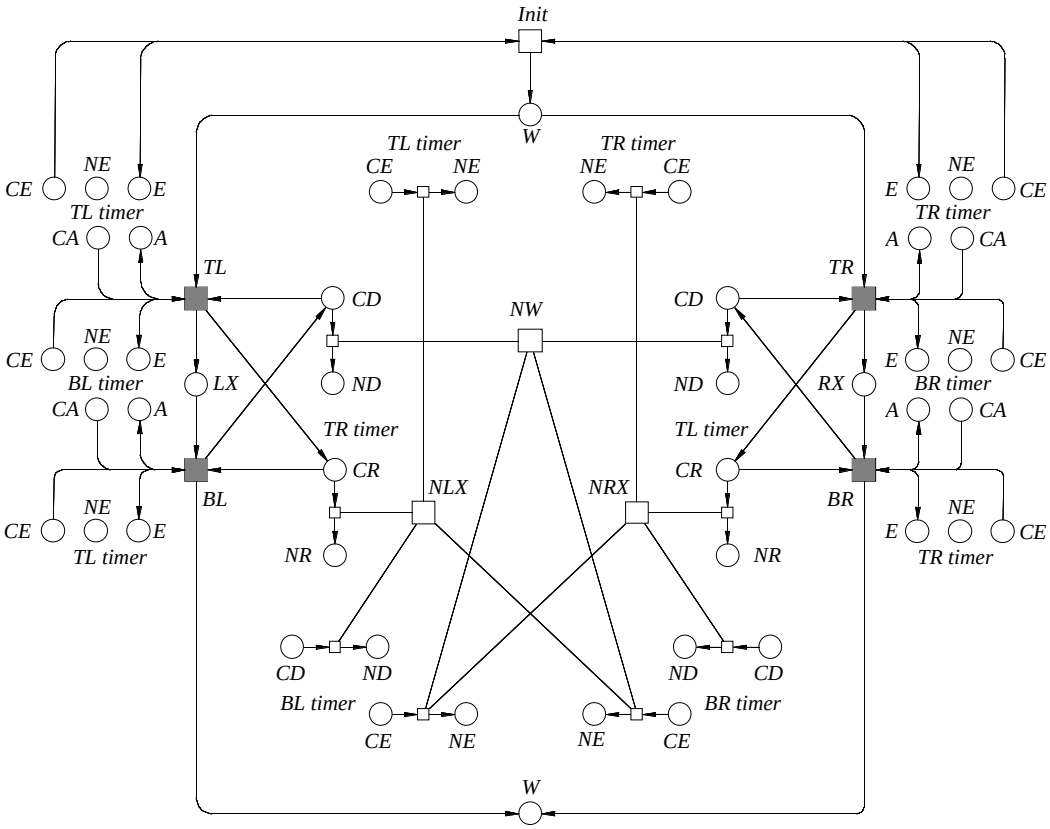

Figure 19: Explicit representation of timing semantics of the timed PN system by means of four event counters for fixed counting distances

6 Conclusion

Event counters support the specification of time requirements such as used in technical systems, but also in organizational systems with clocks, calendars, periods of time, deadlines, expenditure of time and so on. Where such requirements actually exist, it is advantageous to integrate them into the initial model rather than adding them in a later step to the "actual" causal structure of the system. This is even true if time requirements refer to a real time that cannot be influenced. However, it seems to be still a matter for dispute whether timing and functional requirements should be joined or separated. For example, in [22] it is argued that timing requirements should be included where the functional behavior is specified since they are inextricably connected. In [33], by contrast, timing constraints are maintained separately from the rest of the system specification although they are used to express restrictions on when an event takes place, on the order in which events occur, or on the separation (in time) between successive events.

Event counters allow the system modeler to refer to occurrences of a reference event without reducing concurrency more than is actually required by the problem at hand. This addresses the issue of *global* vs *local* time. In a sense, such a wording obscures the difference between the scope of a *reference event* and the scope of an *event counter*. A system can use a single global reference event or several (more or less) local reference events. Given a global reference event it can be counted globally with a single counter or locally with several counters working independently of each other. Although *local counting* of a global reference event is sometimes referred to as "local time" or "local clock", it is not a local time in the strict sense, because it is the same event that drives the counters distributed over the system. An example of locally counting a global reference event is the civil time based on the astronomical time GMT (Greenwich Mean Time) or on the modern reference time UTC (Universal Coordinated Time) derived from the International Atomic Time (see e.g. [29, Chap. 3]).

There are applications where several asynchronously interacting subsystems have individual reference events for their internal synchronization. Each of the reference events defines an independent time for the *system region* to which it is connected. If, in a system with distributed time based on several independent regional clocks, time relationships are of actual relevance, time modeling with event counters may help to find constellations in region overlaps that could lead to undesirable system behavior.

Even real-time or reactive systems have "time-free" areas, i.e., system parts without time specifications. As not everything occurring in a system with time dependencies has to be linked to a time scale, the system model should also specify where a linkage to time does *not* exist. Event counters allow time-free areas to be integrated naturally into the same system model together with time related areas.

In this report, we confined the variety of event counters to the features that are necessary for handling typical counting requests. One can easily extend the counters to variants displaying more information at the interface, e.g., the current pulse number. Moreover, they are suitable as basic models for special counters or can be combined to display larger counting units, such as second – minute – hour – day – week – month – etc. in a time modeling application.

Event counters and their variants can serve for defining a precise semantics of time extensions for Petri nets or of language constructs for time relationships. Such constructs are, for example, proposed in [13, 17, 23, 27], or in the context of *synchronous programming* [16]. The latter has become a new design paradigm for programming critical real-time systems. In these systems, all reactions must stabilize before the next tick of a global clock. In synchronous programming, the points of the "real" physical time are considered as events which have no privileged nature compared with events occurring inside the program [17]. This view is shared in the present report as far as time *modeling* is concerned. It is generalized in that "several real times" can be

included in a single system model.

Examples of time modeling concepts and net constructs whose exact meaning could be analyzed with event counters include: weak and strong time model [15], various firing rules in [8], firing delays [32], time dependent predicates [12], minimum/maximum timing constraints and durational timing constraints [30].

Where Petri nets are used to model timed systems, event counters suggest themselves as model components. Since they are modeled as Petri net modules with uniform interfaces, they can be integrated directly into the net model of an overall system. This may be particularly instrumental in applications where the assumption of a global time is not suitable because its global implementation cannot be guaranteed or because bounding the drift between local times, called the *clock skew* in [29], is just the problem. An example are the local times at the nodes of a distributed real-time system, which are kept in parallel by clock synchronization [21]. But also in the case of systems with global time, time modeling using event counters can be useful if several time counts (e.g., different calendars [11]) are considered and interrelated simultaneously.

An alternative is to use event counters to define more compact representations such as parameterized net elements (as shown in Section 5) or expressive symbols for "duration", "deadline", "waiting time" etc. The counters introduced in this report are examples of building blocks that describe time dependencies just in the way in which they are or should be effective in an implementation.

A more theoretical issue in the context of time modeling is limits to the precision of time measurement. The definition of *counting interval* used in this report entails a transitional phase between counting intervals where exact time reading is not possible because it is not defined (see Figure 2). This is like trying to read the time while the hand of the clock is moving. An alternative way to define the concept of *counting interval* could be to consider the transitional phase either the first or the last part of the such a counting interval. This would result in a sequence of counting intervals with coincident changes from one interval to the next. With such a definition a time interval i is always between pulses i and $i+2$. That is, we have a limit as to the precision of time reading: An event occurring in interval i can only be positioned between pulses i and $i+2$, but not relative to pulse $i+1$. This leads us to fundamental questions of measurement and scales as dealt with in [28, 24].

Finally, event counters in system modeling are not restricted to cases where the reference events are seen as generators of points of time. An event counter can also be used, for instance, as a kind of "gear box" with a "counting transmission" placed between subsystems. With such a mechanism events in the one subsystem could temporarily or permanently be coupled to events of the other subsystem such that a given occurrence pattern or a specified tolerance with respect to advance and delay in either subsystem is guaranteed.

Acknowledgement

The author would like to thank Reiner Durchholz for his insightful comments and constructive suggestions on earlier versions of the report and Hartmann Genrich who helped to simulate the timed PN system (Figure 19) with Design/CPN.

References

[1] M. Ajmone Marsan, editor. *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, Berlin Heidelberg, 1993. Springer-Verlag. 591 pages.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Parallel Computing. Wiley, Chichester, 1995. 301 pages.

[3] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, March 1991.

[4] G. Bucci and E. Vicario. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE Transactions on Software Engineering*, 21(12):969–992, December 1995.

[5] G. Buonanno, S. Morasca, M. Pezzè, K. Portman, and D. Sciuto. A new timed Petri net model for hardware representation. In D. Borrione and R. Waxman, editors, *Computer Hardware Description Languages and their Applications*, pages 261–280, Marseille, April 1991. IFIP WG10.2. (Participants Edition).

[6] R. David and H. Alla. Autonomous and timed continuous Petri nets. In Rozenberg [26], pages 71–90.

[7] G. de Michelis and M. Diaz, editors. *Application and Theory of Petri Nets 1995*, volume 935 of *Lecture Notes in Computer Science*, Berlin Heidelberg, 1995. Springer-Verlag. 511 pages.

[8] M. Diaz and P. Sénac. Time stream Petri nets: A model for timed multimedia information. In R. Valette, editor, *Application and Theory of Petri Nets 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 219–238, Berlin Heidelberg, 1994. Springer-Verlag.

[9] R. Durchholz. Causality, time, and deadlines. *Data & Knowledge Engineering*, 6:469–477, 1991.

[10] R. Durchholz. Latency time modelling with elementary Petri Nets. In *Proceedings of International Workshop on Discrete Event Systems WODES'96*, pages 82–87, Edinburgh, August 1996. IEE, IEE.

[11] C. E. Dyreson and R. T. Snodgrass. Timestamp semantics and representation. *Information Systems*, 18(3):143–166, 1993.

[12] M. Felder, D. Mandrioli, and A. Morzenti. Proving properties of real-time systems through logical specifications and Petri net models. *IEEE Transactions on Software Engineering*, 20(2):127–141, February 1994.

[13] R. Gerber and I. Lee. A layered approach to automating the verification of real-time systems. In Kemmerer and Ghezzi [20], pages 768–784.

[14] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A general way to put time in Petri nets. *ACM SIGSOFT Engineering Notes*, 14(3):60–67, May 1989.

[15] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level Petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, February 1991.

[16] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Real-Time Systems. Kluwer, Dordrecht, 1993.

[17] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. In Kemmerer and Ghezzi [20], pages 785–793.

[18] C. A. Heuser and G. Richter. Constructs for modeling information systems with Petri nets. In K. Jensen, editor, *Application and Theory of Petri Nets 1992*, volume 616 of *Lecture Notes in Computer Science*, pages 224–243. Springer-Verlag, Berlin Heidelberg, 1992.

[19] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg, 1992. 234 pages.

[20] R. A. Kemmerer and C. Ghezzi, editors. *Special Issue: Specification and Analysis of Real-Time Systems*, volume 18 (9) of *IEEE Transactions on Software Engineering*, September 1992.

[21] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

[22] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[23] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. In Kemmerer and Ghezzi [20], pages 794–804.

[24] C. A. Petri. Nets, time and space. *Theoretical Computer Science*, 153:3–48, 1996.

[25] G. Richter. Clocks and their use for time modeling. In A. Sernadas, J. Bubenko, Jr., and A. Olivé, editors, *Information Systems: Theoretical and Formal Aspects*, pages 49–66, Amsterdam, 1985. North-Holland.

[26] G. Rozenberg, editor. *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, 1993. 457 pages.

[27] A. C. Shaw. Communicating real-time state machines. In Kemmerer and Ghezzi [20], pages 805–816.

[28] E. Smith. Comparability orders and measurement. In Rozenberg [26], pages 371–405.

[29] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall International, Upper Saddle River, New Jersey, 1995. 614 pages.

[30] J. J. P. Tsai, S. J. Yang, and Y.-H. Chang. Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications. *IEEE Transactions on Software Engineering*, 21(1):32–49, January 1995.

[31] V. Valero, D. de Frutos, and F. Cuartero. Timed processes of timed Petri nets. In de Michelis and Diaz [7], pages 490–509.

[32] W. M. P. van der Aalst. Interval timed coloured Petri nets and their analysis. In Ajmone Marsan [1], pages 453–492.

[33] C. D. Wilcox and G.-C. Roman. Reasoning about places, times, and actions in the presence of mobility. *IEEE Transactions on Software Engineering*, 22(4):225–247, April 1996.

Appendix: Drawing and annotation conventions

The net diagrams are labeled according to CPN conventions. There is one exception: the semantics of terms at high-level arrows. Consider an arrow to or from a place P. The set of marks which is denoted by the term `term` is the singleton set $\{(P, \texttt{term})\}$ if `term` does not denote a set, or $\{P\} \times$ `term` otherwise. That is, the denoted set of colors is either $\{\texttt{term}\}$ or `term`, respectively.

Multisets are not used. Therefore, two opposite arrows whose expressions denote the same color (which may, e.g., result from a fusion of places) should be considered a simplified representation of a more complex PN construct for side-conditions [18] rather than an extension to ordinary Petri nets.

For the sake of legibility, the following *drawing conventions* are used:
• A short arrow connecting a small (named) circle with a box stands for an ordinary arrow from or to the named place. In some diagrams, this place is drawn several times.
• A double arrow with two terms (Fig. 7) stands for two opposite arrows where the term being closer to the box belongs to the entry arrow (entering the box) and the term being more distant belongs to the exit arrow (leaving the box).
• A double arrow with no or a single term (Fig. 8, Fig. 15) stands for two opposite arrows with no or the same term, respectively.
• Extensive use has been made of arrow overlays. Double arrows, however, are not overlaid at all. Different arrow endings (with/without head) never coincide at circles, but may coincide at boxes. This allows an unambiguous resolution of arrow bundles.