Title: Time in Structured Occurrence Nets

Authors: Anirban Bhattacharyya, Bowen Li, and Brian Randell
Title: Time in Structured Occurrence Nets
Authors: Anirban Bhattacharyya, Bowen Li, and Brian Randell

Abstract

This paper presents a new tool-supported formalism based on collections of related timed occurrence nets, namely, timed structured occurrence nets (timed SONs) for the modelling and analysis of causally related events and concurrent events with uncertain or missing time information in evolving systems of systems. The application domain of timed SONs includes accident and crime investigations. A global discrete time model is used to support consistent causal reasoning about a system and time intervals are used to capture uncertainty about time values. We define the timed SONs notation and conditions for checking the consistency of time information, present algorithms of linear computational complexity for estimating missing time intervals using default duration intervals and redundant time information, and describe the facilities provided by the SONCraft tool.
Abstract

This paper presents a new tool-supported formalism based on collections of related timed occurrence nets, namely, timed structured occurrence nets (timed SONs) for the modelling and analysis of causally related events and concurrent events with uncertain or missing time information in evolving systems of systems. The application domain of timed SONs includes accident and crime investigations. A global discrete time model is used to support consistent causal reasoning about a system and time intervals are used to capture uncertainty about time values. We define the timed SONs notation and conditions for checking the consistency of time information, present algorithms of linear computational complexity for estimating missing time intervals using default duration intervals and redundant time information, and describe the facilities provided by the SONCraft tool.

About the authors

Dr Ani Bhattacharyya is currently a Research Associate in the Advanced Model-Based Engineering and Reasoning (AMBER), School of Computing Science at Newcastle University. Ani received a B.Sc. (Hons) in Mathematics from the University of London King’s College in 1982, and an M.Sc. in Information Systems Engineering from South Bank Polytechnic in 1984. His PhD was in Formal Modelling and Analysis of Dynamic Reconfiguration of Dependable Systems from Newcastle University in 2013, supervised by Professor John Fitzgerald. For his PhD, Ani extended the process algebra CCS with a special process (termed a fraction process) and overloaded the semantics of the parallel composition operator in order to model the runtime evolution of a system abstractly.

Bowen Li is currently a Senior Research Associate in the Advanced Model-Based Engineering and Reasoning (AMBER), School of Computing Science at Newcastle University. He is working on the EPSRC funded project UNCOVER (UNderstanding COmplex system eVolution through structurEd behaviours). An overall goal of UNCOVER is to develop a rigorous methodology supported by a toolkit based on structured occurrence nets, in order to provide an effective approach to acquiring and exploiting behavioural knowledge of a complex evolving system.

Professor Brian Randell graduated in Mathematics from Imperial College, London in 1957 and joined the English Electric Company where he led a team that implemented a number of compilers, including the Whetstone KDF9 Algol compiler. From 1964 to 1969 he was with IBM in the United States, mainly at the IBM T.J. Watson Research Center, working on operating systems, the design of ultra-high speed computers and computing system design methodology. He then became Professor of Computing Science at the University of Newcastle upon Tyne, where in 1971 he set up the project that initiated research into the possibility of software fault tolerance, and introduced the "recovery block" concept.

Suggested keywords

Keywords: timed structured occurrence nets, time intervals, constraint propagation, algorithms, tools, SONCraft
Time in Structured Occurrence Nets

Anirban Bhattacharyya, Bowen Li, and Brian Randell

School of Computing Science, Newcastle University
Newcastle upon Tyne, United Kingdom
{anirban.bhattacharyya, bowen.li, brian.randell}@ncl.ac.uk

Abstract. This paper presents a new tool-supported formalism based on collections of related timed occurrence nets, namely, timed structured occurrence nets (timed SONS) for the modelling and analysis of causally related events and concurrent events with uncertain or missing time information in evolving systems of systems. The application domain of timed SONS includes accident and crime investigations. A global discrete time model is used to support consistent causal reasoning about a system and time intervals are used to capture uncertainty about time values. We define the timed SONS notation and conditions for checking the consistency of time information, present algorithms of linear computational complexity for estimating missing time intervals using default duration intervals and redundant time information, and describe the facilities provided by the SONCraft tool.

Keywords: timed structured occurrence nets, time intervals, constraint propagation, algorithms, tools, SONCraft

1 Introduction

The concept of a structured occurrence net (SON) [7, 13, 11] is an extension of that of an occurrence net [3] – a directed acyclic graph that represents causality and concurrency information about a single execution of a system. SONS were created in order to characterise the behaviour of evolving systems of systems. Representing time information about such systems is also important. The SON concept originated from a general investigation of failure analysis, the problem of identifying the faults that might be the causes of an identified computer system failure. However, we believe that SONS are potentially of wide applicability, and two of the areas that we have been considering are accident and crime investigation. For example, in a criminal investigation, constructing a timeline of a crime for each suspected party is helpful in organising the evidence into a cohesive presentation for a court of law. However, in many cases, the time information available about an incident is not precise or is incomplete. For instance, it may not be possible to give an exact time (e.g. 9am) at which a robbery occurred, but it may be possible give time bounds for the robbery (e.g. 9am to 10am). Petri net-based research on uncertainty and computation of time information is limited. Therefore, the contribution of this paper is a new tool-supported formalism (timed SONS) that is based on collections of related timed occurrence
nets and is designed for modelling and reasoning about *causally related events and concurrent events with uncertain or missing time information* in evolving systems of systems.

The rest of the paper is organised as follows: **SONs** are briefly described in Section 2, and the notation of timed **SONs** (based on discrete time intervals) is given in Section 3. Conditions for checking the consistency of time intervals are defined in Section 4, and algorithms for estimating and for increasing the precision of time intervals using default duration intervals and redundant time information are given in Section 5; the algorithms are of linear computational complexity in the number of nodes in the **SON**. Support for timed **SONs** is provided by the SONCraft tool, which is described in Section 6. Related work is briefly reviewed in Section 7.

# 2 Structured Occurrence Nets

In this section, we first introduce the concept of occurrence nets, and then recall from [7, 14] several notions based on the structuring of occurrence nets.

## 2.1 Occurrence nets

An occurrence net (**ON**) is a directed acyclic graph used to record dependencies between events in a single execution of a concurrent system. A standard **ON** consists of three basic elements: **conditions** (denoted by \(C\)), **events** (denoted by \(E\)), and a binary **flow relation** (denoted by \(F\)). Each tuple of the **flow relation** represents an arc of the **ON** from a source condition to a destination event, or from a source event to a destination condition; the source node (condition or event) is termed an **input** of the destination node (event or condition respectively), and the destination node is termed an **output** of the source node. For a given node \(n\), the set of input nodes of \(n\) is denoted by \(\cdot n\), and the set of output nodes of \(n\) is denoted by \(n\). The **initial state** of an **ON** consists of the conditions with empty inputs, and the **final state** of an **ON** consists of the conditions with empty outputs. A **phase** is a fragment of an **ON** that begins with a global state and ends with a causally related subsequent global state and includes all the causally related events and conditions between the two global states. Figure 1 shows an **ON** which has been divided into two phases by three chosen global states.

![Fig. 1. An occurrence net divided into two phases by three global states.](image-url)
Occurrence nets were originally introduced as processes of running PT-nets [4]. Each process unambiguously and explicitly describes the causality and concurrency relations between executed events; more precisely, causally dependent occurrences of events are ordered, whereas concurrent occurrences of events are unordered. It is also possible to derive an occurrence net from historic data (e.g. in log files) in order to represent directly a system’s execution history [12]. However, the generality of causally related events and concurrent events enables them to model not only computing systems and their histories, but also components and systems involving people and natural processes, for example, parties involved in a crime investigation – one of the several application areas we have studied.

Since occurrence nets are acyclic, repetitions of the same condition or event are recorded as new nodes. Partially ordered sets are suitable as the underlying mathematical structure of occurrence nets.

### 2.2 Communication SONs

A standard Petri net represents an asynchronous relation, and does not provide means to synchronise different transitions directly. Communication structured occurrence nets (cSONs) are the basic variant of structured occurrence nets that can express synchronous (as well as asynchronous) interaction between communicating systems. Thus, the cSON concept is an extension of the concept of an occurrence net. Intuitively, a cSON model combines multiple related occurrence nets into a single structure by letting them communicate via two special relationships, namely, synchronous and asynchronous communication.

The original definition of cSONs represented a communication relation by a directed or undirected dashed line between two events [7]. Subsequently, the notion of channel place was introduced [6], which results in a more flexible representation of synchronous communication, see Figure 2. In a synchronous communication, the two communicating events must be executed simultaneously. In an asynchronous communication, the two events can be either executed simultaneously or the sending event executes before the receiving event.

---

Fig. 2. A cSON with two interacting occurrence nets.
2.3 Behavioural SONs

A behavioural structured occurrence net (BSON) represents the activity of an evolving system by representing the evolution of the system at different levels of abstraction.

Using the phase concept, a BSON provides a two-level view of execution history: the structure at the lower level provides the details of the system’s abstract behaviour represented at the upper level. The behavioural relations (denoted by $\beta$ and graphically represented by dashed lines) between the two levels express their consistent dependencies. Figure 3 shows an example of a BSON representing a system undergoing an (online) update. The upper level provides the evolution information concerning system version change caused by an update event. The lower level provides a detailed behaviour of the system. The behavioural relations between the two levels are used to capture the relationships between the two types of behaviour. In this case, the first half of the system behaviours (phaseA in Figure 1) belongs to the pre-update state $a_0$, and the second half of the system behaviours (phaseB in Figure 1) belongs to the post-update state $a_1$.

![Figure 3. A BSON portraying system (online) update.](image)

In order to capture causal dependencies between events at different levels of a BSON, we introduced a special relation $\text{causal}(e)$ in BSONs, which is formally defined in [8].

Intuitively, $\text{causal}(e)$ is the relation representing the directed graph of events that either directly or indirectly cause $e$ or are caused by $e$. For the BSON in Figure 3, we have:

$$\text{causal}(e_0) = \{(f_0, e_0), (f_1, e_0), (e_0, f_2), (e_0, f_3)\}$$

which indicates that $f_0$ and $f_1$ happen before $e_0$, and $e_0$ happens before $f_2$ and $f_3$.

---

1 We write $\text{before}(e)$ instead of $\text{causal}(e)$ in [8]; the change to $\text{causal}(e)$ is for greater clarity.
3 Time Model

In both criminal and accident investigations, it is important to establish the order in which events have occurred (i.e. the ‘chain of events’) and to establish the duration between events in order to determine causes and their effects, and thereby eliminate infeasible hypothesized scenarios and suspects from the investigation and if possible identify the real culprits of a crime or the actual causes of an accident. The notion of a global time enables different investigators to order a given set of events consistently (i.e. in the same order), which facilitates their cooperation. Therefore, the SONS used in an investigation should have a global time model.

Uncertainty is a common and unavoidable feature of investigations, in particular, uncertainty about the time of occurrence of an event, or the duration of the event, or the time at which a state comes into existence, or how long the state lasts. Fortunately, this uncertainty is often bounded. Such uncertainty should be modelled and taken into consideration when making causal inferences during investigations.

3.1 Global time

A global clock with a fixed origin and a fixed granularity implements a global time model. A physical clock has a fixed granularity, and (therefore) cannot order two events with a non-zero separation that is less than the clock granularity (unless the events occur on different sides of a clock tick). Hence, arbitrarily close timestamps of a dense global clock cannot be verified empirically in general. On the other hand, a global clock with a fixed granularity greater than or equal to that of a physical clock supports empirical verification of event ordering. The fixed origin of the global clock supports the correct ordering of events using timestamps.

The use of a fixed granularity for the global clock implies that integers can be used to represent time values, which simplifies computation. The use of different levels of abstraction (in BSONs) requires time abstraction, that is, coarser granularities of time corresponding to higher levels of abstraction, which can be implemented using clocks with larger units of integer-valued time that correspond to higher abstraction levels. The time unit of the base level of abstraction of a SON can be chosen such that the duration of each event is zero, which facilitates the computation of missing time values in an investigation. Therefore, the duration of a node resulting from an abstraction is the sum of the durations of its states at the base level of abstraction.

3.2 Modelling uncertainty

An integer interval is a simple way of representing a time value or a duration and the bounds on its uncertainty. Thus, the start time of a state, the finish time of the state, the start time of an event, the finish time of the event, the duration of the state, and the duration of the event (and the bounds on their respective
uncertainties) can each be represented using an integer interval. Certainty about a time value or a duration can be represented by making the two endpoints of their respective intervals identical.

For example, the occurrence of an instantaneous event at 9.00am can be represented by the interval [0900, 0900]. A state known to have started sometime between 9.00am and 12.30pm can be represented using the interval [0900, 1230]. An event known to have occurred at any time before 12.30pm can be represented using the interval [0000, 1230]. An event known to have occurred at any time after 12.30pm can be represented using the interval [1230, 9999], treating 9999 as the maximum possible time. An unknown time can be represented by the interval [0000, 9999].

Similarly, there are five possibilities for durations of states and of events, and they can be represented in a similar manner.

4 Time Information and its Consistency

We assume that each node of a son (i.e. condition, event, or channel place) has a start time ($T_s$) and a finish time ($T_f$), and that each time value has bounded uncertainty represented by a specified time interval ($[T_{s,l}, T_{s,u}]$ and $[T_{f,l}, T_{f,u}]$ respectively). We also assume the node has a duration ($D$) with bounded uncertainty represented by a specified duration interval ($[D_l, D_u]$), see Figure 4.

Fig. 4. Relationship between the unknown time and duration values of a node and their known bounds on the global timeline.

4.1 Notation

Let $n$ be a node of a son. The time information for $n$ is defined as follows.

The start and finish times of $n$ are denoted by $T^n_s$ and $T^n_f$ respectively.

The start time interval of $n$ represents the bounded uncertainty about the value of $T^n_s$, and is denoted by:

$$I^n_s \equiv [T^n_{s,l}, T^n_{s,u}]$$

where $T^n_{s,l}$ and $T^n_{s,u}$ are the lower and upper bounds respectively on the start time of $n$. $I^n_s$ is well-defined if and only if the following inequality is satisfied:

$$T^n_{s,l} \leq T^n_{s,u} \quad (1)$$
The finish time interval of \( n \) represents the bounded uncertainty about the value of \( T^n_{f} \), and is denoted by:

\[
I^n_{f} \triangleq [T^n_{f,l}, T^n_{f,u}]
\]

where \( T^n_{f,l} \) and \( T^n_{f,u} \) are the lower and upper bounds respectively on the finish time of \( n \). \( I^n_{f} \) is well-defined if and only if the following inequality is satisfied:

\[
T^n_{f,l} \leq T^n_{f,u} \quad (2)
\]

We assume the start time of \( n \) is at, or before, the finish time of \( n \), which is expressed by the restriction: \( T^n_{s} \leq T^n_{f} \). In order to ensure consistency with this restriction, the start and finish time intervals of \( n \) must satisfy the following inequalities:

\[
T^n_{s,l} \leq T^n_{f,l} \land T^n_{s,u} \leq T^n_{f,u} \quad (3)
\]

The duration of \( n \) is denoted by \( D^n \). The duration interval of \( n \) represents the bounded uncertainty about the value of \( D^n \), and is denoted by:

\[
I^n_{d} \triangleq [D^n_{l}, D^n_{u}]
\]

where \( D^n_{l} \) and \( D^n_{u} \) are the lower and upper bounds respectively on the duration of \( n \). \( I^n_{d} \) is well-defined if and only if the following inequality is satisfied:

\[
0 \leq D^n_{l} \leq D^n_{u} \quad (4)
\]

In this paper, if the node \( n \) is clear from the context, we will omit the superscript \( n \).

### 4.2 Time consistency

**Time consistency in linear ONs.** In a linear ON, each event has exactly one input condition and one output condition, and each condition has at most one input event and at most one output event.

We assume that for any two directly connected nodes (i.e. a condition ending in an event, or an event that starts a condition), the finish time of the source node is equal to the start time of the destination node. Therefore, we have the following:

\[
\forall n_1, n_2 \in (E \cup C) \ ((n_1, n_2) \in F \implies I^n_{f} = I^{n_2}_{s}) \quad (5)
\]

Let \( n \) be a node in a linear ON. The information with respect to the start time, finish time, and duration of \( n \) is defined to be **node consistent** if and only if the following inequalities are satisfied:

\[
[T_{s,l} + D_l, T_{s,u} + D_u] \cap [T^n_{f,l}, T^n_{f,u}] \neq \emptyset \quad (6)
\]

\[
[T^n_{f,l} - D_u, T^n_{f,u} - D_l] \cap [T_{s,l}, T_{s,u}] \neq \emptyset \quad (7)
\]

\[
[max\{0, T^n_{f,l} - T_{s,u}\}, T^n_{f,u} - T_{s,l}] \cap [D_l, D_u] \neq \emptyset \quad (8)
\]
The specified start time and duration intervals of \( n \) in combination bound uncertainty about the finish time of \( n \), and Condition (6) verifies the bounds are consistent (i.e. overlap) with the specified finish time interval of \( n \). Similarly, the specified finish time and duration intervals of \( n \) in combination bound uncertainty about the start time of \( n \), and Condition (7) verifies the bounds are consistent with the specified start time interval of \( n \). Condition (8) verifies that the bounds on uncertainty about the duration of \( n \) determined from the specified start and finish time intervals of \( n \) are consistent with the specified duration interval of \( n \). Condition (8) handles two cases, namely, the case where the uncertainty is such that the start and finish time intervals overlap and can be identical, when the condition evaluates to \([T_s, T_f, u - T_s, l] \cap [D_l, D_u] \neq \emptyset\), and the case where the two intervals are disjoint, when the condition evaluates to \([T_f, l - T_s, u, T_f, u - T_s, l] \cap [D_l, D_u] \neq \emptyset\).

A linear \( \text{ON} \) is defined to be \textit{time consistent} if and only if for all nodes \( n \) in the \( \text{ON} \), \( n \) is node consistent and the flow relation \( F \) of the \( \text{ON} \) satisfies Condition (5).

For example, consider the linear \( \text{ONs} \) shown in Figure 5. The time interval shown above each arc (prefixed by ‘T:’) represents the finish time interval of its source node as well as the start time interval of its destination node, and the duration interval of a node is prefixed by ‘D:’. The absence of a time or a duration interval of a node indicates that the information is unspecified, that is, \([0000, 9999]\). Using Condition (6) above, we can see that the time information of event \( e_1 \) in (a) is inconsistent, because its estimated finish time interval is \([T_s + D_l, T_s + D_u] = [0910, 1020]\), and its specified finish time interval is \([1030, 1100]\), and the two intervals do not intersect. In contrast, event \( e_1 \) in (b) is node consistent.

**Fig. 5.** Two linear \( \text{ONs} \) with time information.

**Time consistency in concurrent \( \text{ONs} \).** In a concurrent \( \text{ON} \), each event has at least one input condition and at least one output condition, and each condition has at most one input event and at most one output event.

We assume that for any two directly connected nodes, the finish time of the source node is equal to the start time of the destination node (as in linear \( \text{ONs} \)). An event starts if and only if all its input conditions are satisfied, and its output conditions are satisfied if and only if the event finishes. We assume there is no delay in the occurrence of the event. Therefore, the finish time of the input conditions must be the same as the event’s start time, and the start time of the output conditions must be the same as the event’s finish time. Therefore, we have the following definition.
Let \( e \) be an event in a concurrent \( \text{ON} \). The time information of \( e \) is defined to be \textit{concurrently consistent} if and only if the following conditions are satisfied:

\[
\forall c \in \bullet e (I^f_c = I^e_f) \tag{9}
\]

\[
\forall c' \in e^\bullet (I^e_s = I^f_{c'}) \tag{10}
\]

\( e \) is \textit{node consistent} \( \tag{11} \)

Thus, for any event \( e \) with multiple inputs and outputs, verifying its concurrent consistency consists of verifying that the finish time intervals of \( \bullet e \) are equal to the start time interval of \( e \), that the start time intervals of \( e^\bullet \) are equal to the finish time interval of \( e \), and that \( e \) is node consistent, that is, the start time, finish time, and duration intervals of \( e \) satisfy Conditions (6), (7), and (8).

A concurrent \( \text{ON} \) is defined to be \textit{time consistent} if and only if for all conditions \( e \) in the \( \text{ON} \), \( e \) is node consistent, and for all events \( e \) in the \( \text{ON} \), \( e \) is concurrently consistent.

\textbf{Time consistency in CSONs.} In CSONs, communication between events is represented using channel places that behave identically to conditions. In asynchronous communication, the sending event \( e \) executes either before the receiving event \( e' \), or \( e \) and \( e' \) execute simultaneously, and the two events are connected through an \textit{asynchronous channel place} that records information about the communication using a condition. In synchronous communication, the two communicating events execute simultaneously and are connected through two \textit{synchronous channel places} that record the communication information using conditions and have the same timing characteristics as the events.

Formally, let \( q \) be a channel place and let \( e, e' \) be the input and output events of \( q \) respectively. The time information of \( q \) is defined to be \textit{a/synchronously consistent} if and only if the following conditions are satisfied:

\[
I^f_q = I^e_f \tag{12}
\]

\[
I^e_s = I^f_{e'} \tag{13}
\]

\( q \) is \textit{node consistent} \( \tag{14} \)

Figure 6 shows how time information in a CSON can reveal the behaviour of events during asynchronous communication. In (a) the events \( f_0 \) and \( e_0 \) have the same start and finish time intervals, which indicate that the two events execute simultaneously. In (b) the time intervals indicate that \( f_0 \) executes earlier than \( e_0 \).

\textbf{Time consistency in BSONs.} The verification of time consistency in BSONs involves verifying time consistency between occurrence nets at different levels of abstraction using the behavioural (\( \beta \)) and \textit{causal} relations. For simplicity, we assume the different abstraction levels have the same time origin and granularity.

Given a BSON, let \( \text{causal}U \) be the binary relation consisting of the causally related pairs of events of the BSON that is defined as follows:

\[
\text{causal}U \triangleq \bigcup_{e \in \text{E}} \text{causal}(e)
\]
where $\mathbf{E}$ is the set of events in the ONs of the BSON. The time information of $\textit{causalU}$ is defined to be time consistent if and only if the following condition is satisfied:

$$\forall (g,h) \in \textit{causalU} \ (T^g_{s,l} \leq T^h_{s,l} \land T^g_{s,u} \leq T^h_{s,u})$$ (15)

For all conditions $c_i, c'_i \in \mathbf{C}$ ($\mathbf{C}$ is the set of conditions in the ONs of the BSON) such that $(c_i, c'_i) \in \beta$ and $c_i$ belongs to the initial state of a lower level ON of the BSON, the following condition must be satisfied:

$$I^c_i = I^{c'_i}$$ (16)

Moreover, for all conditions $c_t, c'_t$ such that $(c_t, c'_t) \in \beta$ and $c_t$ belongs to the final state of a lower level ON of the BSON, the following condition must be satisfied:

$$I^c_t = I^{c'_t}$$ (17)

For example, Figure 7 portrays a system undergoing an ‘offline modification’. The behaviour of the system is represented by two disjoint occurrence nets, since the situation portrayed is that of a modified system restarting its activities from some given initial state, rather than continuing from the state reached before the system modification started. The behaviour of such offline modification is reflected in the correspondence between time intervals in the two levels, as shown in the figure. The finish time interval of the pre-modified system ($c_3$) and the start time interval of the post-modified system ($c_4$) are identical to those of their corresponding upper level conditions.

5 Computation of Time Intervals

Investigations of crimes and accidents typically encounter situations where information is missing, or is unavailable, or is unknown. In such cases, it is often required to estimate the information that would have filled the gaps. Furthermore, in cases where complete time information is available (i.e. the start time, finish time, and duration intervals of all nodes are specified) the precision of the information can be increased. In the following, the estimated value of a quantity $X$ is denoted by $\hat{X}$, and all specified information is assumed to be consistent using the conditions defined in Section 4.
For a given node, the estimations of $I_s$, $I_f$, and $I_d$ are defined as follows:

Let $\tilde{I}_s \triangleq [\tilde{T}_{s,l}, \tilde{T}_{s,u}]$ and $\tilde{I}_f \triangleq [\tilde{T}_{f,l}, \tilde{T}_{f,u}]$ and $\tilde{I}_d \triangleq [\tilde{D}_l, \tilde{D}_u]$ (18)

In situations where complete time information is available for a node, the precision of the information can be increased using the following equations:

$$[\tilde{T}_{s,l}, \tilde{T}_{s,u}] = [T_{f,l} - D_u, T_{f,u} - D_l] \cap [T_{s,l}, T_{s,u}]$$

(19)

$$[\tilde{T}_{f,l}, \tilde{T}_{f,u}] = [T_{s,l} + D_l, T_{s,u} + D_u] \cap [T_{f,l}, T_{f,u}]$$

(20)

$$[\tilde{D}_l, \tilde{D}_u] = \max(\{0, T_{f,l} - T_{s,u}\}, T_{f,u} - T_{s,l}) \cap [D_l, D_u]$$

(21)

The start time, finish time, and duration intervals of a node collectively contain redundant information. Therefore, a missing interval of the node can be estimated if the other two intervals are specified, as shown below:

$$[\tilde{T}_{s,l}, \tilde{T}_{s,u}] = [T_{f,l} - D_u, T_{f,u} - D_l]$$

(22)

$$[\tilde{T}_{f,l}, \tilde{T}_{f,u}] = [T_{s,l} + D_l, T_{s,u} + D_u]$$

(23)

$$[\tilde{D}_l, \tilde{D}_u] = \max(\{0, T_{f,l} - T_{s,u}\}, T_{f,u} - T_{s,l})$$

(24)

In situations where a time interval and the duration interval of a node are missing, we assume it will be possible to use a default duration interval as an estimate based on statistics of durations of similar events or conditions that have occurred in the past, for example, the minimum and maximum duration of a telephone call, or the minimum and maximum duration of a train journey from London to York. Hence, the missing time interval of any node can be estimated using the default duration interval, the specified time interval, and Equation (22) or (23). If a node has a type, then the default duration interval of the node can be regarded as part of the node’s type information.

In situations where both time intervals of a node are missing, it is necessary to use a specified time interval of another node. We now describe algorithms for estimating missing time intervals of nodes in a SON using two approaches: the first approach is to estimate the intervals of an individual node, the second approach is to estimate the intervals of all the nodes of the SON.
5.1 Computation of time intervals of a node

The estimation of unspecified time intervals of a node involves traversing a SON structure using its causality relations, including the flow relations of its ONs, asynchronous and synchronous communications in CSONs, and causal(e) relations in BSONs. Algorithm 1 gives the structure of function causalPostset, which obtains the causal output neighbours of a given node n. Line 3 deals with flow relations: all nodes that are the outputs of n in the ON containing n are added to the result Postset. Lines 4 – 5 deal with CSON relations: the channel places contained in the outputs of n together with the output events of the channels (located in a different ON from that of n) are added to Postset. Lines 6 – 8 concern causal(e) relations in BSONs: for any causal(e) relation with domain \{n\}, the range is added to Postset.

Algorithm 1 (Causal postset)

1: function causalPostset(Node n)
2: Postset := \emptyset
3: add n* to Postset // output of n in ON containing n
4: for all channel place q such that n is the input of q do
5: add q and its output event to Postset // a/sync output of n
6: for all (e, f) ∈ causalU do
7: if e = n then
8: add f to Postset
9: return Postset

To determine the computational complexity of Algorithm 1, let k be the total number of nodes in the SON containing the node n. The total number of node additions to Postset is less than k, and the total number of tests of node equality to n is less than k. Therefore, the computational complexity of causalPostset is \(O(k)\).

A SON is essentially a directed acyclic graph, and (therefore) traversal of a SON can be performed in two directions. Function causalPreset obtains the causal input neighbours of a given node n; the structure of the function is given by Algorithm 8 in Appendix B. The computational complexity of causalPreset is \(O(k)\).

causalPostset and causalPreset are both used in estimating the unspecified time intervals of an individual node. Algorithm 2 describes the structure of procedure estimateFinish, which computes the finish time interval of a node n using the causal functions and is outlined below. The algorithm assumes the SON containing node n is represented by an acyclic structure.

1. Given a node n with unspecified finish time interval, perform forward breadth-first-search (BFS) using the findRightBoundary procedure to identify the nodes with a specified finish time interval that are nearest to n along paths beginning at n; if no such node exists on such a path, the final node of the SON on the path is used.
2. Using the identified nodes, perform backward BFS using the procedure `backwardBFSTimes` to calculate unspecified duration and time intervals of the nodes causally related to \( n \) (Lines 27 – 28, 32 – 33) or to recalculate the intervals to increase their precision (Lines 34 – 35), until node \( n \) is reached.

3. A count is kept (in `visits`) of the number of times a node is visited during the backward BFS traversal (Line 31) in order to handle multiple paths of unequal length between two nodes and to ensure that the time and duration intervals of a node are fully calculated exactly once (Lines 37 – 44).

Notice that the `estimateFinish` procedure can fail to compute the finish time interval of \( n \) if the computation involves an empty intersection between intervals (i.e. the SON contains inconsistent time information) or if the BFS is unable to find a node with a specified finish time interval on a path from \( n \) (i.e. the SON contains insufficient time information). The computational complexity of `estimateFinish` is \( O(k) \), since the number of operations on a node is bounded by a constant and each node of the SON is fully processed at most once.

Figure 8 shows the use of `estimateFinish` to compute the finish time interval of the initial node of an ON. The ON contains two specified time intervals: \( I_{c1}^f = [2001, 2005] \) and \( I_{c2}^f = [2004, 2008] \), and we assume the duration interval for each node is \([0001, 0001]\). To estimate the finish time interval of \( c_0 \), the forward search is used to find the right boundary, that is, \( \{e_2, e_1\} \). Then the backward BFS is performed to calculate intervals iteratively until reaching the initial node \( c_0 \). Notice that during the iteration, \( e_0 \) cannot be added to the working set until both its output conditions \( c_1 \) and \( c_2 \) have been visited.

```
Fig. 8. Estimating the finish time interval of a node.
```

Estimation of the start time interval of a node \( n \) is done in the reverse way to `estimateFinish` using the procedure `estimateStart` (not shown in this paper). The structural similarity of the two procedures implies their computational complexity is the same, that is, \( O(k) \).

### 5.2 Computation of time intervals of a SON

Procedure `estimateSONTimesBFS` estimates the unspecified time and duration intervals of an entire SON and increases the precision of the specified intervals through intersection using the principle that each interval of a SON must be able to affect the computation of every other interval. The structure of the procedure
Algorithm 2 (Estimates finish time interval of a node using causal relation)

1: procedure estimateFinish(Node \( n \))
2: \( RBoundary := \emptyset \) // nearest right nodes of \( n \) with specified finish time intervals
3: \( RNeighbourhood := \{ n \} \) // nodes on paths from \( n \) to \( RBoundary \) nodes
4: findRightBoundary(\( n \), \( RBoundary \), \( RNeighbourhood \))
5: backwardBFSTimes(\( n \), \( RBoundary \), \( RNeighbourhood \))

6: procedure findRightBoundary(Node \( n \), Set \( Boundary \), Neighbourhood)
7: \( Working := \{ n \} \) // nodes used for forward boundary searching
8: while \( Working \neq \emptyset \) do
9: \( NextWorking := \emptyset \) // nodes with unspecified finish time intervals
10: for all \( m \in Working \) do
11: \( \text{if causalPostset}(m) = \emptyset \) then
12: add \( m \) to \( Boundary \)
13: else
14: for all \( nd \in \text{causalPostset}(m) \) do
15: add \( nd \) to \( Neighbourhood \)
16: if \( nd.\text{finish}.\text{specified} \) then
17: add \( nd \) to \( Boundary \)
18: else
19: add \( nd \) to \( NextWorking \)
20: remove \( m \) from \( Working \)
21: \( Working := NextWorking \)

22: procedure backwardBFSTimes(Node \( n \), Set \( Boundary \), Neighbourhood)
23: \( Working := Boundary \) // nodes used for backward estimation of time intervals
24: while \( Working \neq \{ n \} \) do
25: \( NextWorking := \emptyset \) // nodes with unspecified time intervals
26: for all \( m \in Working \) do
27: \( \text{if } \neg m.\text{duration}.\text{specified} \) then
28: \( m.\text{duration} := \overline{I_{default}}(\text{typeof}(m)) \)
29: for all \( nd \in \text{causalPreset}(m) \cap Neighbourhood \) do
30: add \( nd \) to \( NextWorking \)
31: \( nd.\text{visits} := nd.\text{visits} + 1 \)
32: if \( \neg nd.\text{finish}.\text{specified} \land m.\text{finish}.\text{specified} \) then
33: \( nd.\text{finish} := m.\text{finish} - m.\text{duration} \) // Equation (22)
34: else if \( m.\text{finish}.\text{specified} \) then
35: \( nd.\text{finish} := \text{min}(m.\text{finish} - m.\text{duration}) \) // Equation (19)
36: for all \( nd \in NextWorking \) do
37: if \( \neg nd.\text{visits} = |\text{causalPostset}(nd)| \) then
38: for all \( ndout \in \text{causalPostset}(nd) \) do
39: \( \text{if } \neg ndout.\text{start}.\text{specified} \land ndout.\text{finish}.\text{specified} \) then
40: \( ndout.\text{start} := nd.\text{finish} \)
41: \( ndout.\text{duration} := ndout.\text{duration} \land (ndout.\text{finish} - ndout.\text{start}) \) // Equation (21)
42: \( nd.\text{visits} := 0 \)
43: else
44: remove \( nd \) from \( NextWorking \)
45: \( Working := NextWorking \)
is given by Algorithm 9 in Appendix B. Basically, a temporary pre-initial input node is created for the initial nodes of the SON and a temporary post-final output node is created for the final nodes of the SON to enable concurrent nodes to affect each other indirectly. This construction is often used in solving problems of directed acyclic graphs, for example, the maximum flow problem [10]. The finish time interval of the pre-initial node is calculated using the procedure estimateFinish described earlier, the SON is traversed going forward using BFS until the post-final node is reached, after which the SON is traversed going backward using BFS until the pre-initial node is reached. The computational complexity of estimateSONTimesBFS is $O(k)$, see Algorithm 9.

Figure 9 shows a cSON with one specified time interval. The pre-initial and post-final nodes of the SON are respectively connected to both initial conditions and both final conditions. The algorithm first estimates the finish time interval $[0958, 0958]$ of the pre-initial node, then uses the interval to estimate time information for the entire SON. Notice the time granularity in this example is 24-hour-clock/mins.

![Figure 9. A cSON with pre-initial and post-final nodes (all duration intervals are assumed to be $[0001, 0001]$).](image)

## 6 Implementation

We have implemented timed SONs and their analysis algorithms in SONCraft – an open source tool for the construction and analysis of structured occurrence nets. In this section, we provide an overview of the major features of SONCraft, then present the implementation of the time-based tools.

### 6.1 SONCraft

SONCraft supports the visual editing, simulation, analysis, and verification of structured occurrence nets. The tool is implemented as a Java plug-in within the Workcraft platform, which provides a flexible framework for the development and analysis of interpreted graph models. Detailed descriptions of SONCraft and Workcraft and their manuals can be found in [1,9]. The present version of SONCraft handles three types of SON structure, namely, cSONs, bSONs, and temporal abstractions [7]. A single SON can incorporate multiple instances of each type of structure. The major features of SONCraft are as follows.
Graphical user interface: an intuitive and easy to use graphical interface for editing, simulating, and analysing SON-based models, see Figure 10.

Simulator: a built-in simulator that can handle multiple ONS, CSONs, and BSONs in a SON. Simulations can be conducted either manually or automatically by firing a succession of enabled events, causing states to evolve, event colouring to be updated, and the simulation record to be augmented.

Analysis tools: a set of analysis tools is integrated into SONCraft. The structural property tool provides several structural verification algorithms that can be used to validate a model. It is important to verify the correctness of structure before further analysis; otherwise, the results are likely to be incorrect. Error tracing is a failure analysis facility integrated with the simulator. The facility is used to analyse how a ‘faulty’ event passes through and affects the system states. The reachability tool is used for verifying the reachability property of a SON. The analysis establishes whether or not given states are mutually concurrent.

Fig. 10. SONCraft interface.

6.2 Time visualisation

The time mode in SONCraft enables the display of time information of a SON model, as shown in the screenshot in Figure 11. Notice the editor window in the figure has been maximized for presentation purposes here, thereby minimizing all other windows (e.g. the editor tool window and the property editor window) in order to show only a single work file (i.e. the current SON model). In this mode, an initial condition is represented by a circle with a thick small arrow, and a final condition is represented by a double circle (inspired by the state representation used in finite state machines). The time representations of different node types are displayed differently because of the amount of visual space they occupy. Thus,
rather than display all three intervals (i.e. start, finish, and duration) for every node, we simplify the representation by merging and displaying some intervals on arcs. More precisely, the time interval displayed on each arc indicates the finish time interval of its source node as well as the start time interval of its destination node. Thus, each non-initial and non-final node shows only its duration value (see condition 'Ver 0.2'). However, there is no input arc for an initial condition and no output arc for a final condition; so, some of their time information is displayed directly against the node. For example, in Figure 11, the start time interval [2000, 2000] of the initial condition 'Ver 1.0' is displayed directly next to the node.

![Timed son visualisation.](image)

**Fig. 11.** Timed son visualisation.

### 6.3 Time property setting tool

The time property setting tool shown in Figure 12 is an interface for specifying time information of a given node in a son model. The time granularity panel at the top of the interface currently offers two granularities: year/year and 24-hour-clock/mins. Different granularities have their own time and duration bounds as well as arithmetic. For example, the time and duration bounds of the 24-hour-clock/mins granularity are 0001-2400 and 0000-0060 respectively.

Users can either manually or automatically set time information for a selected node in the time value panel. For each manually input interval, the tool verifies whether or not the interval is well-defined. The verification criteria are based on Conditions (1) – (4). Moreover, for each input interval, checking of its time or duration bounds is performed according to its time granularity. The tool also provides efficient time estimation for nodes with unspecified time intervals. Depending on the user selection, different algorithms (e.g. Algorithm 2 or entire son estimation) can be used for a calculation.
6.4 Time consistency checking tool

The time consistency checking tool provides consistency checking for the time information that is specified. The tool encodes the conditions and equations given in Section 4, and provides a user interface for additional settings. The interface is divided into the following three (sub-)panels.

Two granularities are present in the time granularity panel. Causal consistency checking can be activated using the causal consistency panel. The facility implements Algorithm 7 in Appendix A and aims to verify the nodes with incomplete time information using causal relations. The panel also includes means of specifying the default duration setting that is used in time estimation. The user settings panel has an option to request the highlight of time inconsistent nodes (if any) in the editor window after the verification.

Figure 13 shows a consistency verification result of the son displayed in Figure 11. The result shows three time inconsistency errors. For example, the first error message involves condition $c_4$ and shows that its start time lower bound (2001) is greater than its finish time lower bound (2000), that is, the condition can cease to hold before it starts to hold.

7 Related Work

Petri net-based research on uncertainty, consistency checking, and computation of time information is limited. However, there is research on genealogies (e.g., [5] and [15]) and on temporal logics (e.g., [2]) that addresses these issues. In [5], a mathematical relaxation method (of exponential complexity in the number of dates) is used to adjust iteratively the endpoints of intervals containing unknown dates of birth, marriage, and death until the endpoints finally stabilise. The restrictions on the dates are encoded in the algorithm, and their parameters are set manually. In [15], the restrictions on the dates determine intervals, which
are intersected in order to determine the final endpoints. The parameters of the restrictions are determined by statistical analysis of the input data. In [2], an algebra of relations between temporal intervals is developed with a method (of quadratic complexity in the number of intervals) for determining the relation (one of thirteen) between any two intervals. However, none of the research models abstraction of events or of states or models communication.

8 Concluding Remarks

This paper has presented the notion of a timed SON in order to model and reason about causally related events and concurrent events with uncertain or missing time information in evolving systems of systems. Discrete intervals have been used to capture uncertainty about time values. Conditions have been defined to verify the consistency of time information. Algorithms have been presented that are based on the use of default duration intervals and redundant time information and are of linear computational complexity in the number of nodes in a SON in order to estimate missing time intervals and to increase the precision of user-specified intervals. Finally, the facilities provided by the SONCraft tool have been described.

Future work will extend timed SONS to handle multiple scenarios so that different hypotheses about the cause of an accident or the chain of events of a
crime can be modelled and investigated. Probabilities will be added to support multiple scenarios, and different time granularities will be represented to support modelling at different levels of abstraction. Furthermore, we hope to use timed actors to model and analyse cyber crime, problems involving ‘big data’, neuro- logical processes, dynamic reconfiguration of real-time software, and hardware design, in order to demonstrate the generality and exploit the full potential of the formalism.

Acknowledgements

This work is funded by the EPSRC project UNderstanding COmplex system eVolution through structurEd behaviouRs (UNCOVER). The authors acknowledge the help and advice given by Victor Khomenko and Maciej Koutny.

References

1. Workcraft homepage, \url{http://workcraft.org}
A Algorithms for Checking Consistency

We describe algorithms for checking consistency that implement the conditions given in Section 4.

nodeConsistency is the basic function that implements Conditions (6), (7), and (8) to verify the consistency of a given node with specified start, finish, and duration intervals. The structure of the function is given by Algorithm 3. Lines 3, 6, and 9 compute the intersection between a specified interval and its estimate, and return FALSE if the intersection is empty. Thus, the function returns FALSE if the provided time information does not satisfy a condition; otherwise, it returns TRUE.

Algorithm 3 (Node consistency)

1: function Boolean nodeConsistency(Node n)
2:  ˜I_in := n.start + n.duration // Equation (23)
3:  if ˜I_in ∩ n.finish = ∅ then // Condition (6)
4:      return FALSE
5:  ˜I_is := n.finish − n.duration // Equation (22)
6:  if ˜I_is ∩ n.start = ∅ then // Condition (7)
7:      return FALSE
8:  ˜I_id := n.finish − n.start // Equation (24)
9:  if ˜I_id ∩ n.duration = ∅ then // Condition (8)
10:     return FALSE
11:    return TRUE

The concurConsistency function implements Conditions (9), (10), and (11), and is invoked whenever the main consistency checking task attempts to verify the consistency of an event. The structure of the function is given by Algorithm 4. The function first verifies concurrent consistency with respect to the finish time intervals of the input states of the given event, then with respect to the start time intervals of the output states of the event, then invokes nodeConsistency for the basic consistency checking of the event itself.

Algorithm 4 (Concurrent consistency)

1: function Boolean concurConsistency(Event e)
2:  for c ∈ e do
3:    if c.finish ≠ e.start then // Condition (9)
4:      return FALSE
5:  for c ∈ e do
6:    if c.start ≠ e.finish then // Condition (10)
7:      return FALSE
8:  return nodeConsistency(e) // Condition (11)
The `asynConsistency` function is invoked only if a SON contains a communication relation, since only a channel place can be passed as a parameter. The structure of the function is given by Algorithm 5. The function applies Conditions (12) and (13) for the asynchronous and synchronous-based checking, then invokes `nodeConsistency` for the basic consistency checking of the channel place itself.

Algorithm 5 (A/Synchronous consistency)

1: function Boolean asynConsistency(Channel place q)
2: if q.input.finish ≠ q.start ∨ q.output.start ≠ q.finish then // Cnds. (12), (13)
3: return FALSE
4: return nodeConsistency(q) // Condition (14)

The `bhvConsistency` function is used to verify the time consistency of a BSON. The structure of the function is given by Algorithm 6. The function first verifies the consistency of all binary relations in `causalU` using Condition (15), then uses Conditions (16) and (17) to verify the restrictions on the initial and final states of the BSON. The return value of the function is all nodes which are behaviourally inconsistent in the BSON.

Algorithm 6 (Behavioural consistency)

1: function bhvConsistency(Relation causalU)
2: Result := ∅ // behaviourally inconsistent nodes
3: for (e₁, e₂) ∈ causalU do // Condition (15)
4: if e₁.start.lower > e₂.start.lower ∨ e₁.start.upper > e₂.start.upper then
5: add e₁, e₂ to Result
6: for c ∈ C do
7: if *c = ∅ ∧ c.start ≠ β(c).start then // Condition (16)
8: add c to Result
9: else if c̃ = ∅ ∧ c.finish ≠ β(c).finish then // Condition (17)
10: add c to Result
11: return Result

Algorithm 7 gives the structure of the `sonConsistency` function, which verifies the time consistency of an entire SON. The algorithm begins by assigning to each node of the SON with an unspecified duration interval a user-defined default duration interval that corresponds to the node type and is used later for time estimation. Then, the algorithm traverses the whole SON structure and verifies the time consistency of the specified time intervals of each node. Any node with complete time information is passed directly to the `consistencyTask` function for consistency checking. For a node with partial time information, the missing time interval is estimated before invoking `consistencyTask`. 
Algorithm 7 (Consistency checking task)

1: function sonConsistency(SON S)
2: input: SON S
3: output: Set INCONSIS – set of nodes with inconsistent time/duration intervals
4:
5: for all node n in SON do
6: if ¬n.duration.specified then
7: n.duration := I_{default}(typeof(n))
8: for all node n in SON do
9: if n.start.specified ∧ n.finish.specified then
10: consistencyTask(n)
11: else if n.start.specified ∧ ¬n.finish.specified then
12: estimateFinish(n)
13: if n.finish.specified then
14: consistencyTask(n)
15: else if ¬n.start.specified ∧ n.finish.specified then
16: estimateStart(n)
17: if n.start.specified then
18: consistencyTask(n)
19: add all nodes in bheConsistency(causalU) to INCONSIS
20: function consistencyTask(Node n)
21: if n is event ∧ ¬concurConsistency(n) then
22: add n to INCONSIS
23: else if n is channel place ∧ ¬asynConsistency(n) then
24: add n to INCONSIS

B Algorithms for Computing Time Intervals

Algorithm 8 gives the structure of function causalPreset, which obtains the causal input neighbours of a given node n of a SON. Line 3 deals with flow relations: all nodes that are the inputs of n in the ON containing n are added to the result Preset. Lines 4 – 5 deal with cson relations: the channel places contained in the inputs of n together with the input events of the channels (located in a different ON from that of n) are added to Preset. Lines 6 – 8 concern causal(e) relations in bsons: for any causal(e) relation with range \{n\}, the domain is added to Preset.

The structural similarity of Algorithm 8 to Algorithm 1 implies the computational complexity of the two algorithms is the same, that is, \(O(k)\).

Algorithm 9 gives the structure of procedure estimateSONTimesBFS, which estimates the unspecified time and duration intervals of a given SON (Lines 32 – 33, 30 – 31) and increases the precision of the specified intervals (Lines 21 – 24, 28 – 29, 34 – 36). The algorithm enables nodes (including concurrent nodes) to affect the computation of each other’s intervals, which is achieved as follows: a single temporary pre-initial input node for the initial nodes of the SON and a single temporary post-final output node for the final nodes of the SON are created (Lines 5 – 8). Then, the estimateFinish procedure (described in Section 5.1) is
used to combine the start time intervals of the concurrent initial nodes of the SON through intersection with the finish time interval of the pre-initial node (Line 9), which is then used to compute intervals throughout the SON by forward BFS using the forwardBFSsonTimes procedure (Line 10) until the post-final node is reached. Then, the backwardBFSsonTimes procedure (not shown in this paper) is used to perform the reverse BFS traversal of the SON in order to compute the effect of the node intervals on the right of the SON on the node intervals on the left (Line 11); backwardBFSsonTimes and forwardBFSsonTimes are similar in structure. Finally, the pre-initial and post-final nodes are deleted (Line 12). Notice that in order to handle multiple paths of unequal length between nodes, forwardBFSsonTimes maintains the visits counter for each node (Line 20). The counter records the number of visits made to a node by its input nodes, and the final processing of the node is delayed (Lines 38 – 39) until all its input nodes have visited (Lines 26 – 37). Thus, forwardBFSsonTimes processes each node of the SON exactly once.

The computational complexity of estimateSONTimesBFS is $O(k)$, since the number of operations on a node and the number of times the node is processed are bounded by a constant.
Algorithm 9 (Estimates time intervals for an entire SON)

1: **procedure** estimateSONTimesBFS(SON $S$)
2: **input:** SON $S$
3: **output:** SON $S$ with estimated time intervals of all nodes
4: 5: add pre-initial node $e_1$ to $S$
6: add all causal relations $(e_1, x)$ to $S$ such that $x \in \text{Int}_S$, where $\text{Int}_S$ is the set of
7: initial conditions of $S$
8: add post-final node $e_2$ to $S$
9: add all causal relations $(y, e_2)$ to $S$ such that $y \in \text{Fin}_S$, where $\text{Fin}_S$ is the set of
10: final conditions of $S$
11: estimateFinish($e_1$)
12: forwardBFSsonTimes($e_1$)
13: backwardBFSsonTimes($e_2$)
14: remove pre-initial node $e_1$, post-final node $e_2$, and their causal relations from $S$

13: **procedure** forwardBFSsonTimes(Node $n$)
14: $\text{Working} := \{n\}$  // nodes used for forward estimation of time/duration intervals
15: while $\text{Working} \neq \emptyset$ do
16: $\text{NextWorking} := \emptyset$  // nodes whose time/duration intervals are to be estimated
17: for all $m \in \text{Working}$ do
18: for all $nd \in \text{causalPostset}(m)$ do
19: add $nd$ to $\text{NextWorking}$
20: $nd\text{.visits} := nd\text{.visits} + 1$
21: if $nd\text{.start}\text{.specified}$ then
22: $m\text{.finish} := m\text{.finish} \cap nd\text{.start}$
23: for all $nd \in \text{causalPostset}(m)$ do
24: $nd\text{.start} := m\text{.finish}$
25: for all $nd \in \text{NextWorking}$ do
26: if $nd\text{.visits} = |\text{causalPreset}(nd)|$ then
27: if $|\text{causalPreset}(nd)| > 1$ then
28: for all $ndin \in \text{causalPreset}(nd)$ do
29: $ndin\text{.finish} := nd\text{.start}$
30: if $\neg nd\text{.duration}\text{.specified}$ then
31: $nd\text{.duration} := I_{\text{default}}(\text{typeof}(nd))$
32: if $\neg nd\text{.finish}\text{.specified}$ then
33: $nd\text{.finish} := nd\text{.start} + nd\text{.duration}$  // Equation (23)
34: else
35: $nd\text{.finish} := nd\text{.finish} \cap (nd\text{.start} + nd\text{.duration})$  // Equation (20)
36: $nd\text{.duration} := nd\text{.duration} \cap (nd\text{.finish} - nd\text{.start})$  // Equation (21)
37: $nd\text{.visits} := 0$
38: else
39: remove $nd$ from $\text{NextWorking}$
40: $\text{Working} := \text{NextWorking}$