Visualisation and Analysis of Complex Behaviours using Structured Occurrence Nets

Bowen Li

School of Computing Science

University of Newcastle

A thesis submitted for the degree of

Doctor of Philosophy

January 2017

Acknowledgements

I would like to say a massive thank you to my supervisor, Prof. Maciej Koutny. This research would not have been possible without your guidance, encouragement and continued support. I would like to express my special appreciation and thanks to Prof. Brian Randell for your countless insightful ideas and suggestions.

I am also grateful for the support received from my colleagues in the School of Computing Science and School of Electrical and Electronic Engineering at Newcastle Univerisy. Special thanks go to Dr. Anirban Bhattacharyya for the collaboration and fruitful discussions, Dr. Danil Sokolov for the support of SONCraft implementation, and my office-mate Dr. Frank Burns for the help and advice.

Many friends have been exceptionally helpful and made my PhD life enjoyable. In particular, I would like to say thank you to Dr. Zhenyu Wen and Qi Shen for being my friends since we did MSc degree in Newcastle.

I wish to thank my examiners, Prof. Marian Gheorghe and Dr. Victor Khomenko for their expertise and suggestions for improving my thesis.

Finally, and most importantly, I thank my parents and my wife Min. All the support, love and encouragement they have provided me was the greatest gift anyone has ever given me.

This work was supported in part by EPSRC EP/K001698/1 UNderstanding COmplex system eVolution through structurEd behaviouRs (UNCOVER) project.

Abstract

A complex evolving system consists of a large number of sub-systems which may proceed concurrently and interact with each other or with the external environment, while its behaviour is subject to modification by other systems. Structured occurrence nets (SONs) are a Petri net based formalism for modelling the behaviour of complex evolving systems. The concept extends that of occurrence nets, a formalism that can be used to record causality and concurrency information concerning a single execution of a system. In SONs, multiple occurrence nets are combined using various types of relationships in order to represent dependencies between communicating and evolving sub-systems.

The work presented in this thesis aims to develop a tool and extend existing methodology for structured representations of the behaviours of complex evolving system. The theoretical development focuses on the extension of existing SON concepts. It addresses the issue of efficient SON model checking and simulation, representations of alternative behaviour and time information, structuring SON-based unfolding, and algorithms for constructing the unfolding. The implementation aims to develop tools for SON-based model visualisation, simulation and analysis. An open source tool called SONCraft has been developed to support these functionalities. SONCraft provides a user-friendly graphical interface that facilitates model entry, supports interactive visual simulation, and allows the use of a set of analytical tools for model checking.

Contents

Chapter 1

Introduction

1.1 Background

Over the past decades, the study of complex systems has become ever more important. Such systems exhibit some common characteristics:

1. The system has intricate *structure*, which may consist of a large number of (sub)systems interacting with each other and with the system's environment.

2. The system exhibits *emergent* behaviours: that is, the behaviours that arise from the interaction of subsystems that on their own do not have such properties.

3. The system can *evolve*: a complex system is not created all at once but instead is subject to modification by other systems and changes over time.

Examples include the economy, biological systems, financial markets, our society and information systems. Consider a cloud computing system which is composed of a huge amount of interacting services and clients as subsystems. The communication between subsystems may be either asynchronous or synchronous. The cloud infrastructure can suffer from component break-downs, reconfigurations and replacement, and the software is continually updated or patched. Such very diverse 'event-based' systems can have a very high complexity of both design and behaviour, with extremely large volumes of recordable events or facts

and the consequential combinatorial state space explosion. They can also exhibit intricate dependencies in the representation of state information, and dynamic (planned or unforeseen) system evolution and reconfiguration.

In complex systems, the role of structure is often considered to be essential to coping with design complexity. For example, in the software engineering domain, structuring facilities include procedures, threads, classes and types, and in the VLSI design domain, with its higher order logics, graph-based models, and design notations, and so on are used. The effective use of such structuring notations can significantly reduce the cognitive complexity of designs, and the resources, both storage and computational, involved in their representation and manipulation.

Evolution is another important property which needs to be considered in the design of complex systems, especially software systems. Any design approach based on the assumption of complete and correct requirements is detrimental to the development of reliable and usable software systems. In software engineering, successful software always evolves. Empirical data shows that maintenance absorbs 40-60 percent of the life-cycle costs of a complex system and 75 percent of the total maintenance efforts are enhancements [54]. In light of this, system evolution should be considered *a priori* by the designer.

The purpose of system design is to define how a system will behave. To improve the dependability of a system, system behaviour representations are often used for system visualisation, verification, synthesis and failure analysis. There are many well-established conventional behavioural models used in industry nowadays; for example, Petri nets [57], process algebras [38], and Unified Modelling Language (UML) [53], but very few of them support modelling evolution in an efficient way. Moreover, the cognitive complexity of the notation for recording complex system behaviour is not always the dominant concern compared with those for system design notation. Until recently there has been very little work on the structuring of behaviour notation.

In order to manage the cognitive complexity of complex and evolving system behaviours, it is necessary to choose an appropriate notation. The *directed acyclic graph* is a typical notation for recording the behaviour of an asynchronous system. The most well-developed such notation, in terms of its formal theoretical basis and tool support, is that of an *occurrence net* (ON) [19, 20]. ONs are used for

representing causality and concurrency information concerning a single execution of a system. One can derive an ON in various ways: as a process underpinning a run of Petri nets [35], process algebras [49], network diagnostics [5], and system diagnosis [44]; or even as a direct representation of a system's execution history. In ONs, only information about concurrency and causality between events and visited local states is represented, and the underlying mathematical structure is a partial order, and so they are in widespread use in partial order verification [25].

The formalism of *structured occurrence nets* (SON) [14, 16, 40], which is an extension of the occurrence net formalism, has been introduced to characterise the behaviour of complex evolving systems. The underlying idea of a SON is to combine multiple related occurrence nets by using various formal relationships (in particular, abstractions) in order to express dependencies between interacting and evolving systems. For example, SONs can directly represent the behaviour of system evolution through their use of a new formal relation termed 'behavioural abstraction'. By means of these relations, a SON is able to portray an explicit view of system evolution, involving various types of communication, system upgrades, reconfigurations and replacements, that allows one to exploit the behavioural knowledge of a complex evolving system.

The original interest in SONs arose out of a wish to improve the understanding of the concept of basic dependability, and in particular that of 'fault-error-failure chains' [6]. Now, however, the main motivation for defining and exploring the formal properties of SONs is to achieve a significant reduction in the cognitive difficulties and computational effort required in using ONs for the modelling and analysing of the behaviour of complex evolving systems. For example, the idea of extended SONs that facilitate the structuring and analysis of incomplete, contradictory and uncertain behaviours in complex systems involving software, hardware and people (e.g., cybercrime) has been discussed [41]; and the full system semantics of SONs which can be regarded as specifications for system designs have been investigated [28].

1.2 Aims and Contributions

The importance of understanding and analysing the behaviour of complex evolving systems is well-accepted in many application areas. The concept of structured occurrence nets can play an important role with regard to the representation of the behaviours of such systems. However, there is a scarcity of research on the assessment of the practical usefulness of the new representations. To achieve such an assessment, effective formal support including automated analysis and formal verification are required, and appropriate design tools facilitating the manipulation and analysis of SONs are necessary. Our research hypothesis is that structured occurrence nets supported by an appropriate toolkit can deliver an effective approach to exploiting and analysing knowledge of the behaviour of a complex evolving system.

To validate this hypothesis, a generic methodology involving both theory and practical research is employed. The main aims of the study are as follows:

Aim-1, Theory: To provide a formal foundation for SON concepts, involving the extension of current formalisation and proofs of a number of results concerning the well-formedness of SONs which hence govern the correct use, manipulation and analysis of various types of abstraction relation.

Aim-2, Toolkit: To develop a platform for constructing and editing SON-based models, and to provide dedicated SON-based tools for system verification, simulation and analysis.

Aim-3, Evaluation: To assess the utility of SON-based models.

With regard to Aim-1, we propose several additional properties of the basic SON structure. We provide new execution semantics for all SON variants for a step by step simulation, and we design algorithms for structural property verification, reachability checking and the simulation of SONs.

We extend the existing basic SON model to formally support alternative representations of a given behaviour based on the idea in [41]. The new structure allows one to model multiple alternative scenarios that could have occurred in

particular states. We also extend basic SON simulation algorithms to support new representations.

We introduce a time property to basic and alternative SONs. We formally describe how such timing information as is provided in a SON can be checked for consistency, and how the estimated time information of a given node or entire SON can be calculated by using causal relations. We present time-based algorithms for consistency checking and time estimations.

We investigate the unfolding of CSPT-nets, where CSPT-nets (communication structured place transition nets) are one of the generator nets in SONs supporting a representation of synchronous or asynchronous interaction between multiple sub-systems [28]. Such an unfolding contains a representation of all the possible running processes of the original net. The prefix of the unfolding has a smaller size than the reachability graph of a system, so could alleviate the state space explosion problem in model checking; for example, for reachability analysis. We provide an algorithm for the construction of CSPT unfolding.

In pursuit of Aim-2, we develop SONCraft, which is an open source tool for SON visualisation, verification, and model analysis. The tool is implemented as a Java plug-in to the Workcraft platform system which provides a flexible framework for the development and analysis of Interpreted Graph Models [22, 23]. SONCraft provides a user-friendly graphical interface that facilitates SON model entry, supports interactive visual simulation, and integrates a set of analysis tools.

More specifically, for basic SONs we implement the essential functionalities for their creation, visualisation and manipulation as well as facilities for their simulation, failure analysis, structural property verification and reachability checking. We implement time based algorithms for visualising time properties, checking consistency and estimating missing time information.

With regard to Aim-3, we apply SONCraft to a scenario related to train accident in order to assess the practicality of SON-based modelling.

1.3 Outline of the Thesis

The rest of the thesis is organised as follows.

Chapter 2 presents the notions and properties concerning structured occurrence nets and their verification and simulation, and presents a case study of an accident scenario.

Chapter 3 defines representations of SONs with alternatives, their properties and model checking approaches.

Chapter 4 describes SONs with time information, and algorithms for time consistency checking and estimation.

Chapter 5 outlines the SONCRAFT framework, and describes additional tools that have been added for SON-based models verification, simulation and analysis.

Chapter 6 discusses the theory of and algorithm for SON-based unfolding.

Chapter 7 summarises and concludes the work and proposes directions for further work.

1.4 List of Publications

Portions of the work within this thesis have been documented in the following publications:

Conferences/Workshops

1. Li, B., *Branching processes of communication structured PT-nets.* In: PhD session at International Conference on Application of Concurrency to System Design (ACSD), 2013

2. Li, B., Koutny, M. *Unfolding CSPT-nets.* In: International Workshop on Petri Nets and Software Engineering (PNSE), 2015.

3. Bhattacharyya. A., Li, B., Randell, B. *Time and space in SONs.* In: International Workshop on Petri Nets and Software Engineering (PNSE), 2016.

Reports

1. Li, B., Randell, B. *SONCraft user manual.* Tech. Rep. CS-TR-1448, School of Computing Science, Newcastle University, 2015.

2. Li, B., Koutny, M., Randell, B. *SONCraft: A tool for construction, simulation and verification of structured occurrence nets.*, Tech. Rep. CS-TR-1493, School of Computing Science, Newcastle University, 2016.

Chapter 2

Structured Occurrence Nets

2.1 Introduction

Structured occurrence nets (SONs) are a Petri net-based formalism that can be used to model the behaviour of complex evolving systems. The concept extends that of occurrence nets (ONs) which are directed acyclic graphs that represent causality and concurrency information concerning a single execution of a system. SONs are sets of related ONs, employing different types of formally-defined relations and supporting various types of abstraction. By means of different variants a SON can represent, for example, the event of one system modifying another system, and the causal antecedents and consequences of this event in each system.

Communication structured occurrence nets (CSONs) are the fundamental variant of structured occurrence nets that has the capability of representing asynchronous and synchronous interactions between communicating systems. Intuitively, a CSON combines two or more occurrence nets into a single structure by letting them communicate via two special relationships, viz. synchronous and asynchronous communication. The former implies that a sender waits for an acknowledgement of a message before proceeding, while in the latter the sender proceeds without waiting.

Behavioural structured occurrence nets (BSONs) convey information about the evolution of individual systems. A system in BSON shows a two-level view of its execution history, where the structure at a lower level provides details of

its abstract behaviour represented at an upper level. The abstract (behavioural) relations between two different levels show their consistent dependencies.

Temporal structured occurrence nets (TSONs) allow the use of temporal abstraction to define atomic actions, that is, actions that appear to be instantaneous to their environment. Intuitively, a TSON shows a system abbreviation as that part of the behaviour that is hidden by the abstraction.

In this chapter we first introduce the concept of occurrence nets, and then recall from previous research [40] several notions and properties based on the structure of occurrence nets. In addition, we propose algorithms used for SON model checking, and present a case study of an accident scenario modelled by a SON.

2.2 Occurrence Nets

Occurrence nets were initially introduced as processes of running Petri nets [19]. Each process unambiguously and explicitly describes the concurrency and causality relations between executed events; more precisely, causally dependent occurrences of events are ordered while their concurrent occurrences are unordered. It is also possible to derive an ON as a direct representation of a system's execution history [15]; such a system may involve not only computer components, but also components and systems involving people and physical processes; for example, parties involved in a crime and accident investigation. Since ONs are acyclic, repetitions of the same condition or event are recorded as new elements. Partially ordered sets are suitable as the underlying mathematical structure of ONs.

An *occurrence net* is a finite triple $\text{ON} = (C, E, F)$, where C and E are disjoint sets of respectively *conditions* and *events* (collectively referred to as the *nodes*), and $F \subseteq (C \times E) \cup (E \times C)$ is the *flow* relation. The *inputs* and *outputs* of a node x are respectively defined as ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$[1]. For a set of nodes $X \in (C \cup E)$, we respectively denote using ${}^\bullet X$ and $X^\bullet$ the sets of all inputs and outputs of a node in X. It is also assumed that the following are satisfied:

[1] In this thesis, sometimes we will, for the purpose of clarity, use the notations $pre(x)$ and $post(x)$ instead of a 'dot' to represent input and output.

- For all $c \in C$ and $e \in E$: $|{}^\bullet c| \leq 1$, $|c^\bullet| \leq 1$, $|{}^\bullet e| \geq 1$, and $|e^\bullet| \geq 1$.
- The causality relation $\prec$ over E is acyclic, where $e \prec f$ if there is $c \in C$ with $c \in e^\bullet \cap {}^\bullet f$.

$M_0^{\text{ON}} = \{c \in C \mid {}^\bullet c = \emptyset\}$ is the *initial marking* of an ON, and $M_{Fin}^{\text{ON}} = \{c \in C \mid c^\bullet = \emptyset\}$ is the *final marking* of an ON (in general, a *marking* is any set of conditions).

To summarise, an occurrence net is an acyclic directed graph, which consists of conditions, events, and arcs. Each arc runs 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. Each condition has at most one input event and at most one output event; and each event has at least one input condition and at least one output condition. The set of all conditions with no input events is the initial marking, and the set of all conditions with no output events is the final marking.

Two nodes, x and y, are causally related if $(x, y) \in F^+$ or $(y, x) \in F^+$; otherwise they are concurrent (denoted by $x \; co \; y$). A *co-set* is a set $B \subseteq C$ comprising pairwise concurrent conditions. Moreover, a *cut* is any maximal (w.r.t. $\subseteq$) co-set.

Next we recall notions and properties concerning occurrence nets which are useful in the rest of the study. In this thesis, if a variant of SONs is clear from the context, we will write the corresponding initial marking as M_0, and the final marking as M_{Fin}.

Given an initial marking, the execution of an occurrence net proceeds by the occurrence (or firing) of sets of events. The firing rule below specifies the conditions under which a marking enables a set of events (called a *step*), and how the firing of the events changes the current marking.

Definition 2.1 (ON firing rule)**.** *Let* $\text{ON} = (C, E, F)$ *be an occurrence net,* M *be a marking, and* U *be a step of* ON*.*

1. U *is* ON*-enabled at* M *if* ${}^\bullet e \subseteq M$*, for every* $e \in U$*.*

2. *If U is* ON*-enabled at M, then U can be fired and produce a new marking M' given by $M' = (M \setminus {}^\bullet U) \cup U^\bullet$,*
This is denoted by $M[U\rangle_{\text{ON}} M'$.

A *step sequence* of ON is a sequence $\lambda = U_1 \dots U_n$ $(n \geq 0)$ of steps such that there exist markings $M_1, \dots, M_n$ satisfying:

$$M_0^{\text{ON}}[U_1\rangle_{\text{ON}} M_1, \dots, M_{n-1}[U_n\rangle_{\text{ON}} M_n \ . \tag{2.1}$$

The *reachable markings* of ON are defined as the smallest (w.r.t. $\subseteq$) set *reach*(ON) containing M_0^{ON} and such that if there is a marking $M \in$ *reach*(ON) and $M[U\rangle_{\text{ON}} M'$, for a step U and a marking M', then $M' \in$ *reach*(ON).

Proposition 2.1. *(see [40]) Given a step sequence of* ON *defined by Definition 2.1, we have that:*

1. *If $i \neq j$ then $U_i \cap U_j = \varnothing$, i.e. no event occurs more than once.*
2. *There is a step sequence involving all the events in E.*
3. *$M_{Fin} = M_n$ iff $E = U_1 \cup \dots \cup U_n$, i.e. each event of E has occurred.*
4. *If $i \geq j$ then $(U_i \times U_j) \cap \prec^+ = \varnothing$, i.e. the causal predecessors of an event can never be executed after or together with that event.*

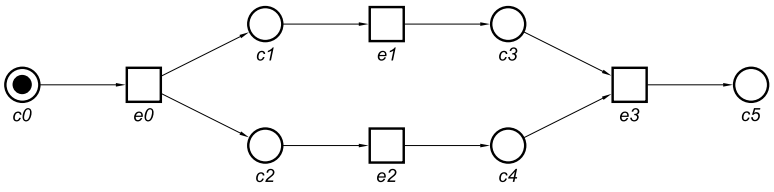


Figure 2.1: An occurrence net.

Figure 2.1 shows an occurrence net whose conditions are represented by circles and events are represented by boxes. The initial marking is $\{c_0\}$ which is indicated by showing a token inside the starting condition. There are five cuts in the ON:

$\{c_0\}$, $\{c_1, c_2\}$, $\{c_1, c_4\}$, $\{c_3, c_4\}$, $\{c_2, c_3\}$, and $\{c_5\}$. Thus, a possible step sequence is $\lambda = \{e_0\}\{e_1, e_2\}\{e_3\}$. One can observe that the corresponding sequence of markings starts with $M_0 = \{c_0\}$ and ends with $M_{Fin} = \{c_5\}$.

A *phase* of ON is a non-empty set of conditions $\pi \subseteq C$ such that the set $Min_\pi \subseteq \pi$ of the minimal conditions of π (w.r.t. F^+) is a cut; the set $Max_\pi \subseteq \pi$ of the maximal conditions of π (w.r.t. F^+) is a cut; and π comprises all conditions $c \in C$ for which there are $b \in Min_\pi$ and $d \in Max_\pi$ satisfying $(b, c) \in F^*$ and $(c, d) \in F^*$. Moreover, a *phase decomposition* of ON is a sequence $\pi_1 \ldots \pi_m$ of phases of the occurrence net such that $M_0 = Min_{\pi_1}$, $Max_{\pi_i} = Min_{\pi_{i+1}}$ (for $i \leq m-1$), and $Max_{\pi_m} = M_{Fin}$. A *block* of ON is a non-empty set $B \in (C \cup E)$ such that $B \cap C = {}^\bullet(B \cap E) \cap (B \cap E)^\bullet$ and $({}^\bullet B \setminus B) \times (B^\bullet \setminus B) \subseteq \prec$.

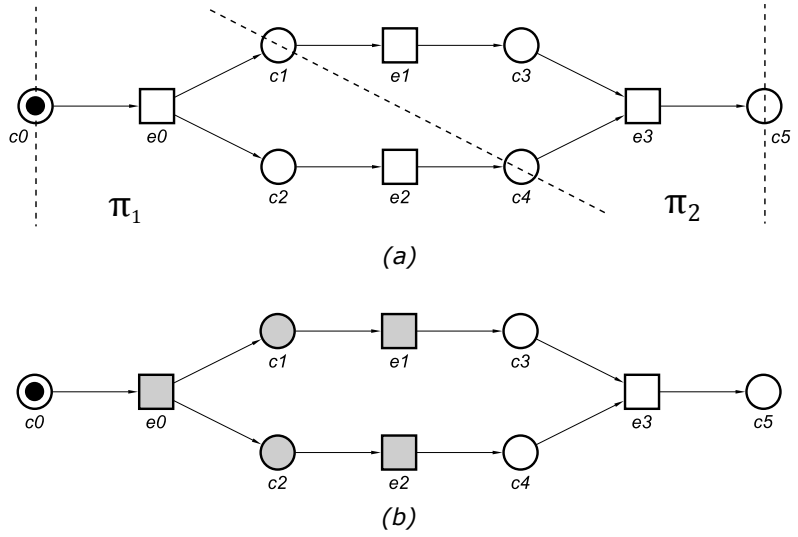


Figure 2.2: The occurrence net shown in Figure 2.1 with (a) phase decompositions, and (b) a block.

Each phase is a fragment of an ON beginning with a cut and ending with a cut which follows it in the causal sense, including all the conditions occurring between these cuts. A *phase decomposition* is a sequence of phases from the initial state to the final state, and whenever one phase ends, its maximal cut is the starting point of the successive one (minimal cut). The notion of a block represents a contiguous fragment of activity within the behaviour represented by an ON. Intuitively, a block is a set of nodes in which all conditions are internal to the block, that is,

the input and output events of each condition must belong to the block; and all block ending events are causally linked to all starting events. Note that a single event can constitute a block. As an example, the ON in Figure 2.2 (a) has been divided into two phases by the three depicted cuts. The corresponding phase decomposition is $\pi_1\pi_2 = \{c_0, c_1, c_2, c_4\}\{c_1, c_3, c_4, c_5\}$. Figure 2.2 (b) shows a block $B = \{e_0, c_1, c_2, e_1, e_2\}$ with its nodes shadowed.

2.3 Communication Structured Occurrence Nets

Communication structured occurrence nets (CSONs) are able to portray different kinds of communication between separate systems. It will usually be the case that, if an occurrence net in fact represents the combined activity of several interacting systems, it will be beneficial to split the model into a set of component occurrence nets, and to create specific devices to represent communication between the component occurrence nets (subsystems). In the model we are interested in communication can be synchronous or asynchronous.

A CSON is composed of a set of component ONs representing separate subsystems. When it is determined that there is a potential for an interaction between subsystems, an asynchronous or synchronous communication link can be made between events in different ONs via a special element called a *channel place*. Communication relations were represented by a directed dashed line between two events in the original definition of CSONs [40]. The notion of a channel place, which was introduced later [28], is a more flexible means of representing such relations.

Two events involved in a synchronous communication link must be executed simultaneously. On the other hand, events involved in an asynchronous communication can be either executed simultaneously or one after the other.

Definition 2.2 (CSON). *A* communication structured occurrence net *(*CSON*) is a tuple*

$$\text{CSON} = (\text{ON}_1, \ldots, \text{ON}_k, Q, W)$$

such that $\text{ON}_i = (C_i, E_i, F_i)$ *for* $i = 1, \ldots, k$ *are occurrence nets (below we denote by* $\mathbf{C} = \bigcup_{i=1}^{k} C_i$, $\mathbf{E} = \bigcup_{i=1}^{k} E_i$ *and* $\mathbf{F} = \bigcup_{i=1}^{k} F_i$ *their conditions, events and arcs);*

Q is a set of channel places; and $W \subseteq (\boldsymbol{E} \times Q) \cup (Q \times \boldsymbol{E})$ are the arcs between the channel places and events. It is further assumed that:

1. *The ON_is and Q are mutually disjoint.*

2. *The sets of input and output events of $q \in Q$,*

$$^\bullet q = \{e \in \boldsymbol{E} \mid (e, q) \in W\} \quad \textit{and} \quad q^\bullet = \{e \in \boldsymbol{E} \mid (q, e) \in W\} \,,$$

   belong to distinct component ON_i's; and moreover, $|^\bullet q| = 1$ and $|q^\bullet| \leq 1$.

3. *The relation*

$$(\sqsubset \cup \prec)^* \circ \prec \circ (\prec \cup \sqsubset)^* \tag{2.2}$$

   over $\boldsymbol{E}$ is irreflexive, where:

   - *$e \prec f$ if there is $c \in \boldsymbol{C}$ with $c \in e^\bullet \cap {}^\bullet f$;*
   - *$e \sqsubset f$ if there is $q \in Q$ with $q \in e^\bullet \cap {}^\bullet f$.*

In Definition 2.2(2.2), we use the relation $\sqsubset$ (*weak causality*) to represent a/synchronous communication between two events (see [33, 48]). Intuitively, the original causality relation $\prec$ represents the 'earlier than' relationship of the events, and $\sqsubset$ represents the 'not later than' relationship. The input and output sets of a node in a CSON are also extended to include channel places with the relation W. In order to ensure that the resulting causal dependencies remain consistent, in Definition 2.2(2.2) we require the acyclicity of not only each component occurrence net but also any path involving both $\sqsubset$ and $\prec$.

The initial marking M_0^{CSON} of a CSON is the union of $M_0^{\text{ON}_1}, \ldots, M_0^{\text{ON}_k}$ (assuming that there are no channel places in M_0^{CSON}). The final marking M_{Fin}^{CSON} of a CSON is the union of $M_{Fin}^{\text{ON}_1}, \ldots, M_{Fin}^{\text{ON}_k}$. In general, a marking in a CSON is a set of conditions and channel places. A step in a CSON is a set of events which may come from one of more component occurrence nets.

Definition 2.3 (CSON firing rule)**.** *Let* CSON *be a communication structured occurrence net as in Definition 2.2, M be a marking, and U be a step of the* CSON*.*

1. *U is* CSON-enabled *at M if $(^\bullet U \setminus U^\bullet) \subseteq M$.*

2. *If U is* CSON*-enabled at M, then U can be* fired *and produce a new marking M' is given by: $M' = (M \cup U^\bullet) \setminus {}^\bullet U$. This is denoted by $M[U\rangle_{\text{CSON}} M'$.*

The *step sequences* and *reachable markings* of CSON are then defined similarly as for an occurrence net.

The firing rule above means that a step U involving synchronous behaviour can use not only the tokens that are already available in channel places at marking M, but also can use the tokens deposited there by events from step U during the execution of U. In this way, events from step U can 'help' each other individually and synchronously pass resources (tokens) among themselves. Thus, in contrast to the step sequence of an occurrence net, where a step consists of a number of enabled events, the execution of a step in a CSON (i.e. $M[U\rangle M'$) may involve synchronous communications, where events execute simultaneously and behave as a transaction. Such a mode of execution provides possibility to execute multiple events in a single step, and therefore is strictly more expressive than that used in ONs,

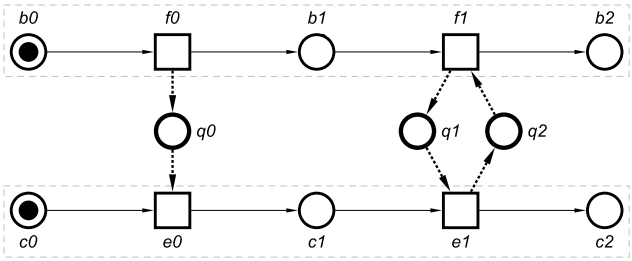


Figure 2.3: A CSON with two interacting occurrence nets.

Figure 2.3 shows a CSON which consists of two interacting occurrence nets connected by three channel places (portrayed graphically by bold circles). The thick dashed lines indicate relation W. The connection between events f_0 and e_0 is an asynchronous communication, which means that e_0 cannot happen before f_0. Events f_1 and e_1 are connected by a pair of empty channel places, q_1 and q_2, forming a cycle. Such a cycle does not violate CSON acyclicity because it involves only weak causality, but the two connected events can only be executed synchronously. The channel places q_1 and q_2 will be filled and emptied synchronously when both

f_1 and e_1 participate in a step being fired. Therefore, a possible step sequence of this CSON is $\lambda = \{f_0\}\{e_0\}\{f_1, e_1\}$. Note that $\lambda' = \{f_0, e_0\}\{f_1, e_1\}$ is also a valid step sequence where f_0 and e_0 execute simultaneously.

Proposition 2.2 ([40]). *Let* CSON *be a communication structured occurrence net as in Definition 2.2, and* $\lambda = U_1 \dots U_n$ *(*$n \geq 0$*) be a step sequence of the* CSON.

1. *If* $i \neq j$ *then* $U_i \cap U_j = \varnothing$*, i.e. no event occurs more than once.*
2. *There is a step sequence involving all the events in* $\mathbf{E}$.
3. *If* $i \geq j$ *then* $(U_i \times U_j) \cap ((\sqsubset \cup \prec)^* \circ \prec \circ (\prec \cup \sqsubset)^*) = \varnothing$*, i.e. the causal predecessors of an event can never be executed after or together with that event.*

2.3.1 Synchronous Cycles

In the following, we introduce the notion of the *sync-cycle*.

Definition 2.4 (sync-cycle). *Let* CSON *be a communication structured occurrence net as in Definition 2.2.*

A sync-cycle *of a* CSON *is a maximal nonempty set of events* $\mathbb{S} \subseteq \mathbf{E}$ *such that for all distinct* $e, f \in \mathbb{S}$*,* $(e, f) \in W^+$*. The set of all sync-cycles of a* CSON *will be denoted by* SC^{CSON}.

A channel place q *is* synchronous *if there exist a sync-cycle* $\mathbb{S} \in SC^{\text{CSON}}$ *such that* $q \in \mathbb{S}^\bullet \cap {}^\bullet\mathbb{S}$*. Otherwise,* q *is* asynchronous.

The notion of a sync-cycle captures the idea of a synchronous communication involving a maximal number of sub-systems. Its events graphically form a weak causal cycle connected by synchronous channel places.

We first show that there is no reachable marking which includes synchronous channel places.

Proposition 2.3. *Let* CSON *be a communication structured occurrence net as in Definition 2.2, and* Q^s *be its synchronous channel places. Then* $Q^s \cap M = \varnothing$*, for every reachable marking* $M \in reach(\text{CSON})$.

Proof. By the definition of CSON, we have $Q^s \cap M_0^{\text{CSON}} = \varnothing$. Hence, it suffices to show that if $M \in reach(\text{CSON})$ is such that $Q^s \cap M = \varnothing$, and M' and U are such that $M[U\rangle_{\text{CSON}} M'$, then $Q^s \cap M' = \varnothing$.

Suppose $q \in Q^s$ is such that $q \in M'$. Then there is a sync-cycle $\mathbb{S} \in SC^{\text{CSON}}$ and $e, f \in \mathbb{S}$ such that $q \in e^\bullet \cap {}^\bullet f$. By Definition 2.4, there is a sequence $e_0 q_0 e_1 \dots e_{m-1} q_{m-1} e_m$ such that $e_0 = f$, $e_m = e$ and $q_i \in e_i^\bullet \cap {}^\bullet e_{i+1}$, for $i < m$. We also recall that $|q_i^\bullet| = |{}^\bullet q_i|$, for every $i < m$.

Since $q \in M'$, we have $f = e_0 \notin U$. Hence, since $q_0 \notin M$, we have $e_1 \notin U$. By proceeding m times in this way, we obtain $e_m = e \notin U$. This and $q \notin M$ means that $q \notin M'$, a contradiction. As a result, $Q^s \cap M' = \varnothing$. □

The next result implies that all events in a sync-cycle are always enabled and fired simultaneously.

Proposition 2.4. *Let* CSON *be a communication structured occurrence net as in Definition 2.2,* $\mathbb{S} \in SC^{\text{CSON}}$ *be a sync-cycle, and* U *be a step enabled at a reachable marking* $M \in reach(\text{CSON})$*. Then* $e \in U \Leftrightarrow f \in U$*, for all* $e, f \in \mathbb{S}$*.*

Proof. Follows from Proposition 2.3, Definition 2.4, and the definition of an enabled step. □

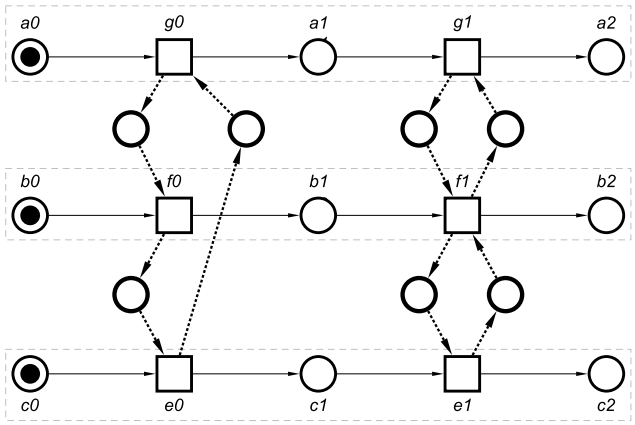


Figure 2.4: Three occurrence nets that are synchronous with each other.

Consider the CSON in Figure 2.4. One can observe there are two sync-cycles. Sync-cycle $\{e_0, f_0, g_0\}$ in fact is composed of three asynchronous communications.

The communication in any of two events is asynchronous. However, all three events can only fire in a single step. The run for the sync-cycle $\{e_1, f_1, g_1\}$ is also simultaneous. Although it consists of two 'component' synchronous interactions, it is impossible to fire either of them individually. To simplify the representation, in the rest of the thesis we will use bold dashed lines without arcs to indicate any synchronous cycles linked directly between two events.

The following proposition addresses the minimal firing concerning asynchronous communication.

Proposition 2.5. *Let* CSON *be a communication structured occurrence net as in Definition 2.2, q be an asynchronous channel place, e and f be the input and output events of q respectively, M be a reachable marking, and U be a* CSON*-enabled step at M. Then $f \in U$ and $q \notin M$ implies $e \in U$.*

Proof. Suppose that $e \notin U$. From Definition 2.3(1), $f \in U$ implies $q \in M$, a contradiction. □

In Figure 2.3, if e_0 is in U and q_0 is not marked, then the occurrences of e_0 and f_0 must happen together.

2.3.2 Compute CSON-enabled Steps

A *token-player* simulation algorithm for CSON can be simply described by a finite process starting from an initial marking and terminating at a final marking, where at each iteration a set of enabled events are determined and fired in order to change the marking. The main computational problem of CSON simulation is to identify the enabled step U in each iteration. In this section, we propose an approach that can be used to efficiently compute the CSON-enabled steps at a given marking.

The procedure is given in Algorithm 1. The idea is to first compute a step U including all the ON-enabled events of CSON in order to narrow down the size of the search, and then to remove the events which do not meet the CSON-enabled requirement from U.

Unlike the execution of a standard occurrence net, where a step sequence can be composed of a sequence of single event firings, in CSON there may exist *mini-*

Algorithm 1 (Computing a CSON-enabled step)

Inputs:

CSON — communication structured occurrence net

M — current marking

Output:

U — a step CSON-enabled at M

```
1:  U := ∅
2:  Del := ∅                                              // deleted events
3:  for all e ∈ E do
4:    if •e ⊆ M then                                      // e is ON-enabled
5:      add e to U
6:  for all e ∈ U do
7:    if e ∉ Del then
8:      min := minParallel(e)                             // minimal parallel firings of e
9:      for all f ∈ min do
10:       if f ∉ U then      // minimal parallel firings of e is not ON-enabled
11:         add all events in min to Del
12:         break
13: U := U \ Del

14: function minParallel(input: e)
15: Result := ∅
16: mark e visited
17: add e to Result
18: for all g such that g ⊏ e do
19:   if g is unvisited and q ∉ M, where q ∈ g• ∩ •e then
20:     add all events in minParallel(g) to Result
21: return Result
```

mal parallel firings for an event, where one enabled event implies that all events in the minimal parallel firings are enabled as well. Both $\{f_1, e_1\}$ in Figure 2.3 and $\{e_0, f_0, g_0\}$ in Figure 2.4 are such steps because of their synchronised behaviour. Note that such minimal parallel firing can involve either synchronous (see Proposition 2.4) or asynchronous (see Proposition 2.5) communication. Therefore, in the algorithm it is not possible to consider only the enabling for a single event. Instead, all its minimal parallel firings should be considered in the computation.

The pseudocode for computing the minimal parallel firings of a given event is

presented in the function *minParallel*. The function uses a working list, *Result*, initialized to the given event. Then it recursively visits the weak causal predecessors of the node in the list. The predecessor can be added to the working list if it is unvisited and the channel place between the two events is unmarked.

2.4 Behavioural Structured Occurrence Nets

Behavioural structured occurrence nets (BSONs) allow the activity of an evolving system to be modelled. They use a two-level view to represent an execution history, with the lower level providing details of its behaviours during the different evolution stages represented in the upper level view. Thus a BSON gives information about the evolution of an individual system, and the phases of the overall activity are used to represent each successive stage of the evolution of this system. Figure 2.5 shows a simple example of a BSON in a (off-line) system update. The upper level represents a version change caused by an update event. The lower level provides the detailed behaviour of the system before and after the update. The dashed lines between the two levels are used to capture the relevant relationships between the two types of behaviours. (The update portrayed is termed "offline", in contrast to an online update, such as would be exemplified in the figure if the final state of ON_1 were also the initial state of ON_2.)

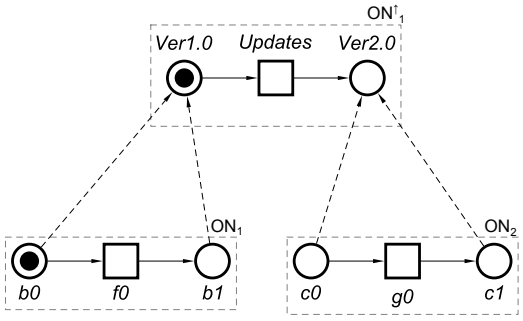


Figure 2.5: A BSON example portraying a system (off-line) update.

Before formalizing the model, we first recall two relations in CSON which extend the definitions of $pre(x)$ (i.e. ${}^\bullet x$) and $post(x)$ (i.e. $x^\bullet$). Given a CSON as in Definition 2.2 and $e \in \mathbf{E}$ as an event, the sets $Pre(e)$ and $Post(e)$ respectively

comprise all conditions $c \in \mathbf{C}$ satisfying $(c, e) \in \mathbf{F} \circ \sqsubset^*$ and $(e, c) \in \sqsubset^* \circ \mathbf{F}$. Intuitively, the new relations capture weak causal chains passing through the events in different occurrence nets. For example, the new relationships in Figure 2.3 are:

$$\begin{array}{lllllllll} Pre(f_0) & = & \{b_0\} & Pre(f_1) & = & \{b_1, c_1\} & Pre(e_0) & = & \{b_0, c_0\} \\ Pre(e_1) & = & \{b_1, c_1\} & Post(f_0) & = & \{c_1, b_1\} & Post(f_1) & = & \{b_2, c_2\} \\ Post(e_0) & = & \{c_1\} & Post(e_1) & = & \{b_2, c_2\}\ . & & & \end{array}$$

We now introduce the concept of a BSON by using the notions above, and by generalising the definition of [40]. Below we assume that an occurrence net ON is *line-like* if $|M_0^{\text{ON}}| = 1$ and $|{}^\bullet e| = |e^\bullet| = 1$, for every event e. Such an occurrence net can be represented in a unique way by a chain $\xi_{\text{ON}} = c_1 e_1 \ldots e_{l-1} c_l$ of alternating (all) conditions and (all) events satisfying ${}^\bullet e_i = \{c_i\}$ and $e_i^\bullet = \{c_{i+1}\}$, for every $i < l$.

Definition 2.5 (BSON). *Let* CSON *be a communication structured occurrence net as in Definition 2.2, and* $\text{CSON}^\uparrow = (\text{ON}_1^\uparrow, \ldots, \text{ON}_m^\uparrow, Q^\uparrow, W^\uparrow)$ *be another (disjoint) communication structured occurrence net such that* $\text{ON}_i^\uparrow = (C_i^\uparrow, E_i^\uparrow, F_i^\uparrow)$ *is line-like, for* $i \leq m$. *Moreover, let* $\mathbf{C}^\uparrow = \bigcup_{i=1}^m C_i^\uparrow$, $\mathbf{E}^\uparrow = \bigcup_{i=1}^m E_i^\uparrow$, *and* $\mathbf{F}^\uparrow = \bigcup_{i=1}^m F_i^\uparrow$.

A behavioural structured occurrence net *(or* BSON*) is a tuple*

$$\text{BSON} = (\text{CSON}, \text{CSON}^\uparrow, \beta)$$

such that $\beta \subseteq \mathbf{C} \times \mathbf{C}^\uparrow$.
It is assumed that the following hold:

1. *For every* ON_i, *there exists exactly one* $\text{ON}_j^\uparrow$ *satisfying* $\beta(C_i) \cap C_j^\uparrow \neq \varnothing$.

2. *For every* $\text{ON}_j^\uparrow$ *represented by a chain* $\xi_{\text{ON}_j^\uparrow} = c_1 e_1 \ldots e_{l-1} c_l$, *the sequence* $\pi_1 \pi_2 \ldots \pi_l = \beta^{-1}(c_1)\beta^{-1}(c_1)\ldots\beta^{-1}(c_l)$ *is a concatenation of phase decompositions of different occurrence nets in* CSON. *We also denote, for all* c_j *and* e_j *occurring in the chain* $\xi_{\text{ON}_j^\uparrow}$, $\pi(c_j) = \pi_j$, *and*

$$causal(e_j) = pre(Max_{\beta^{-1}(Pre(e_j))}) \times \{e_j\} \cup \{e_j\} \times post(Min_{\beta^{-1}(Post(e_j))})$$

3. *The relation*

$$(\sqsubset \cup \prec \cup \lhd)^* \circ (\prec \cup \lhd) \circ (\prec \cup \sqsubset \cup \lhd)^*$$

over $\mathbf{E} \cup \mathbf{E}^\uparrow$ *is irreflexive, where:*

- $e \prec f$ *if there is* $c \in \mathbf{C} \cup \mathbf{C}^{\uparrow}$ *with* $c \in e^{\bullet} \cap {}^{\bullet}f$*;*
- $e \sqsubset f$ *if there is* $q \in Q \cup Q^{\uparrow}$ *with* $q \in e^{\bullet} \cap {}^{\bullet}f$*; and*
- $e \lhd f$ *if* $(e, f) \in \bigcup_{e' \in \mathbf{E}^{\uparrow}} causal(e')$.

The initial marking M_0^{BSON} *of* BSON *is the initial marking of the* CSON$^{\uparrow}$ *together with the initial markings of all the* ON$_i$*s such that* $\beta(M_0^{\text{ON}_i}) \cap M_0^{\text{CSON}^{\uparrow}} \neq \varnothing$*. The final marking* M_{Fin}^{BSON} *of the* BSON *is the final marking of the* CSON$^{\uparrow}$ *together with the final markings of all the* ON$_i$*s such that* $\beta(M_{Fin}^{\text{ON}_i}) \cap M_{Fin}^{\text{CSON}^{\uparrow}} \neq \varnothing$.

A BSON consists of two CSONs linked by the behavioural relation β. In CSON$^{\uparrow}$, where all the component occurrence nets are line-like and all the conditions are the end points of β, the upper level represents the evolution of a system. The CSON is the lower level net representing the detailed behaviour of the system. The behavioural relation β connecting the two levels is used to provide dependencies between the evolution and detailed information of the system. In such a structured view, the upper part provides the necessary information for the desired sequencing of the occurrence nets (which are called phases) in the lower part. Definition 2.5(1) implies that each phase points to exactly one condition of the upper level ON, and Definition 2.5(2) means that each upper level condition maps to a single phase of a lower level ON. The ordering of the upper level conditions must match that of the phase decompositions of the lower level ON. The term $causal(e)$ [1] captures some new causal dependencies between events coming from both levels of the BSON. Intuitively, it represents the 'happened before' relationship of the events. In Definition 2.5(3), it is required that the new dependencies, together with the communication (i.e. $\sqsubset$) and ordinary causal relations (i.e. $\prec$) which are already present in the model are acyclic.

Note that in a BSON, the initial marking of a lower level ON may not belong to the initial marking of the BSON. Such a net may be 'waiting' for the firing of some events in other ONs. For the BSON in Figure 2.5, $\{c_0\}$ is the initial marking of ON$_2$ but it is not the initial marking of the BSON. $\{c_0\}$ can be reached only if the 'update' event has happened.

[1] We write $before(e)$ instead of $causal(e)$ [13], and the change to $causal(e)$ is for greater clarity.

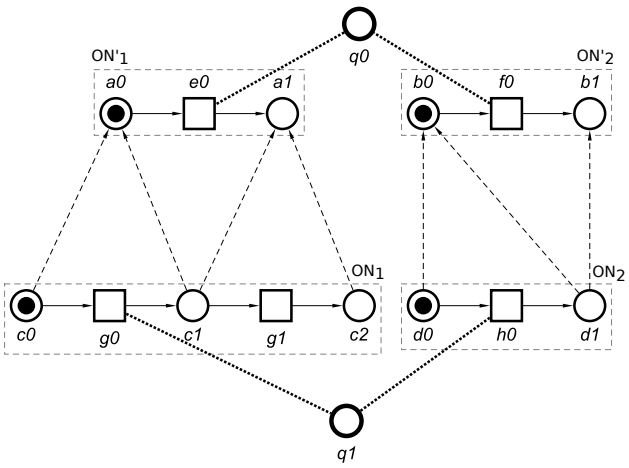


Figure 2.6: A BSON with two upper level ONs and two lower level ONs. To simplify the representation, here we use bold dashed lines without arcs to indicate synchronous cycles.

Figure 2.6 shows an example of a BSON involving synchronous communications in both levels. The lower level CSON consists of two interacting systems, ON_1 and ON_2. Information about their evolution is provided in the upper level by $\text{ON}_1^{\uparrow}$ and $\text{ON}_2^{\uparrow}$ respectively. The initial marking is $M_0^{\text{BSON}} = \{a_0, b_0, c_0, d_0\}$, and the final marking is $M_{Fin}^{\text{BSON}} = \{a_1, b_1, c_2, d_1\}$ The related phase decompositions are as follows:

$$\begin{array}{lllllllll} \beta^{-1}(C_1) & = & \beta^{-1}(a_0) & \beta^{-1}(a_1) & = & \pi_1 & \pi_2 & = & \{c_0, c_1\} \quad \{c_1, c_2\} \\ \beta^{-1}(C_2) & = & \beta^{-1}(b_0) & \beta^{-1}(b_1) & = & \pi_3 & \pi_4 & = & \{d_0, d_1\} \quad \{d_1\} \end{array}$$

where C_1 and C_2 are sets of conditions in $\text{ON}_1^{\uparrow}$ and $\text{ON}_2^{\uparrow}$ respectively. One can observe that the succession of the conditions in each upper ON corresponds to a valid phase decomposition in the lower ONs. For the phases $\pi_1\pi_2$ in $\text{ON}_1^{\uparrow}$, we have $Min_{\pi_1} = \{c_0\}, Max_{\pi_1} = Min_{\pi_2} = \{c_1\}$ and $Max_{\pi_2} = \{c_2\}$. Using the phase information and the relations captured by CSON, we obtain the $causal(e)$ relations

of two upper level events e_0 and f_0, in the following way:

$$\begin{aligned}
causal(e_0) &= pre(Max_{\beta^{-1}(Pre(e_0))}) \times \{e_0\} \quad \cup \quad \{e_0\} \times post(Min_{\beta^{-1}(Post(e_0))}) \\
&= (pre(Max_{\beta^{-1}(a_0)}) \cup pre(Max_{\beta^{-1}(b_0)})) \times \{e_0\} \quad \cup \\
&\quad \{e_0\} \times (post(Min_{\beta^{-1}(a_1)}) \cup post(Min_{\beta^{-1}(b_1)})) \\
&= pre(Max_{\{c_0,c_1\}}) \cup pre(Max_{\{d_0,d_1\}})) \times \{e_0\} \quad \cup \\
&\quad \{e_0\} \times (post(Min_{\{c_1,c_2\}}) \cup post(Min_{\{d_1\}})) \\
&= (pre(c_1) \cup pre(d_1)) \times \{e_0\} \quad \cup \quad \{e_0\} \times (post(c_1) \cup post(d_1)) \\
&= \{g_0, h_0\} \times \{e_0\} \quad \cup \quad \{e_0\} \times \{g_1\} \\
&= \{(g_0, e_0), (h_0, e_0), (e_0, g_1)\} \\
causal(f_0) &= \{g_0, h_0\} \times \{f_0\} \quad \cup \quad \{f_0\} \times \{g_1\} \\
&= \{(g_0, f_0), (h_0, f_0), (f_0, g_1)\} \ .
\end{aligned}$$

Thus, we have the following causal relationships (over events) for this BSON:

$$\begin{aligned}
causality &: \prec = \{(g_0, g_1)\} \\
weak\ causality &: \sqsubset = \{(e_0, f_0), (f_0, e_0), (g_0, h_0), (h_0, g_0)\} \\
causal &: \lhd = \{(g_0, e_0), (e_0, g_1), (h_0, e_0), (g_0, f_0), (f_0, g_1), (h_0, f_0)\} \ .
\end{aligned}$$

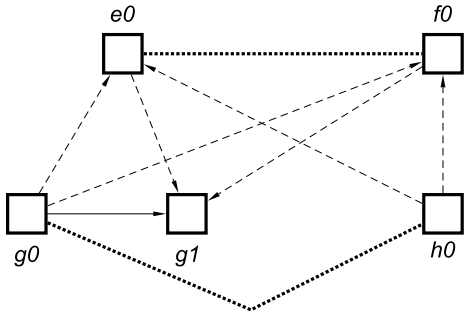


Figure 2.7: Three types of relationships over events of Figure 2.6.

Figure 2.7 illustrates the above relationships diagrammatically. The solid lines represent the causal relations $\prec$; the bold dashed lines indicate the dependencies $\sqsubset$ captured by a/synchronous communications; and the dashed lines represent

$\lhd$ relations. Intuitively, the meaning of $\lhd$ is that, for example, g_0 and h_0 must happen before e_0, while g_1 must happen after e_0, since the former two events belongs to the pre-phase of e_0 while the latter one belong to the post-phase of e_0. We can observe from the diagram that this BSON satisfies the acyclicity conditions described in Definition 2.5(3).

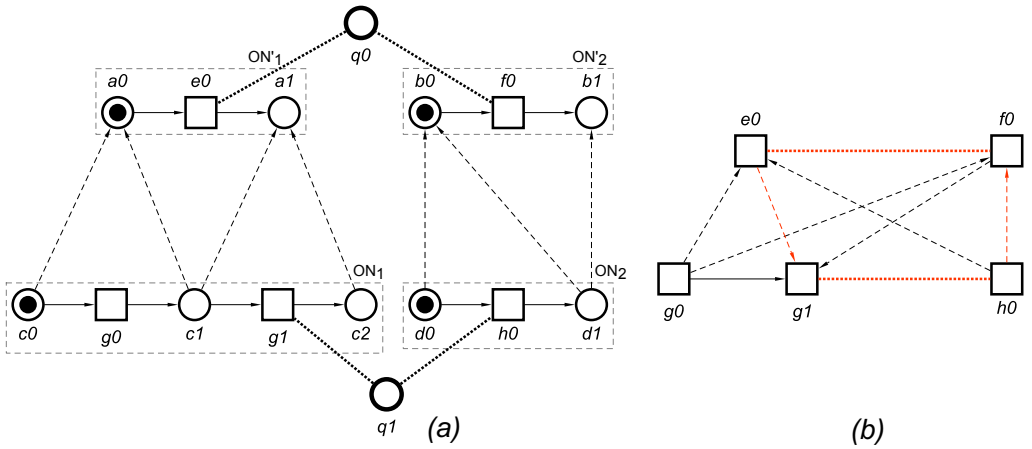


Figure 2.8: (a) An invalid BSON which involves a causal cycle; (b)three types of relationships over events of (a).

Remark 2.1. *A causal cycle in a* BSON *in general involves occurrence nets in both levels. For instance, the model in Figure 2.8(a) is as in Figure 2.6(a) except for the synchronous communication in the lower level between g_1 and h_0. Such a model is not a valid* BSON *structure. The events $\{e_0, g_1, h_0, f_0\}$ form a causal cycle (see the relationships portrayed in Figure 2.8(b)). It indicates that e_0 happens before g_1, and h_0 happens before f_0, but $\{e_0, f_0\}$ and $\{g_0, h_0\}$ must execute simultaneously due to synchronisation. As a result, none of them can ever be executed.* □

Next we define the BSON firing rule. Below we assume that if $e \in E$ is an event in an upper level $\text{ON}^\uparrow$ (i.e. $\text{ON}^\uparrow = (C, E, F)$), then $\pi({}^\bullet e) = \beta^{-1}({}^\bullet e \cap C)$ is the phase of the input condition of e, and $\pi(e^\bullet) = \beta^{-1}(e^\bullet \cap C)$ is the phase of the output condition of e. The markings and steps of the BSON are defined similarly to those for a CSON.

Definition 2.6 (BSON firing rule)**.** *Let* BSON *be as in Definition 2.5. $M \subseteq \mathbf{C} \cup \mathbf{C}^{\uparrow}$ be a marking, and $U \subseteq \mathbf{E} \cup \mathbf{E}^{\uparrow}$ be a step of* BSON*.*

1. *U is* BSON-enabled *at M if*
   - $({}^{\bullet}U \setminus U^{\bullet}) \subseteq M$*;*
   - $Max_{\pi({}^{\bullet}e)} \subseteq M$*, for every $e \in \mathbf{E}^{\uparrow} \cap U$;*
   - $\beta({}^{\bullet}e') \cap \beta(e'^{\bullet}) \in M$*, for every $e' \in \mathbf{E} \cap U$.*
2. *If U is* BSON*-enabled at M, then U can be* fired *and produce a marking M' given by:*
$$M' = (M \setminus ({}^{\bullet}U \cup Max_{\pi({}^{\bullet}U)})) \cup U^{\bullet} \cup Min_{\pi(U^{\bullet})}$$
where $Max_{\pi({}^{\bullet}U)} = \bigcup_{e \in U} Max_{\pi({}^{\bullet}e)}$ and $Min_{\pi(U^{\bullet})} = \bigcup_{e \in U} Min_{\pi(e^{\bullet})}$. This is denoted by $M[U\rangle_{\text{BSON}} M'$.

The definitions of *step sequences* and *reachable markings* of BSON are similar to those for CSONs.

The firing rule above takes care of the marking moving across different phases. Given a marking, there are three requirements to decide whether or not a step is BSON-enabled: (i) it is CSON-enabled (Definition 2.3); (ii) for each upper level event in U, the maximal conditions in the phase of its input condition are in the current marking; and (iii) for each lower level event in U, its corresponding upper level condition is in the current marking.

For example, $\{g_0, h_0\}\{e_0, f_0\}\{g_1\}$ is a possible step sequence of the BSON in Figure 2.6. The only step U_1 enabled at the initial marking $\{a_0, b_0, c_0, d_0\}$ is $U_1 = \{g_0, h_0\}$ since it is CSON-enabled, and the corresponding upper level conditions (a_0 and b_0) are also marked. The firing of U_1 changes the marking to $\{a_0, b_0, c_1, d_1\}$ which enables the step $U_2 = \{e_0, f_0\}$ (note that the conditions in $Max_{\pi}({}^{\bullet}e_0) = \{c_1\}$ and $Max_{\pi}({}^{\bullet}f_0) = \{d_1\}$ are marked). The firing of U_2 produces $\{a_1, b_1, c_1, d_1\}$ and also enables $U_3 = \{g_1\}$ which produces the final marking $\{a_1, b_1, c_2, d_1\}$.

Proposition 2.6 ([40])**.** *Given a step sequence of* BSON *$\lambda^{\text{BSON}} = U_1 \dots U_n$ ($n \geq 0$), we have that:*

1. *If $i \neq j$ then $U_i \cap U_j = \varnothing$, i.e. no event occurs more than once.*

2. *There is a step sequence involving all the events in* $\mathbf{E}$.

3. *If* $i \geq j$ *then* $(U_i \times U_j) \cap ((\sqsubset \cup \prec \cup \lhd)^* \circ (\prec \cup \lhd) \circ (\prec \cup \sqsubset \cup \lhd)^*) = \varnothing$, *i.e. the causal predecessors of an event can never be executed after or together with that event.*

Note that Proposition 2.6(3) states the consistency between the temporal ordering of events involved in a step sequence and the relations provided by the BSON.

2.5 Temporal Structured Occurrence Nets

Atomicity has long been recognized as an important concept in concurrent and distributed system design. The concept has been studied before by various researchers in both theory and practical areas. For example, [18] formally models the atomicity in asynchronous systems; [50] discusses the use of atomic action schemes in designing practical applications. The effective use of atomicity can significantly reduce the cognitive complexity of structured system behaviours, where one can define "abbreviated" parts of systems where their detailed behaviours are collapsed and hidden in a lower level and replaced by simple symbols in an upper level. Then the analysis of behaviour can be performed efficiently at the upper level of abstraction and, after identifying a problem, mapping it to a corresponding behaviour at the lower level in order to continue the analysis there.

In this section, we introduce the atomicity concept to the type of SONs termed temporal structured occurrence nets (TSON). The idea behind the TSON is to use the notion of the block to define an "abbreviated" parts of an occurrence net representing atomic actions. Figure 2.9(a) depicts the TSON view of system abbreviation, i.e. of that part of the behaviour in the lower level ON that is hidden by the temporal abstraction in the upper level ON. Figure 2.9(b), (c) show this alternative representation of Figure 2.9(a), where the collapsed behaviour part can be replaced by simple event symbols.

Definition 2.7 (TSON)**.** *Let* CSON *be a communication structured occurrence net as in Definition 2.2, and* $\text{CSON}^{\downarrow} = (\text{ON}_1^{\downarrow}, \ldots^{\downarrow}, \text{ON}_m^{\downarrow}, Q^{\downarrow}, W^{\downarrow})$ *be another (disjoint)*

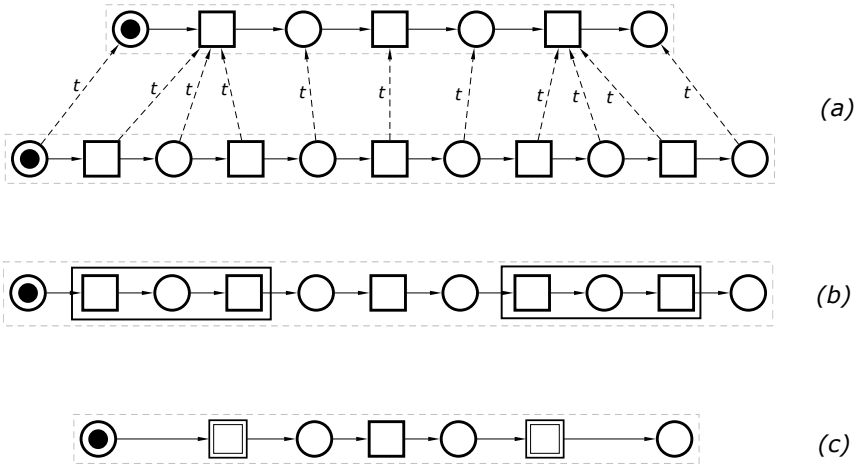


Figure 2.9: (a) A TSON where t-labelled edges indicate a mapping from lower to upper level, while (b) and (c) are the alternative representation of (a).

communication structured occurrence net, and let $\mathbf{C}^{\downarrow} = \bigcup_{i=1}^{m} C_i^{\downarrow}$, $\mathbf{E}^{\downarrow} = \bigcup_{i=1}^{m} E_i^{\downarrow}$, *and* $\mathbf{F}^{\downarrow} = \bigcup_{i=1}^{m} F_i^{\downarrow}$.

A temporal structured occurrence net *(or* TSON*) is a tuple*

$$\text{TSON} = (\text{CSON}, \text{CSON}^{\downarrow}, \tau)$$

such that $\tau : \mathbf{C}^{\downarrow} \cup \mathbf{E}^{\downarrow} \cup Q^{\downarrow} \rightarrow \mathbf{C} \cup \mathbf{E} \cup Q$ *is a mapping from the nodes of* CSON$^{\downarrow}$ *to the nodes of* CSON.
It is assumed that the following hold:

1. $\tau(C_i^{\downarrow} \cup E_i^{\downarrow} \cup Q^{\downarrow}) = C_i \cup E_i \cup Q, \tau^{-1}(C_i) \subseteq C_i^{\downarrow}, \tau(E_i^{\downarrow}) = E_i$, *and* $\tau^{-1}(Q) = Q_i^{\downarrow}$.
2. *for all* $e \in \mathbf{E}, \tau^{-1}(e)$ *are disjoint blocks of the component* ON*s of* CSON *(note that* $\tau^{-1}(e)$ *cannot be empty).*
3. *for all* $c \in \mathbf{C}, |\tau^{-1}(c)| = 1$, *and for all* $q \in Q, |\tau^{-1}(q)| = 1$.
4. $\mathbf{F} = \{(x, y) \in (\mathbf{C} \times \mathbf{E}) \cup (\mathbf{E} \times \mathbf{C}) | (\tau^{-1}(x) \times \tau^{-1}(y)) \cap \mathbf{F}^{\downarrow} \neq \varnothing\}$.
5. $W = \{(u, v) \in (\mathbf{E} \times Q) \cup (Q \times \mathbf{E}) | (\tau^{-1}(u) \times \tau^{-1}(v)) \cap W^{\downarrow} \neq \varnothing\}$.

The above definition slightly modifies Definition 11 in [40] by taking into account the notion of channel place. More specifically, a TSON consists of two CSONs

and a mapping from the lower level CSON to the upper level CSON$^{\uparrow}$ (to simplify the representation, the definition does not cope with the behavioural relation β and the extension to deal with this is left for future research). Definition 2.7(1),(2) describes that all events and some of the conditions in the lower level belong to disjoint blocks, and each such block maps to an event in the upper CSON representing its components collapsed into a single event. Definition 2.7(4),(5) implies that the causal relations between new events and conditions (channel places) which have survived the collapsing are inherited from the events which have been collapsed and the corresponding conditions (channel places) in the lower level. Definition 2.7(3) guarantees the disjoint nature of the two component CSONs by implicitly assuming that the conditions and channel places which survived collapsing are renamed.

The following result shows that any step in the upper level execution can be re-interpreted as a valid step sequence in the lower level net. Therefore, a practical way of using the TSON is to analyse the behaviour at the upper level and, after finding a problem, mapping it to corresponding behaviours at the lower level.

Theorem 2.1 ([40]). *Let* TSON *be the temporal abstraction structured occurrence net as in Definition 2.7, and $U_1 \ldots U_{n-1}U_n$ be a step execution of* CSON. *Then there is a step sequence of* CSON$^{\downarrow}$,

$$U_1^1 \ldots U_1^{m_1} \ldots U_{n-1}^1 \ldots U_{n-1}^{m_{n-1}} U_n^1 \ldots U_n^{m_n},$$

such that $U_i^i \cup \cdots \cup U_i^{m_i} = \tau^{-1}(G_i) \cap \mathbf{E}^{\downarrow}$, for all $i \leq n$.

Remark 2.2. *The collapsing operation in* TSON *is much trickier with occurrence nets that represent asynchronous activity, since there is a need to avoid introducing cycles (see [18]), and the possibility to alter communication types. This is shown in Figure 2.10. The* TSON *initially has two un-collapsed blocks, and collapsing one of the blocks results in the introduction of a causal cycle. However, if one collapses both blocks, then what results is a* TSON *with synchronous communication, and so it is a valid structure.* □

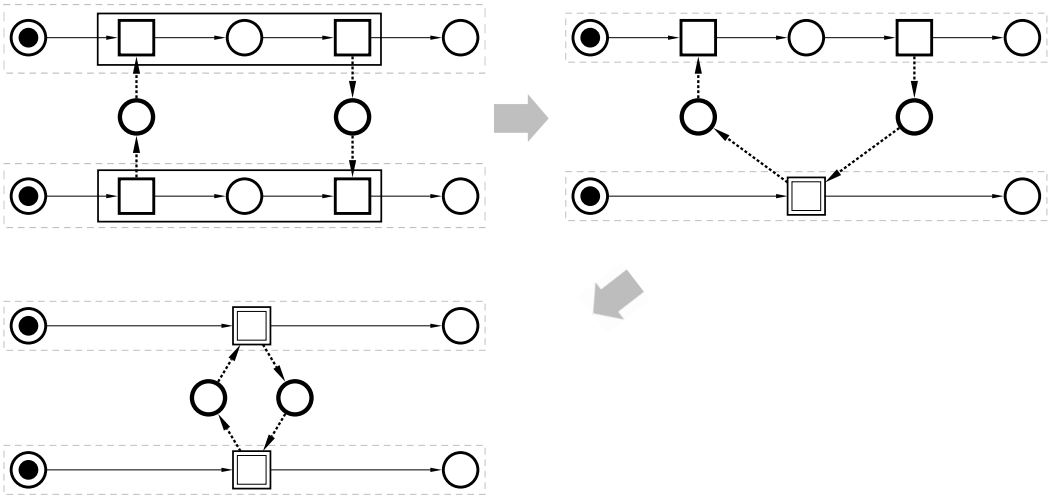


Figure 2.10: A CSON with two (un-collapsed) blocks (top-left); The collapse of one of the blocks results in an invalid CSON (top-right); The collapse of both blocks changes the type of commutation from asynchronous to synchronous (bottom).

2.6 Automated Verification in SONs

Using SONs in complex evolving system design involves steps of modelling, analysis and implementation. The modelling step concerns the use of the appropriate notation and structure to represent system behaviour. The SONs formalism introduced in the previous sections can be directly used for modelling the behaviour of complex evolving systems. The analysis step includes the analysis of behavioural and structural properties. The former refer to the properties which depend on the initial marking, while structural properties are those that depend on the topological structures of the model [57].

In this section, we discuss the two types of properties in SONs and provide algorithms for the related verifications.

2.6.1 Structural Properties

The structural properties in SONs are particularly important due to the various relationships. It is important to verify the correctness of the structure before further analysis (e.g. simulation and behavioural property checking), or otherwise

the results are likely to be incorrect. The verification criteria follow from the formal definitions and properties introduced in this chapter, including:

1. Conflict-freeness: each condition or channel place has at most one input event and at most one output event (see Definition 2.2(2) and the definition of ON in Section 2.2).

2. Component ONs disjointness: the connections between two component ONs are either a/synchronous communications or behavioural relations (see Definitions 2.2(1) and 2.5).

3. Phase decomposition: the maximal and minimal bounds of each phase are cuts, and the back-to-back structure is of a phase decomposition (see the definition of phase in Section 2.2 and Definition 2.5(2)).

4. Behavioural dependency: the ordering of the upper level conditions must match that of the phase decomposition of the lower level ON; the upper level conditions and phases are in a one-to-one relation (see Definitions 2.5(1) and (2)).

5. Block causality: all ending events of a block are causal successors of all starting events (see the definition of block in Section 2.2).

6. Acyclicity: SONs are causal-cycle-free (see the definition of ON in Section 2.2, and Definitions 2.2(3) and 2.5(3)).

As an example, Algorithm 2 carries out SONs acyclicity checking. The verification of such a property in practice comes down to searching strongly connected components (SCC) in a SON model. The idea is to apply Tarjan's algorithm [51] to compute maximal SCCs, and then use a filter to obtain the desired results. The algorithm first converts SON to a graph $G = (V, R)$, where V is the set of nodes including all conditions, events and channel places of the SON, and the set R is the arcs representing all causal relationships and weak causal relationships. The algorithm then computes $causal(e)$ for every upper level event as additional relations for the input graph. The *filter*() function at the end of the algorithm aims to remove all the cycles which only involve weak causality; that is sync-cycles.

Algorithm 2 (SON cycle detection)

Inputs:

SON — structured occurrence net

Output:

Result — causal cycles

```
1: convert SON to G := (V, R).
2: for all e ◁ f do
3:    add (e, f) to R
4: Result := tarjan(G)                    // compute SCCs of G
5: for all SCC ∈ Result do
6:    if |SCC| = 1 then
7:       remove SCC from Result
8:    else if SCC does not on ≺ then
9:       remove SSC from Result
```

2.6.2 Reachability

Reachability is a fundamental basis for studying the behavioural properties of any system. Such analysis establishes whether a given marking, i.e. a set of conditions and/or channel places, can be reached from the initial marking. Various different approaches have been developed to determine the reachability of a marking in Petri nets, such as reachability graphs [46], net invariants [62] and unfolding [30]. The complexity of the reachability problem for Petri nets is not only determined by the different approaches used but also by the structural restrictions of Petri nets themselves. Since SONs are essentially acyclic (no causal cycles), conflict-free (no alternative behaviour is allowed) and 1-safe (a condition/channel place can contain at most one token), it has been proved that the reachability problem in such a subclass of Petri nets turns out to be polynomial [8].

In this section we propose an algorithm for reachability checking in SONs. Given a set of required conditions and channel places, the algorithm proceeds as follows (see Algorithm 3 for details):

1. compute all the causal predecessors of required nodes (e.g. the relations presented in Figure 2.5(b));

2. check that none of the required nodes is consumed by (is the input of) their

causal predecessors;

3. check that none of the corresponding upper level conditions (w.r.t β) of the required node is consumed by their causal predecessors.

Algorithm 3 (Reachability checking)

Inputs:

SON — Structured occurrence nets

M — Marking of SON

Output:

Whether M is reachable from the initial marking

```
1: Pred := ∅                                          // all predecessors of M
2: Cons := ∅          // input conditions and channel places of events in Pred
3: for all c ∈ M do
4:    Predecessors(c)
5: for all n ∈ Pred do
6:    if n is an event then
7:       add all nodes in •n to Cons
8: for all c ∈ M do
9:    if c ∈ Cons or Cons contains all β(c) then
10:      return FALSE
11: return TRUE

12: procedure Predecessors (input: c)
13: mark c visited
14: add c to Pred
15: for all c' ∈ CausalPreset(c) do
16:    if c' is unvisited then
17:       Predecessors(c')

18: function CausalPreset(input: c)
19: Preset := ∅
20: for all node c' such that (c', c) ∈ F ∨ (c', c) ∈ W do
21:    add c' to Preset
22: for (e, f) ∈ ◁ do
23:    if f = c then
24:       add e to Preset
25: return Preset
```

The computation of causal predecessors in Step 1 takes into account all three types of causal relation in ONs, CSONs and BSONs (note that there is no additional causal relation in TSONs). The procedure *Predecessors* is recursively called to explore the causally related nodes of M in the backward direction. The function *CausalPreset* is invoked in each iteration to obtain the causal input neighbours of a given node c. If a node is visited twice or a condition is in the initial state, then the procedure reaches the stop conditions. The set *Pred* is used to store all causal predecessors during the exploration. Step 2 concerns the basic reachability criterion in SON. That is, M is unreachable if there exist two conditions/channel places in M such that one causally precedes the other. This criterion follows from Propositions 2.1(4), 2.2(3) and 2.6(3). Step 3 addresses the consistency between the markings in different BSON levels, i.e. M is unreachable if there is an upper level condition c in M and a lower level condition c' in M such that c causally precedes $\beta(c')$.

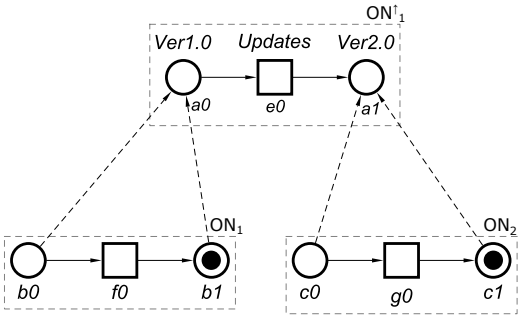


Figure 2.11: A BSON with unreachable marking.

Consider the BSON in Figure 2.11. The causal predecessors of the marking $\{b_1, c_1\}$ are $\{f_0\}$ and $\{e_0, f_0, g_0\}$ respectively. This marking is unreachable from the initial marking. This is because the upper level condition of b_1 (viz. a_0) is consumed by the causal predecessor e_0. Intuitively, unreachability follows since $\{a_0, b_1\}$ will change to $\{a_1, c_0\}$ after firing e_0.

Using the result of Proposition 2.3, (i.e. there is no synchronous cycle in any reachable marking) the algorithm can potentially be improved by performing a check as to whether or not the given marking involves synchronous channel places at the beginning of the algorithm, where the synchronous channel place set can

be efficiently obtained during the acyclicity check presented in Algorithm 2. Then the marking is unreachable and the algorithm stops if the check returns true.

2.7 A Case Study

In this section, we apply SON to model an accident scenario — the *Ladbroke Grove rail crash*. We show how the use of SON can provide a comprehensive and clear structure for the system involving multiple parties.

Ladbroke Grove, London, was the scene of a serious railway accident in October 1999. An outbound three-car diesel train collided with an eight-coach high speed train at a combined speed of 130 mph, with 31 people killed and more than 500 injured. The immediate cause of the disaster is due to the diesel train passing signal SN109 at red for danger (also known as SPAD — Signal Passed At Danger). After the accident, a public inquiry into the crash was held and explored many more details. Figure 2.12 illustrates a SON model of the crash which is based on the information provided in [56, 61].

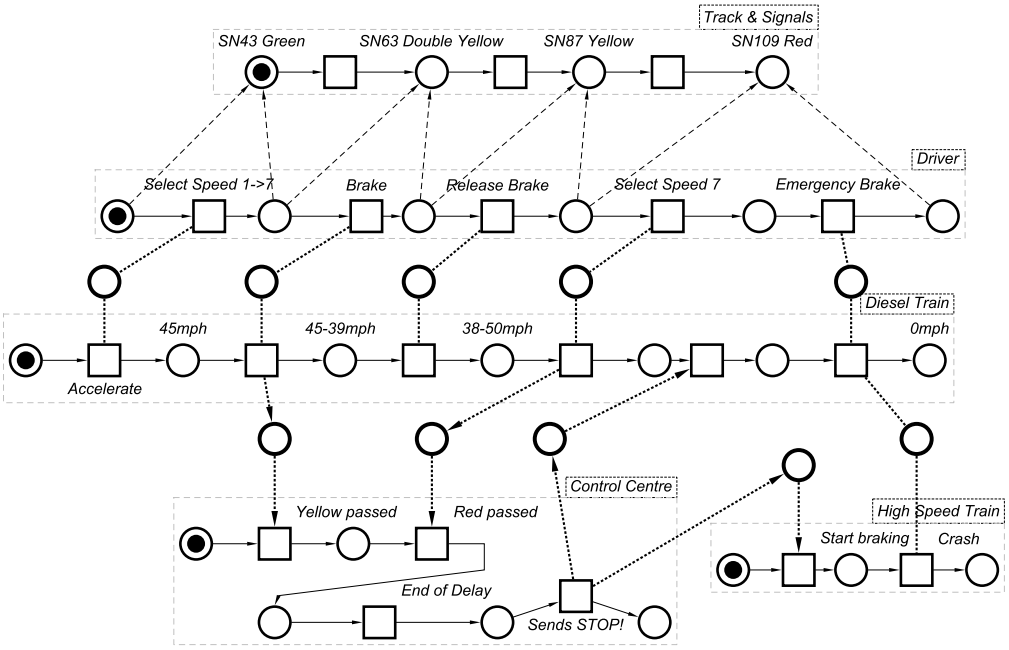


Figure 2.12: The ladbroke grove rail crash model.

The model consists of five parties which have been divided and represented by separate ONs:

Track and Signals represents the the behaviours of the signals that diesel train passed in sequence. The first signal SN43 is showing a green aspect (proceed), the next two signals SN63 and SN87 are both showing yellow aspects indicating a "preliminary caution". The last one SN109 is showing a red aspect meaning stop.

Driver shows the behaviour of the driver of the diesel train. The behaviour includes the movement of the train speed control with seven speed notches 1-7 and the action of emergency brake.

Diesel Train models the speed behaviour of the diesel train which is affected by the driver.

Control Centre models the behaviour of the signaller who is in charge of monitoring the situation.

High Speed Train models the behaviour of the high speed train which collides with the diesel train.

The connections between these ONs reveals their communications and relationships. For example, using the behavioural relations it is possible to show how the activities of *Track and Signals* relate to the *Driver*'s activity, e.g. these exhibit the corresponding driver's responses in accordance with different aspects of the signals. One can observe that, concurrent with the red signal, the driver moved the speed controller to seven (maximum). Synchronous communications between *Driver* and *Diesel Train* indicate the immediate reactions of the train (i.e. accelerate, decelerate, etc.) when the driver manipulated the controller. Communications between *Control Centre* and the two trains are asynchronous since the signaller sent no response messages.

Using the SONs to model the accident gives an explicit view of its behaviours. This has the potential of helping an investigator to understand how the accident has taken place and to explore the backward chain of events to find the accident's "cause".

2.8 Conclusion

In this chapter, we discussed structured occurrence nets. The execution semantics for each variant of SONs have been defined. We introduced and investigated the notion of a channel place in CSONs which gives a flexible way to represent asynchronous and synchronous communications. In particular, synchronous communication exists not only between two events, but may also involve multiple events in different occurrence nets. This led to the notion of a sync-cycle. In BSONs, a refinement of the relation $causal(e)$ has been proposed which captures causality between upper and lower level CSONs using causal dependencies between events. In [40], such a dependency is captured by conditions which cannot guarantee the acyclicity of SONs.

We discussed automated verification in SONs. Structural property checking aims to verify the correctness of the SON's structure and relationships. It is necessary to perform the check before further analysis, otherwise the results may be incorrect. Reachability analysis establishes whether a given marking can be reached from the initial marking. The algorithm we proposed is based on backward exploration and can run in linear time.

We applied SONs to the modelling of an accident scenario which involves multiple parties (sub-systems). The effective use of SONs can portray an explicit view of the system in order to reduce the cognitive complexity needed to understand it.

In the next chapter, we address an extension of basic SON model which is used to represent alternative behaviours of a system.

Chapter 3

Alternative Structured Occurrence Nets

3.1 Introduction

The SONs concept introduced in Chapter 2 plays an important role in the representation of complex evolving system behaviours. However, the concept mainly concerns the modelling of single executions of a system, and there is lack of a direct way to represent a system involving alternative behaviours. In this chapter, we generalise the above idea and introduce an extension of SONs which supports the modelling of alternative behaviours, named alternative structured occurrence nets (ASONs). The idea of adding alternates to SONs initially arose from [41] for the propose of modelling and analysing more complex evolving systems, e.g. major (cyber) crimes or accidents, both of which are likely to give rise to a mass of contradictory or uncertain evidence. In criminal and accident investigations, the evidence available to the investigation team is often contradictory, unclear or uncorroborated. Investigators typically hypothesize alternate scenarios in order to explain a crime or an accident, of which at most one scenario actually occurred. Different scenarios may not be completely independent of each other. They may represent the same system, so that common behaviours are shared by several scenarios.

A SON model can explicitly portray the behavioural records about a single

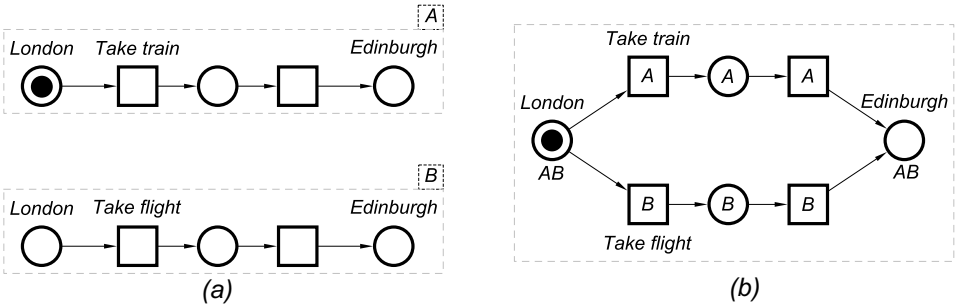


Figure 3.1: (a) Multiple scenarios modelled by two occurrence nets, and by (b) a single AON.

scenario. However, there is a lack of way to represent multiple scenarios (with shared behaviours) efficiently. Consider the example in Figure 3.1(a). Using the SONs idea, the activities of a man travelling to a destination in alternative ways would have to be represented in separate ONs (A and B), where each ON represents a unique scenario. Modelling in this way causes many duplicated states since these two ONs actually describe the same system. Figure 3.1(b) shows an enhancement of SONs for such a situation. Basically, the ONs in (a) are merged into a single structure by 'gluing' common states (conditions) together. So a state shared by more than one scenario can branch to multiple events with each corresponding to a different scenario; the different branches can subsequently merge from their respective end events to a state, with the result that different scenarios share the same state. Therefore, the representation of such structure is more flexible than branching processes [24] where two branches outgoing from a condition will never meet again.

There are already-existing ON extensions which support alternative behaviours. [29] introduced a 'barb' concept to describe an event that could have occurred, given the condition that existed, but did not execute due to another relevant event being selected. The alternative ON proposed in [41] is based on the barb concept, so as to allow the simultaneous modelling of different scenarios. The idea behind [41] is to tag each scenario (ON) with a symbol, and an alternative ON is essentially an overlay of individual scenarios with each element marked with one or more scenario symbols.

The ASON concept introduced in this chapter further extends the SON nota-

tion, by adopting the tagging idea in [41]. In Section 3.2, we define alternative occurrence nets (AONs), and propose Petri net-based execution semantics for AONs. We show several important properties based on the structure and step sequence of AON. Section 3.3 introduces an extension of CSONs which allows portrayal of different kinds of communication between AONs. Section 3.4 discusses the modelling of evolution information in AON. Section 3.5 concludes the chapter.

3.2 Alternative ON

In this section, we formally define the AON model. The notation we use is based on the formalisms for occurrence nets and temporal structural occurrence nets introduced in Chapter 2. The only new concept is that of a set of *alternative scenarios*, $AS = \{A_1, \ldots, A_\varphi\}$.

Definition 3.1 (Tagged-ON)**.** *Let* $AS = \{A_1, \ldots, A_\varphi\}$ *be a set of alternative scenarios. A* tagged occurrence net *is a tuple* $\text{TAGON} = (C, E, F, \vartheta)$ *where* C *and* E *are disjoint sets of respectively* conditions *and* events *(collectively referred to as the* nodes*),* $F \subseteq (C \times E) \cup (E \times C)$ *is the* flow *relation, and* $\vartheta : C \cup E \cup F \to 2^{AS} \setminus \{\varnothing\}$ *is a mapping, such that, for each* $A \in AS$*,*

$$\text{TAGON}(\text{A}) = (C(\text{A}), E(\text{A}), F(\text{A}))$$

is an occurrence net, where for $X \in \{C, E, F\}, X(\text{A})$ *is the set of all* $x \in X$ *such that* $A \in \vartheta(x)$*. The initial marking* M_0^{TAGON}*, final marking* M_{Fin}^{TAGON}*, input and output of a node* x*, i.e.* ${}^\bullet x$ *and* $x^\bullet$*, in* TAGON *are defined in the same way as in* ON*. It is further assumed that for all* $A \in AS$*,* $M_0^{\text{TAGON}} = M_0^{\text{TAGON(A)}}$ *and* $M_{Fin}^{\text{TAGON}} = M_{Fin}^{\text{TAGON(A)}}$*.*

We define two nodes, x and y, as being *alternatively related* (or *conflicting*) if there are distinct events, e and f, such that ${}^\bullet e \cap {}^\bullet f \neq \varnothing$ and $(e, x) \in F^*$ and $(f, y) \in F^*$ (denoted by $x \# y$). They are causally related if $(x, y) \in F^+$ or $(y, x) \in F^+$. A *block* of the TAGON is a non-empty set $B \in (C \cup E)$ such that $B \cap C = {}^\bullet(B \cap E) \cap (B \cap E)^\bullet$, $({}^\bullet B \setminus B) \times (B^\bullet \setminus B) \subseteq \prec$, and for all $e, f \in (B \cap E)$, $\neg(e \# f)$.

Definition 3.1 incorporates the basic idea of portraying the alternative behaviours of a system. Essentially, a TAGON can be regarded as an overlay of

multiple ONs, each being tagged by a symbol A in AS representing a unique scenario (or world). This is specified by the mapping ϑ. Thus each element in an TAGON is tagged by one or more tags to indicate to which scenarios it belongs. The tagging is not completely arbitrary; all the elements with the same tag form a valid occurrence net (i.e. TAGON(A)). This represents the behavioural report of what has happened from the point of view of one of the possible scenarios $A \in AS$.

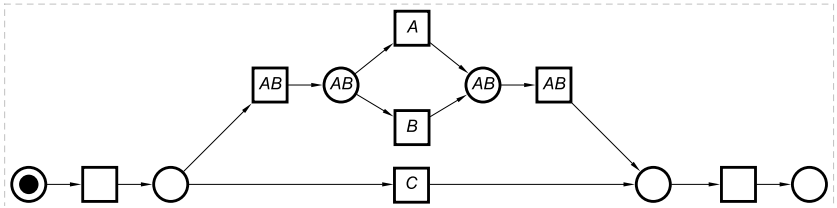


Figure 3.2: A tagged occurrence net with three scenarios.

The last statement of the definition postulates that the initial and final markings of TAGON respectively correspond to the initial and final markings of each occurrence net TAGON(A). Intuitively, this attempts to express that, although there may exist alternative scenarios, a TAGON is still used to model the behaviours of a single system which starts and ends at unique states. This property is particularly important when applying the phase concept to alternative occurrence nets, which will be addressed later. To give an example, Figure 3.2 illustrates a TAGON tagged by three scenarios A, B and C showing inside the elements (we omit tags for certain elements and flow relations). One can observe that each scenario forms a valid occurrence net which shares the same initial and final markings with the TAGON.

Having defined the tagged occurrence nets, we are able to introduce alternative occurrence nets by adding additional restrictions to the TAGON. Below we assume that a TAGON is *sequential* if $|M_0^{\text{TAGON}}| = 1$; and for all $e \in E$, $|{}^\bullet e| = 1$ and $|e^\bullet| = 1$. For example, the TAGON in Figure 3.2 is sequential.

Definition 3.2 (AON). *Let* $\text{TAGON}^\downarrow = (C^\downarrow, E^\downarrow, F^\downarrow, \vartheta^\downarrow)$ *be a tagged occurrence net as in Definition 3.1, and* $\text{TAGON} = (C, E, F, \vartheta)$ *be a sequential tagged occurrence nets.*

*An alternative occurrence net (*AON*) is a pair*

$$\text{AON} = (\text{TAGON}^{\downarrow}, \tau)$$

such that $\tau : C^{\downarrow} \cup E^{\downarrow} \to C \cup E$ *is a mapping from the nodes of* TAGON$^{\downarrow}$ *to the nodes of* TAGON.
It is assumed that the following hold:

1. $\tau(C^{\downarrow} \cup E^{\downarrow}) = C \cup E, \tau^{-1}(C) \subseteq C^{\downarrow}$, *and* $\tau^{-1}(E) = E^{\downarrow}$.
2. *for all* $e \in E, \tau^{-1}(e)$ *are disjoint blocks of sequential-*TAGON.
3. *for all* $c \in C, |\tau(c)| = 1$.
4. $F = \{(x, y) \in (C \times E) \cup (E \times C) | (\tau(x) \times \tau(y)) \cap F^{\downarrow} \neq \varnothing\}$.
5. *for all* $x \in (C^{\downarrow} \cup E^{\downarrow} \cup F^{\downarrow})$, $\vartheta(x) = \vartheta(\tau(x))$.

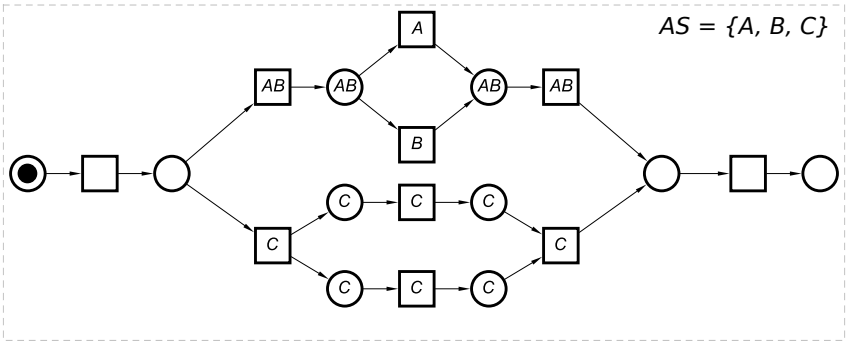


Figure 3.3: An alternative occurrence net.

We also define that AON(A) = TAGON$^{\downarrow}$(A) is a valid occurrence net. The intuitive meaning of the above definition is similar to the idea of the temporal abstraction (TSON). An AON is a tagged occurrence nets with a mapping from TAGON$^{\downarrow}$ to a sequential TAGON. Each event in the TAGON corresponds to a disjoint block in the TAGON$^{\downarrow}$; and each event in TAGON$^{\downarrow}$ belongs to an event in TAGON. Definition 3.2(5) additionally restricts that the scenario tags of each node in the TAGON$^{\downarrow}$ are inherited from the tags of its corresponding node in the

TAGON. Thus, a TAGON is an AON if and only if there exist a sequential TAGON and a mapping τ between the two nets. Those restrictions make sure the consistency between the scenario tags and the concurrency in AON. That is, any two concurrently related nodes must hold the same tags. In this way, the complexity of coupling alternation with communication and evolution behaviours can be greatly reduced. As an example, Figure 3.3 shows an alternative occurrence net. Its corresponding sequential TAGON is the net shown in Figure 3.2.

The following propositions address the relationships between tags and nodes in alternative relations. Given an AON, we first show that any two output (input) events of a condition never hold the same scenario tag. Moreover, the scenario tags of the output (input) events of a condition are inherited from it.

Proposition 3.1. *Let* AON *be an alternative occurrence net as in Definition 3.2, and c be a condition in* AON*.*

1. *For all $e, f \in {}^\bullet c : e \neq f \iff \vartheta(e) \cap \vartheta(f) = \varnothing$, and for all $e', f' \in c^\bullet : e' \neq f' \iff \vartheta(e') \cap \vartheta(f') = \varnothing$.*

2. $\vartheta(c) = \bigcup_{e \in {}^\bullet c} \vartheta(e) = \bigcup_{f \in c^\bullet} \vartheta(f)$.

Proof. (1) Suppose that $\vartheta(e) \cap \vartheta(f) = \{A\}$ and $\vartheta(e') \cap \vartheta(f') = \{A\}$. Then AON(A) is an invalid ON, as $|{}^\bullet c| > 1$, and $|c^\bullet| > 1$ contradicting Definition 3.1.

(2) Suppose that $\vartheta(c) \neq \bigcup_{e \in {}^\bullet c} \vartheta(e)$, there exist $A \in AS$ such that $A \notin \vartheta(c)$ and $A \in \bigcup_{e \in {}^\bullet c} \vartheta(e)$. Then AON(A) is an invalid ON, as $|{}^\bullet e| = \varnothing$ contradicting Definition 3.1. In the case $A \in \vartheta(c)$ and $A \notin \bigcup_{e \in {}^\bullet c} \vartheta(e)$, c is the initial condition of AON(A) but not the initial condition of AON contradicting Definition 3.1.

Similarly, $\vartheta(c) \neq \bigcup_{e \in c^\bullet} \vartheta(e)$, $A \notin \vartheta(c)$ and $A \in \bigcup_{e \in c^\bullet} \vartheta(e)$ imply $|e^\bullet| = \varnothing$; $A \in \vartheta(c)$ and $A \notin \bigcup_{e \in c^\bullet} \vartheta(e)$ imply c is the final condition of AON(A) but not the final condition of AON, contradicting Definition 3.1.

□

The next property shows that, if two nodes are alternatively related, then they must hold different tags. For example, in Figure 3.3 any uncertain event tagged by A and any uncertain event tagged by C are alternatively related, so as to have different tags.

Proposition 3.2. *Let* AON *be an alternative occurrence net as in Definition 3.2, and x, y be two nodes in* AON*. Then $x\#y$ implies $\vartheta(x) \cap \vartheta(y) = \varnothing$.*

Proof. $x\#y$ and $\vartheta(x) \cap \vartheta(y) = \{A\}$ indicates that AON(A) is not a valid occurrence net. A contradiction with Definition 3.1.

□

A *phase* of an AON is defined as a non-empty set of conditions $\pi \subseteq C$ such that there is a $A \in AS$ where π is a valid phase in the AON(A). A *phase decomposition* of AON is a sequence of phases such that there is a $A \in AS$ where the sequence is a valid phase decomposition in the AON(A). Thus in AON there may be more than one phase decompositions in a single AON, each of which belongs to a different scenario. Consider again the AON in Figure 3.3. One of the possible phases is composed of the two conditions which are tagged by AB. In fact such a phase is shared by two scenarios A and B.

The AON firing rule is defined below. It explains how the states change in a system involving different scenarios. More precisely, if a marked condition branches to multiple events, there is exactly one of these events that can be chosen to fire. Note that a step in the AON is a set of events.

Definition 3.3 (AON firing rule)**.** *Let* AON *be an alternative occurrence net, M be a marking, U be a step of* AON*, and $A \in AS$ be an alternative scenario.*

1. *U is* AON*-enabled at M and A if*
   - *$M(c) \geq \sum_{e \in c^\bullet} U(e)$, for every condition $c \in C$; and*
   - *$A \in \vartheta(e)$, for every event $e \in U$.*
2. *If U is* AON*-enabled at M and A, then U can be fired producing a new marking: $M' = (M \setminus {}^\bullet U) \cup U^\bullet$;*

   this is denoted by $M[U\rangle^A_{\text{AON}} M'$.

A *step sequence* of AON (w.r.t. scenario A) is a sequence $\lambda^A = U_1 \ldots U_n$ ($n \geq 0$) of steps, such that there exist markings $M_1, \ldots, M_n$ satisfying:

$$M_0^{\text{AON}}[U_1\rangle^A_{\text{AON}} M_1, \ldots, M_{n-1}[U_n\rangle^A_{\text{AON}} M_n \ .$$

Thus a step sequence portrays a valid system execution with respect to a particular scenario.

The *reachable markings* of AON (w.r.t. scenario A) are defined as the smallest (w.r.t. $\subseteq$) set $reach_A(\text{AON})$ containing M_0^{AON} and such that if there is a marking $M \in reach_A(\text{AON})$ and $M[U\rangle^A_{\text{AON}} M'$, for a step U and a marking M', then $M' \in reach_A(\text{AON})$.

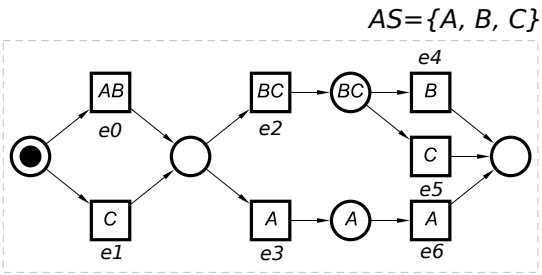


Figure 3.4: An alternative occurrence net.

Considering the AON shown in Figure 3.4, $\lambda^A = \{e_0\}\{e_3\}\{e_6\}$ and $\lambda^B = \{e_0\}\{e_2\}\{e_4\}$ are two possible step sequences which respectively represent the executions of scenarios A and B. However, $\{e_1\}\{e_3\}\{e_6\}$ is not a valid step sequence, since e_1 belongs to scenario C while e_3 and e_6 belong to scenario A. Intuitively, this can be explained by saying that a single run cannot happen in different worlds. We can also observe that the two valid step sequences described above correspond to the maximal executions of the occurrence nets AON(A) and AON(B). This is not a coincidence; the following results show their relationships.

Proposition 3.3. *Let* AON *be an alternative occurrence net as in Definition 3.2, and* $\lambda^A = U_1 \ldots U_n$ $(n \geq 0)$ *be a step sequence of* AON*, w.r.t. an alternative scenario* $A \in AS$*.*

1. λ^A *is a step sequence of the occurrence net* AON(A)*.*
2. *For each* $A \in AS$*, there is a step sequence involving all events in* E(A)*.*
3. *If* $i \neq j$ *then* $U_i \cap U_j = \varnothing$*, i.e. no event occurs more than once.*

Proof. We prove the proposition for sequential TAGONs, then the general result follow from Theorem 2.1.

(1) By Definition 3.2 each occurrence net TAGON(A) is a line-like ON

$$c_0 e_0 \dots e_m c_m \dots e_n c_n$$

which starts at the initial marking, ends at the final marking, and tag A is shared by all its elements. By Definition 3.3 and the fact that there is no concurrent behaviour in the TAGON, the step sequence λ^A together with the corresponding reachable markings is a sequence $c'_0 e'_0 \dots e'_m c'_m$ starting from the initial marking.

By Definition 3.1, $c_0 = c'_0$. By Definition 3.3 and Proposition 3.2, $e_0 = e'_0, \dots e_m = e'_m$ and $c_m = c'_m$.

(2) Suppose that there is no such step sequence, this implies that there exists an event $e \in E(\text{A})$ such that e is AON-unenabled, w.r.t. all reachable markings of AON. Since each TAGON(A) in the AON is a line-like sequence, e is unenabled, which implies $\vartheta(e) \neq \vartheta({}^\bullet e)$. This is a contradiction with Proposition 3.1(1).

(3) The proof follows from Propositions 3.3(1) and 2.1(1).

□

3.3 Alternative CSONs

The AONs introduced in the previous section are based on the occurrence net. The extension is able to model alternative behaviours in a single system. In this section, we extend the fundamental concept of CSONs with the notion of alternative behaviours.

Definition 3.4 (ACSON). *Let* $\text{AON}_i = (C_i, E_i, F_i, \vartheta_i)$ *for* $i = 1, \dots, k$ *be alternative occurrence nets (we denote respectively by* $\mathbf{C}$, $\mathbf{E}$ *and* $\mathbf{F}$ *their conditions, events and arcs);* Q *be a set of channel places;* $W \in (\mathbf{E} \times Q) \cup (Q \times \mathbf{E})$ *be the arcs between the channel places and events; and* $\vartheta_0 : Q \cup W \to 2^{AS} \setminus \{\varnothing\}$ *be a mapping.*

*A alternative communication structured occurrence net (*ACSON*) is a tuple*

$$\text{ACSON} = (\text{AON}_1, \dots, \text{AON}_k, Q, W, \vartheta_0)$$

such that, for every $A \in AS$,

$$\textsc{acson}(\textsc{a}) = (\textsc{aon}_1(\textsc{a}), \ldots, \textsc{aon}_k(\textsc{a}), Q(\textsc{a}), W(\textsc{a}))$$

is a communication structured occurrence net, where for $X = Q, W, X(\textsc{a})$ *is the set of all* $x \in X$ *such that* $A \in \vartheta_0(x)$. *It is further assumed that:*

1. *The sets of input and output events of* $q \in Q$,

   $$^\bullet q = \{e \in \boldsymbol{E} \mid (e, q) \in W\} \quad \textit{and} \quad q^\bullet = \{e \in \boldsymbol{E} \mid (q, e) \in W\}\ ,$$

   belong to distinct component $\textsc{aon}_i$*s; and moreover,* $|^\bullet q| = 1$ *and* $|q^\bullet| \leq 1$.

2. *For all* $q \in Q$, $\vartheta(q) = \vartheta(^\bullet q)$ *and* $\vartheta(q) = \vartheta(q^\bullet)$ *if* $|q^\bullet| = 1$, *i.e.* q, *its input and output have the same tags.*

We also define that the initial marking $M_0^{\textsc{acson}}$ of the ACSON is the union of $M_0^{\textsc{aon}_1}, \ldots, M_0^{\textsc{aon}_k}$, and the final marking $M_{Fin}^{\textsc{acson}}$ is the union of $M_{Fin}^{\textsc{aon}_1}, \ldots, M_{Fin}^{\textsc{aon}_k}$. A marking in the ACSON is a set of conditions in $\mathbf{C}$ and channel places in Q.

In contrast to the CSON where a set of component ONs are used to represent separate subsystems, the modelling of subsystems in the ACSON are replaced by AONs, and the channel places representing asynchronous or synchronous communication are connected with events in different AONs. In addition, each element in the ACSON is associated with one or more scenario tags. All the elements with the same tag form a valid CSON. Therefore, similarly to the case of AONs, an ACSON can be regarded as a number of CSONs which are overlaid into a single structure, and some of the elements are 'certain' if they are shared by all scenarios, otherwise they are 'uncertain'.

The latter two conditions in the above definition provide restrictions concerning the structure and tag of channel places. In particular, Definition 3.4(1) defines the input and output events of channel places. Definition 3.4(2) implies that the tags held in each channel place are the same as the tags in its input and output events. This condition intuitively means that if an event in ACSON is shared by some scenarios, then any communication linked with the event is also required to be shared by those scenarios. In this way, we guarantee that the initial and final markings of each ACSON(A) are respectively equal to the initial and final markings of the ACSON.

The following definition shows the ACSON firing rule. A step in an ACSON is a set of events. The firing rule is based on the one defined for CSON which takes into account the execution semantics of synchronous behaviours. Additionally, the new definition is able to cope with alternative behaviours for uncertain elements.

Definition 3.5 (ACSON firing rule)**.** *Let* CSON *be an alternative communication structured occurrence net as in Definition 3.4, M be a marking, U be a step of* ACSON*, and $A \in AS$ be an alternative scenario.*

1. *U is* ACSON-enabled *at M and A if*
   - $M(c) \geq \sum_{e \in c^\bullet} U(e)$, *for every condition* $c \in \mathbf{C}$;
   - $M(q) + \sum_{e \in {}^\bullet q} U(e) \geq \sum_{e \in q^\bullet} U(e)$ *for every channel place* $q \in Q$; *and*
   - $A \in \vartheta(e)$, *for every event* $e \in U$.
2. *If U is* ACSON*-enabled at M, then U can be* fired *and produce a new marking M' which is given by: $M' = (M \cup U^\bullet) \setminus {}^\bullet U$. This is denoted by $M[U\rangle^A_{\text{ACSON}} M'$.*

The *step sequences* and *reachable marking* of CSON are then defined in the same way as for the AON.

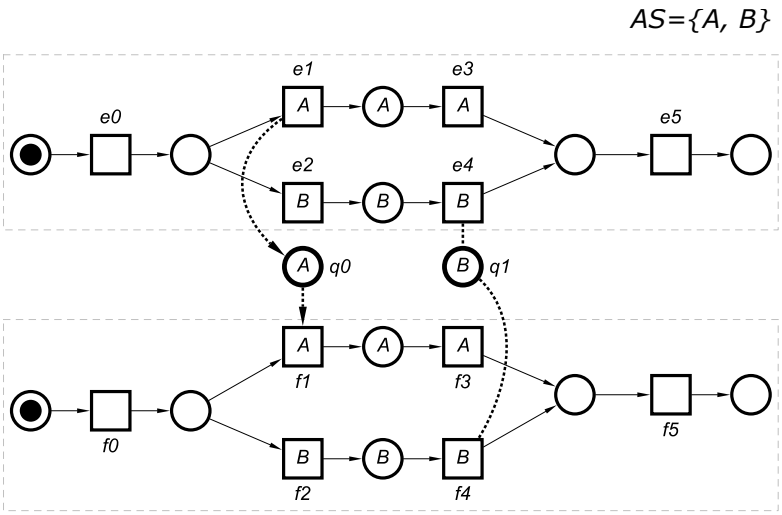


Figure 3.5: An alternative communication structured occurrence net (ACSON).

Figure 3.5 illustrates an example of an ACSON which involves two scenarios A and B. The ACSON consists of two component AONs communicating via two channel places. However, the two communications can never exist in a single system execution since they belong to different scenarios. The CSON formed by tag A only contains an asynchronous communication from e_1 to f_1, while the other CSON formed by tag B has a synchronous communication between e_4 and f_4. Using the firing rule defined above, for each scenario we can find at least one step sequence which is from the initial marking of the ACSON to the final marking. For example, in scenario A we have $\lambda^A = \{e_0, f_0\}\{e_1\}\{f_1\}\{e_3, f_3\}\{e_5, f_5\}$, and in scenario B we have $\lambda^B = \{e_0, f_0\}\{e_2, f_2\}\{e_4, f_4\}\{e_5, c_5\}$.

The following proposition shows that each step sequence in ACSON is also a valid step sequence of one of the scenarios in ACSON.

Proposition 3.4. *Let* ACSON *be an alternative communication structured occurrence net as in Definition 3.4, and* $\lambda^A = U_1 \dots U_n$ $(n \geq 0)$ *be a step sequence of* ACSON*, w.r.t. an alternative scenario* $A \in AS$*.*

1. λ^A *is a step sequence of the communication structured occurrence net* ACSON(A)*.*
2. *For each* $A \in AS$*, there is a step sequence involving all events in* E(A)*.*
3. *If* $i \neq j$ *then* $U_i \cap U_j = \varnothing$*, i.e. no event occurs more than once.*

Proof. (1) Since there are no concurrently related nodes in each component TAGON of the ACSON, for all $e, f \in U$, e, f belong to disjointed AONs. From Definition 3.5, $(e, f) \notin (\sqsubset \cup \prec)^* \wedge (f, e) \notin (\sqsubset \cup \prec)^*$.

The proofs of (2) and (3) above follow from 1 and then Propositions 2.2(1) and (2). □

3.4 Alternative BSONs

In the previous section, we discussed communication in AONs. The new structure is an extended CSON which allows the modelling of interactions between systems with alternative behaviours. In this section, we introduce the notion of evolution to AONs.

Definition 3.6 (ABSON)**.** *Let* ACSON *be an alternative communication structured occurrence net as in Definition 3.4;* $\text{CSON}^\uparrow = (\text{ON}_1^\uparrow, \ldots, \text{ON}_m^\uparrow, Q^\uparrow, W^\uparrow)$ *be a (disjoint) communication structured occurrence net such that* $\text{ON}_i^\uparrow = (C_i^\uparrow, E_i^\uparrow, F_i^\uparrow)$ *is line-like, for* $i \leq m$*, and* $\mathbf{C}^\uparrow$*,* $\mathbf{E}^\uparrow$ *and* $\mathbf{F}^\uparrow$ *be their conditions, events and arcs;* $\beta \subseteq \mathbf{C} \times \mathbf{C}^\uparrow$ *be the arcs between conditions in* ACSON *and* $\text{CSON}^\uparrow$*; and* $\vartheta_0 : \beta \to 2^{AS} \setminus \{\varnothing\}$ *be a mapping.*

*An alternative behavioural structured occurrence net (*ABSON*) is a tuple*

$$\text{ABSON} = (\text{ACSON}, \text{CSON}^\uparrow, \beta, \vartheta_0)$$

such that, for every $A \in AS$*,*

$$\text{ABSON}(\text{A}) = (\text{ACSON}(\text{A}), \text{CSON}^\uparrow, \beta(\text{A}))$$

is a behavioural structured occurrence net, where $\beta(\text{A})$ *is the set of all* $x \in \beta$ *such that* $A \in \vartheta_0(x)$*. It is further assumed that:*

1. *For every* AON_i*, there exists exactly one* $\text{ON}_j^\uparrow$ *satisfying* $\beta(C_i) \cap C_j^\uparrow \neq \varnothing$*. We also denote that for all* $c \in \mathbf{C}_j^\uparrow$*,* $\pi(c)$ *is a set of phases such that for all* $\pi \in \pi(c)$*,* $\beta(\pi) = c$*.*

2. *For every* $c \in \mathbf{C}$*,* $\beta(c)(\text{A}) = \beta(c)(\text{B})$*, where* $A, B \in \vartheta_0(c)$*.*

The initial marking M_0^{ABSON} of ABSON is the initial marking of the $\text{CSON}^\uparrow$ together with the initial markings of all the AON_is such that $\beta(M_0^{\text{AON}_i}) \cap M_0^{\text{CSON}^\uparrow} \neq \varnothing$.

An ABSON consists of a $\text{CSON}^\uparrow$ and an ACSON which are linked by the behavioural relation β. The upper level net of an ABSON is a (non-alternative) CSON with each component ON line-like, while the ACSON is a lower level net representing the detailed behaviour of the system which may involve alternative behaviours indicated by different tags in AS. What is more, each scenario together with the corresponding upper level $\text{CSON}^\uparrow$ forms a valid behavioural structured occurrence net. In fact, every element in $\text{CSON}^\uparrow$ can be treated as a certain one which is tagged by all the tags in AS.

Definition 3.6(1) addresses the relationships between the two levels of ABSON. The first part of the statement implies that (the phases in) each lower level AON points to (the conditions in) exactly one upper level ON. Unlike the BSON where phases and upper level conditions are in an one-to-one relationship, an upper

level condition in the ABSON may map to multiple phases due to the alternative relations. More precisely, recall Definition 2.5(2) that the conditions in the chain $\xi_{\text{ON}_j^\uparrow} = c_1 e_1 \dots e_{l-1} c_l$ of the upper level ON map to a phase decomposition in the lower net. However, in ABSON there may be more than one phase decomposition in a single AON, and the phases belonging to different scenarios may point to the same upper condition. Therefore, we refine the notion $\pi(c)$ to cope with the new relationship.

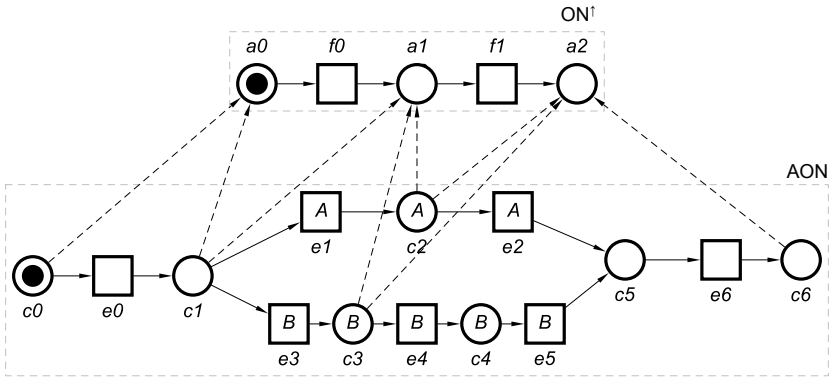


Figure 3.6: An alternative behavioural structured occurrence net.

Consider the ABSON shown in Figure 3.6. The lower level AON involves two scenarios A and B, each of which has its own phase decomposition $\pi_1\pi_2\pi_3$ pointing to the conditions of the chain $\xi_{\text{ON}^\uparrow} = a_0 f_0 a_1 f_1 a_2$ in the upper level $\text{ON}^\uparrow$. We can observe that the phases $\{c_1, c_2\}$ and $\{c_1, c_3\}$ point to a single condition a_1 since they are both the π_2 of AON(A) and AON(B).

Next we define the ABSON firing rule. On the one hand, the rule takes into account the execution of alternatively related nodes in ACSON. On the other hand, it concerns the one-to-many relationships between upper level conditions and phases. A step U is ABSON-enabled if: (i) it is ACSON-enabled (Definition 3.5); and (ii) for each upper level event in U, the maximal conditions in one of the phases of its input condition is in the current marking; and (iii) for each lower level event in U, its corresponding upper level condition is in the current marking.

Below we assume that $V \subseteq U \cap E^\uparrow$ is a subset of upper level events in U such that for every $e \in V$ there is a phase $\pi \in \pi({}^\bullet e)$ such that Max_π is the final

marking of a component AON in ABSON, i.e. $V = \{e \in (U \cap E^{\uparrow}) \mid \exists \pi \in \pi(^{\bullet}e) : Max_{\pi} \subseteq \bigcup_{i=1}^{m} M_{Fin}^{\text{AON}_i}\}$. Moreover, $Max_{\pi(^{\bullet}V)}$ is the union of all maximal phases of $\pi(^{\bullet}V)$, and $Min_{\pi(V^{\bullet})}$ is the union of all minimal phases of $\pi(V^{\bullet})$. Note that $Min_{\pi(V^{\bullet})}$ are also the initial markings of some component AONs. Intuitively, these notions are used to portray the disjoint part of an 'off-line' structure in ABSON.

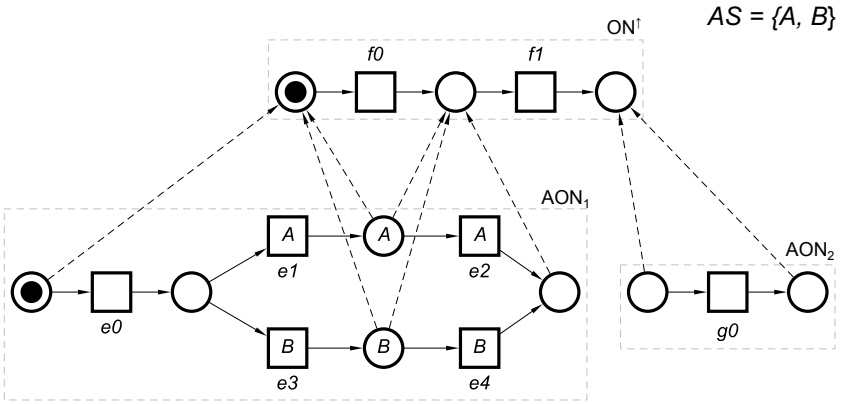


Figure 3.7: An alternative behavioural structured occurrence net.

Definition 3.7 (ABSON firing rule)**.** *Let* ABSON *be as in Definition 3.6. $M \subseteq \mathbf{C} \cup \mathbf{C}^{\uparrow}$ be a marking, $U \subseteq \mathbf{E} \cup \mathbf{E}^{\uparrow}$ be a step of* ABSON*, and $A \in AS$ be an alternative scenario.*

1. *U is* ABSON-enabled *at M and A if*
   - *$M(c) \geq \sum_{e \in c^{\bullet}} U(e)$, for every condition $c \in \mathbf{C} \cup \mathbf{C}^{\uparrow}$;*
   - *$M(q) + \sum_{e \in {}^{\bullet}q} U(e) \geq \sum_{e \in q^{\bullet}} U(e)$ for every channel place $q \in Q \cup Q^{\uparrow}$;*
   - *For every $e \in \mathbf{E}^{\uparrow} \cap U$, there is a phase $\pi \in \pi(^{\bullet}e)$ such that $Max_{\pi} \subseteq M$;*
   - *$\beta(^{\bullet}e') \cap \beta(e'^{\bullet}) \in M$, for every $e' \in \mathbf{E} \cap U$;*
   - *$A \in \vartheta(e)$, for every event $e \in U$.*
2. *If U is* ABSON*-enabled at M, then U can be* fired *and produce a marking M' given by:*

$$M' = (M \setminus (^{\bullet}U \cup Max_{\pi(^{\bullet}V)})) \cup U^{\bullet} \cup Min_{\pi(V^{\bullet})}$$

This is denoted by $M[U\rangle^{A}_{\text{ABSON}} M'$.

The *step sequences* and *reachable marking* of ABSON are then defined in the same way as for AONs.

Using the firing rule defined above, one of the possible step sequences of the ABSON in Figure 3.7 is $\lambda^A = \{e_0\}\{e_1\}\{f_0\}\{e_2\}\{f_1\}\{g_0\}$ which corresponds to the maximal step sequence of scenario A, i.e. ABSON(A).

3.5 Conclusion

In this chapter, we discussed ASON – an extended SON concept aimed at modelling a large body of contradictory records of complex activities. Essentially, any variant of ASONs can be regarded as an overlay of multiple basic SON variants (scenarios). In an ASON each element can be shared by one or more scenarios, and each run corresponds to the run of one of its scenarios. To define the ASON, we also adopt the block concept in order to reduce structural complexity.

Once we can model the alternative behaviours in a system, attributes such as time can be attached to elements, in order to decide what scenarios actually happened, so that all but one of each set of alternative behaviours can be discarded. In the next chapter, we will introduce timed-SONs and timed-ASONs which provide support for the associated time information for SON-based concepts.

Chapter 4

Time in Structured Occurrence Nets

4.1 Introduction

In this chapter, we introduce time property to the basic and alternative SONs concepts. Representing time is an attractive feature in the modelling of complex evolving systems. In software engineering, the design of many information systems must deal with temporal information. For example, the time chain of events in medical records applications is a critical part of the data; and 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 action or state is not precise or is incomplete. For instance, it may not be possible to give an exact time at which a robbery occurred, but it may be possible to give bounds for the time period in which the robbery occurred. Numerous time extensions have been proposed in Petri nets [17, 37, 39]. However, most of them only consider the time information as an abstraction at the level of system specification, and research on attaching time information to the level of system behaviour is limited. Therefore, the contribution of this work 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 chapter is organized as follows: the notation of timed SONs (based on discrete time intervals) is given in Section 4.2. Conditions for checking the consistency of time intervals are defined in Section 4.3, and algorithms for estimating and for increasing the precision of time intervals using default duration intervals and redundant time information are given in Section 4.4; Section 4.6 concludes the chapter.

4.2 Time Model

In a time-based system, such as a crime or accident scenario, 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 or 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.

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

4.2.1 Global Time

A global clock is an abstraction (i.e. a logical construct) that provides global time. It is used for capturing chronological and causal relationships in a distributed system. The global clock has a fixed origin and a fixed granularity, and therefore supports empirical verification of event ordering. The fixed origin of the global clock supports the correct ordering of events using timestamps.

The use of different levels of abstraction (in TSONs) 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 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 maximal sum of the durations of its conditions at the base level of abstraction such that no two conditions are pairwise concurrent.

4.2.2 Modelling Uncertainty

An 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 condition, the finish time of the condition, the start time of an event, the finish time of the event, the duration of the condition, and the duration of the event (and the bounds on their respective uncertainties) can each be represented using an 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 9:00 can be represented by the interval $[09{:}00, 09{:}00]$. A condition known to have started sometime between 9:00 and 12:30 can be represented using the interval $[09{:}00, 12{:}30]$. An event known to have occurred at any time before 12:30 can be represented using the interval $[-\infty, 12{:}30]$. An event known to have occurred at any time after 12:30 can be represented using the interval $[12{:}30, +\infty]$ treating $+\infty$ as the maximum possible time. An unknown time can be represented by the interval $[-\infty, +\infty]$.

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

4.3 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 4.1.

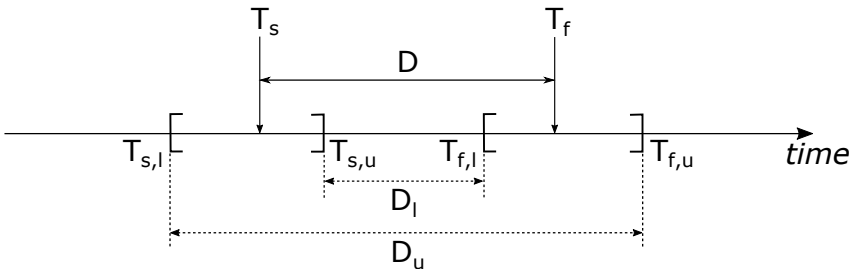


Figure 4.1: Relationship between the unknown time and duration values of a node and their known bounds on the global timeline.

4.3.1 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_s^n and T_f^n respectively.

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

$$I_s^n \triangleq [T_{s,l}^n, T_{s,u}^n]$$

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

$$T_{s,l}^n \leq T_{s,u}^n \tag{4.1}$$

The finish time interval of n represents the bounded uncertainty about the value of T_f^n, and is denoted by:

$$I_f^n \triangleq [T_{f,l}^n, T_{f,u}^n]$$

where $T_{f,l}^n$ and $T_{f,u}^n$ are the lower and upper bounds respectively on the finish time of n. I_f^n is well-defined if and only if the following inequality is satisfied:

$$T_{f,l}^n \leq T_{f,u}^n \tag{4.2}$$

We assume that the start time of n is at, or before, the finish time of n, which is expressed by the restriction: $T_s^n \leq T_f^n$. In order to ensure consistency with this restriction, the start and finish time intervals of n must satisfy the following

inequalities:

$$T^n_{s,l} \leq T^n_{f,l} \ \wedge \ T^n_{s,u} \leq T^n_{f,u} \tag{4.3}$$

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:

$$0 \leq D^n_l \leq D^n_u \tag{4.4}$$

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

4.3.2 Time Consistency

4.3.2.1 Time consistency in line-like ONs

As defined in Chapter 2, in a line-like 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:

$$\forall n_1, n_2 \in (E \cup C)\,((n_1, n_2) \in F \implies I^{n_1}_f = I^{n_2}_s) \tag{4.5}$$

Let n be a node in a line-like 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:

$$[T_{s,l} + D_l, T_{s,u} + D_u] \cap [T_{f,l}, T_{f,u}] \neq \emptyset \tag{4.6}$$

$$[T_{f,l} - D_u, T_{f,u} - D_l] \cap [T_{s,l}, T_{s,u}] \neq \emptyset \tag{4.7}$$

$$[max(\{0, T_{f,l} - T_{s,u}\}), T_{f,u} - T_{s,l}] \cap [D_l, D_u] \neq \emptyset \tag{4.8}$$

The specified start time and duration intervals of n in combination bound uncertainty about the finish time of n, and Condition (4.6) verifies that 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 (4.7) verifies the bounds are consistent with the specified start time interval of n. Condition (4.8) 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 (4.8) 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 $[0, T_{f,u} - T_{s,l}] \cap [D_l, D_u] \neq \emptyset$, 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 line-like ON is defined to be *time consistent* if and only if for all nodes n in the ON, n is node consistent and the flow relation F of the ON satisfies Condition (4.5).

For example, consider the line-like ONs shown in Figure 4.2. 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, $[-\infty, +\infty]$. Using Condition (4.6) above, we can see that the time information of event e_1 in (a) is inconsistent, because its estimated finish time interval is $[T_{s,l} + D_l, T_{s,u} + D_u] = [0910, 1020]$, 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.

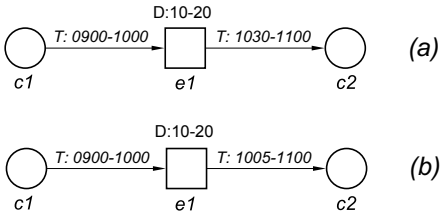


Figure 4.2: Two linear ONs with time information.

4.3.2.2 Time consistency in ONs

In an 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 line-like 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 an ON. The time information of e is defined to be *concurrently consistent* if and only if the following conditions are satisfied:

$$\forall c \in {}^\bullet e \, (I_f^c = I_s^e) \tag{4.9}$$

$$\forall c' \in e^\bullet \, (I_s^{c'} = I_f^e) \tag{4.10}$$

$$e \textit{ is node consistent} \tag{4.11}$$

Thus, for any event e with multiple inputs and outputs, verifying its concurrent consistency involves 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 the start time, finish time, and duration intervals of e satisfy Conditions (4.6), (4.7), and (4.8).

An ON is defined to be *time consistent* if and only if for all conditions c in the ON, c is node consistent, and for all events e in the ON, e is concurrently consistent.

4.3.2.3 Time consistency in alternative ONs

In alternative ONs, each event has at least one input condition and at least one output condition, and each condition has zero or more input and output events.

A condition in alternative ONs can have multiple input and output events that are in different scenarios. The verification of consistency of the alternative ONs

consists in ensuring that each condition is in at least one scenario.

Let c be a condition in an alternative ON. The time information of c is defined to be *alternatively consistent* if and only if the following constraints are satisfied:

$$\exists e \in {}^{\bullet}c \, (I_f^e = I_s^c) \tag{4.12}$$

$$\exists e' \in c^{\bullet} \, (I_f^c = I_s^{e'}) \tag{4.13}$$

$$c \textit{ is node consistent} \tag{4.14}$$

Condition (4.12) states that the start time interval of c is the same as the finish time interval of an input event, and Condition (4.13) states that the finish time interval of c is the same as the start time interval of an output event. Condition (4.14) states that the start time, finish time, and duration intervals of c satisfy Conditions (4.6), (4.7), and (4.8).

An alternative ON is defined to be time consistent if and only if for all conditions c in the ON, c is alternatively consistent, and for all events e in the ON, e is concurrently consistent.

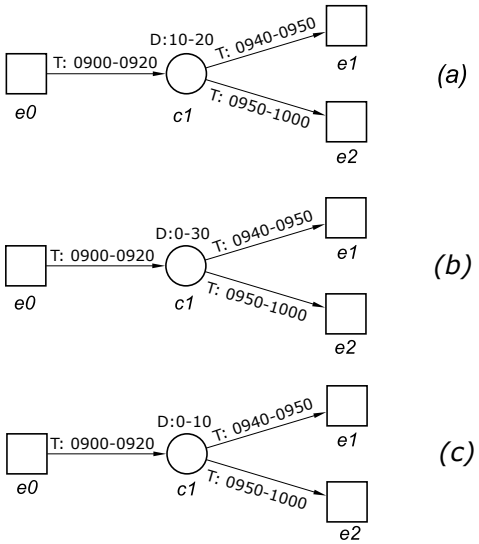


Figure 4.3: Two time consistent AONs (a) and (b), and a time inconsistent AON (c).

During verification of alternative consistency of condition c, we can determine whether or not c belongs to any scenario. Thus, time verification can help investi-

gators to eliminate invalid hypothetical scenarios. For example, Figure 4.3 shows three alternative ON fragments with the same structure and time intervals but different duration intervals. The intervals of c_1 in (a) are alternatively consistent with respect to e_0 and e_1, but are alternatively inconsistent with respect to e_0 and e_2. Therefore, there is only one valid scenario to which c_1 belongs considering the time information, since e_2 cannot happen after the execution of e_0 due to the time inconsistency. Notice that time verification cannot always identify a unique scenario for a given condition: in (b) there are two valid scenarios for c_1 since it is node consistent with both its finish time intervals; and in (c) there is no valid scenario for c_1 since it is node inconsistent with both its finish time intervals.

4.3.2.4 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 *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 *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 *a/synchronously consistent* if and only if the following conditions are satisfied:

$$I_f^e = I_s^q \tag{4.15}$$

$$I_s^{e'} = I_f^q \tag{4.16}$$

$$q \textit{ is node consistent} \tag{4.17}$$

Intuitively, condition (4.15) states that the start time of q equals the finish time of its input event. Condition (4.16) states that the finish time of q equals the start time of its output event. Condition (4.17) states that q must be node consistent with respect to its three time and duration intervals. If q is an asynchronous

channel place, its consistency checking can be regarded as checking a condition that can have a non-zero duration. If q is synchronous channel place, the duration of q is zero due to the cyclic representation of synchronous communication.

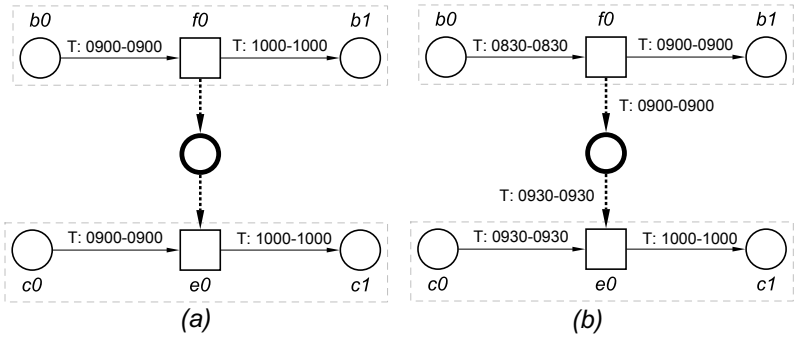


Figure 4.4: Two CSONs with different runs of f_0 and e_0.

Figure 4.4 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.

4.3.2.5 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 (β) and *causal* relations. For simplicity, we assume the different abstraction levels have the same time origin and granularity.

Given a BSON, let $causalU$ be the binary relation consisting of the causally related pairs of events of the BSON that is defined as follows:

$$causalU \triangleq \bigcup_{e \in \mathbf{E}} causal(e)$$

where $\mathbf{E}$ is the set of events in the ONs of the BSON. The time information of $causalU$ is defined to be time consistent if and only if the following condition is satisfied:

$$\forall (g,h) \in causalU \,:\, (T^g_{s,l} \leq T^h_{s,l} \;\wedge\; T^g_{s,u} \leq T^h_{s,u}) \tag{4.18}$$

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 equation must be satisfied:

$$I_s^{c_i} = I_s^{c_i'} \tag{4.19}$$

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 equation must be satisfied:

$$I_f^{c_t} = I_f^{c_t'} \tag{4.20}$$

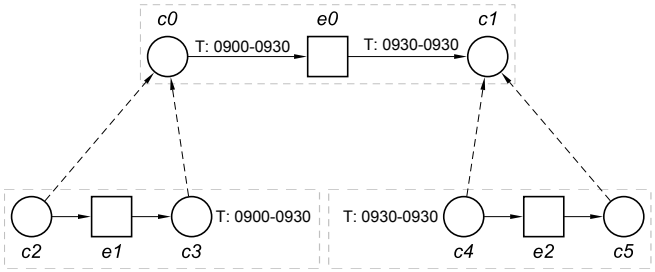


Figure 4.5: Example of a BSON portraying system (off-line) update.

Condition (4.18) states that the start time of event g should be the same as or precede the start time of event h, and extends the *causal* restriction to the entire BSON. Conditions (4.19) and (4.20) impose restrictions on the initial and final states of the BSON: the start (finish) time of a lower level occurrence net must be the same as the start (finish) time of its corresponding upper level condition.

For example, Figure 4.5 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.

4.4 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 $\tilde{X}$, and all specified information is assumed to be consistent using the conditions defined in Section 4.3.

For a given node, the estimations of I_s, I_f, and I_d are defined as follows:

$$\text{Let } \tilde{I}_s \triangleq [\tilde{T}_{s,l}, \tilde{T}_{s,u}] \text{ and } \tilde{I}_f \triangleq [\tilde{T}_{f,l}, \tilde{T}_{f,u}] \text{ and } \tilde{I}_d \triangleq [\tilde{D}_l, \tilde{D}_u] \tag{4.21}$$

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

$$[\tilde{T}_{s,l}, \tilde{T}_{s,u}] = [T_{f,l} - D_u, T_{f,u} - D_l] \cap [T_{s,l}, T_{s,u}] \tag{4.22}$$

$$[\tilde{T}_{f,l}, \tilde{T}_{f,u}] = [T_{s,l} + D_l, T_{s,u} + D_u] \cap [T_{f,l}, T_{f,u}] \tag{4.23}$$

$$[\tilde{D}_l, \tilde{D}_u] = [max(\{0, T_{f,l} - T_{s,u}\}), T_{f,u} - T_{s,l}] \cap [D_l, D_u] \tag{4.24}$$

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:

$$[\tilde{T}_{s,l}, \tilde{T}_{s,u}] = [T_{f,l} - D_u, T_{f,u} - D_l] \tag{4.25}$$

$$[\tilde{T}_{f,l}, \tilde{T}_{f,u}] = [T_{s,l} + D_l, T_{s,u} + D_u] \tag{4.26}$$

$$[\tilde{D}_l, \tilde{D}_u] = [max(\{0, T_{f,l} - T_{s,u}\}), T_{f,u} - T_{s,l}] \tag{4.27}$$

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

(4.25) or (4.26). 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, and the second approach is to estimate the intervals of all the nodes of the SON.

4.4.1 Computation of Time Intervals of a Node

We now describe algorithms for estimating the unspecified time intervals of an individual node. The basic idea is to traverse 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. Since a SON is essentially a directed acyclic graph, any traversal of such a structure can be performed in two directions. In contrast to the function *causalPreset* described in Algorithm 3, Algorithm 4 describes the structure of the function *causalPostset*, which obtains the causal output neighbours of a given node n.

Algorithm 4 (Causal postset)

```
1: function causalPostset(input: n)
2: Postset := ∅
3: for all node n' such that (n, n') ∈ F ∨ (n, n') ∈ W do
4:     add n' to Postset
5: for (e, f) ∈ causalU do
6:     if e = n then
7:         add f to Postset
8: return Postset
```

Algorithm 5 describes the structure of the procedure *estimateFinish*, which computes the finish time interval of a node n using causal relations, and is outlined as follows:

1. Given a node n with an unspecified finish time interval, conduct a forward depth-first-search (DFS) to find all paths such that each path begins at n and ends at the nearest node with a specified finish time interval.

2. For each path, apply Equation (4.25) to compute a possible finish time interval of n, where I_f is the specified finish time interval of the last node in the path and I_d is the accumulated durations of all the nodes in the path except n (default duration intervals are used in the accumulation for nodes with unspecified duration intervals).

3. The estimated finish time interval of n is the intersection of all possible finish time intervals.

Algorithm 5 (Estimate finish time interval using causal relation)

```
1: procedure estimateFinish(Node n)
2: PossibleTimes := ∅      // possible finish time intervals from forward search
3: visited := <>
4: add n to visited
5: forwardDFSDurations(visited)
6: Ĩ_f^n := getOverlapping(PossibleTimes) // Ĩ_f^n is the intersection of all possible
   finish time intervals

7: procedure forwardDFSDurations(List visited)
8: for all m ∈ causalPostset(visited.last) do
9:    I := null                                   // possible finish time interval
10:   if I_f^m is specified then
11:      I := I_f^m − accumulatedDurationsOf((visited\{n}) ∪ {m}) // Eqn.(4.25)
12:      add I to PossibleTimes
13:   else if m ∉ visited then
14:      add m to visited
15:      forwardDFSDurations(visited)
16:      remove visited.last
```

Notice that the procedures for estimating time intervals are restricted to the SON structure with no alternative behaviour (i.e. a particular scenario). This is because alternative scenarios can have different and inconsistent time characteristics, which are required in order to eliminate infeasible scenarios from an investigation. In contrast, the time intervals in an individual scenario must be consistent, because reality is assumed to be consistent. Hence, if a node is in more than one scenario, it is necessary to indicate a particular scenario before the time

estimation can begin so that all determined paths can have the same time characteristics. The estimation can fail to compute a time interval and return a *null* value if there is no intersection between the intervals in *PossibleTimes* (i.e. the scenario contains inconsistent time information) or when the DFS is unable to find any path ending with a specified time interval (i.e. the scenario contains insufficient time information).

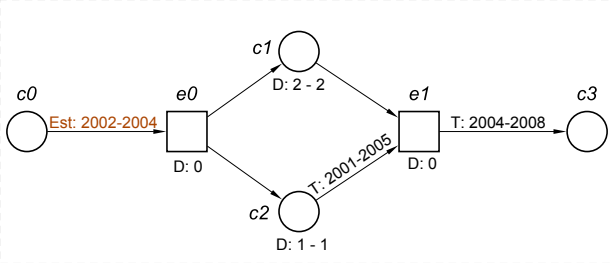


Figure 4.6: Estimating the finish time interval of a node.

Figure 4.6 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: $I_f^{c_2} = I_s^{e_1} = [2001, 2005]$ and $I_f^{e_1} = I_s^{c_3} = [2004, 2008]$, and two duration intervals; $I_d^{c_1} = [0002, 0002]$ and $I_d^{c_2} = [0001, 0001]$. To estimate the finish time interval of c_0, forward DFS is used to find all paths from c_0 to the nearest node with a specified time interval, that is, $\{c_0, e_0, c_1, e_1\}$ and $\{c_0, e_0, c_2\}$. The accumulated duration of the first path is [0002, 0002], which is subtracted from the specified finish time interval of e_1 using Equation (4.25) to calculate a possible finish time interval for c_0 (i.e. $[2004, 2008] - [0002, 0002] = [2002, 2006]$). Similarly, another possible finish time interval is calculated using the second path (i.e. $[2001, 2005] - [0001, 0001] = [2000, 2004]$). The last stage of the procedure is to find the intersection of the intervals in *PossibleTimes* (i.e. $[2002, 2004]$), which is the estimated finish time interval of c_0.

The estimation of the start time interval of a node n is conducted in the reverse way to *estimateFinish*. Thus, *estimateStart* performs backward DFS on the causal preset of node n to find all paths that begin at n and end at the nearest node with a specified start time interval. For each path, the durations of its nodes (except n) are accumulated to compute all possible start time intervals of n using

Equation (4.26), and the computed intervals are intersected in order to estimate the start time interval of n.

4.4.2 Computation of Time Intervals of a SON

In the previous sub-section, we showed how causally related time information can be used to compute the estimated time values of a given node. The approach takes any node with unspecified time information as the input, and then traverses the model in two directions in order to find specific times for the time-related calculation.

Algorithm 6 (Estimate time intervals for entire SON)

procedure estimateSONTimesfwdDFS(SON S)
input: SON S with specified scenario
output: SON S with estimated time intervals of all nodes in the scenario with unspecified time intervals

add pre-initial node e to S
add all causal relations (e, x) to S such that $x \in M_0^S$, where M_0^S is the set of initial conditions of S
$\tilde{I}_f^e := estimateFinish(e)$
$visited := \varnothing$
$forwardDFSTimes(e)$
remove pre-initial node e and its relations from S

procedure forwardDFSTimes(Node n)
add n to $visited$
for all $m \in causalPostset(n)$ **do**
  if $m \notin visited$ **then**
    if I_s^m is unspecified **then**
      $I_s^m := I_f^m$
    if I_d^m is unspecified **then**
      $I_d^m := I_d^{default(typeof(m))}$
    if I_f^m is unspecified **then**
      $I_f^m := I_s^m + I_d^m$ // Eqn. (4.26)
    $forwardDFSTimes(m)$

We now introduce an approach for estimating the time values for all nodes

with unspecified time information in a SON. Algorithm 6 shows the basic idea. Unlike the algorithms for single node time estimation where the start point of DFS is a user-specified node, the start point of the entire SON estimation uses a virtual fixed node called *pre-initial*. The pre-initial node is not a real node of the SON but is a dummy node that is causally related to all the initial conditions of the SON. (This transformation is often used in solving the problems of direct acyclic graphs, for example, maximum flow problem [43].) Figure 4.7 shows a SON with two initial conditions c_0 and c_1. The pre-initial node of the SON is s which is connected to both initial conditions; c_2 is not connected to s since it is not an initial condition of the SON.

Having created the pre-initial node, we are able to compute an estimated finish time for the node by applying Algorithm 5. This value is the start time of all the initial conditions without specified start times. Then the procedure *forwardDFSTimes* traverses the SON in a forward direction, and each visited node with incomplete time information can be assigned a start time taken directly from the finish time of its input node, and a finish time computed using the new assigned start time and its duration. There is no special stop condition for the DFS, and the procedure continues to traversing the SON until the final state is reached which is the default stop condition of the DFS on direct acyclic graph. The pre-initial node we added at the beginning of the algorithm is not necessarily preserved permanently. The last stage of the algorithm is to remove this dummy node and its associated relations from the SON.

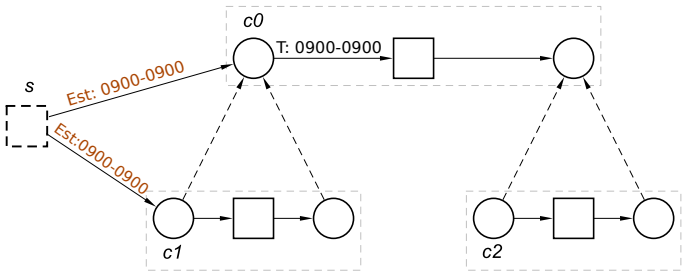


Figure 4.7: A BSON with a pre-initial node.

4.5 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. [58] and [52]) and on temporal logics (e.g. [27]) that addresses these issues. In [58], 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 [52], 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 [27], 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.

4.6 Conclusion

This chapter 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 in a SON in order to estimate missing time intervals and to increase the precision of user-specified intervals.

The implementations of SONs and timed SONs are described in the next chapter.

Chapter 5

SONCraft: A Tool for Construction, Simulation and Verification of Structured Occurrence Nets

5.1 Introduction

In the previous chapters, we described the concept of SONs and time in SONs. Implementations supporting these concepts have been developed. This chapter reports on these implementations.

The visual editing of SON models, their verification, simulation, and analysis are the functionalities supported by the SONCraft toolkit. The toolkit is implemented as a plug-in module within Workcraft [23], a system that provides a flexible framework for the development and analysis of Interpreted Graph Models (IGM) [22]. The framework is built using a plugin-based architecture and supports run-time scripting, which makes it easily extensible to new IGM-based formalisms, and to the provision of support of their analyses and verification methods. It also provides a GUI environment that facilitates model entry and supports interactive visual simulation, together with convenient "single-click" verification. So far, several modules have been implemented and supported by

the platform, including Structured Occurrence Nets (SONCraft), Petri nets, and other Petri net-based formalisms, e.g. Signal Transition Graphs (STGs) [11] and Conditional Partial Order Graphs (CPOGs) [9]. A detailed Workcraft description and manual can be found in [4].

In this chapter, we give an overview on SONCraft's functionality and architecture. The present version of SONCraft deals with the three types of SON variants that have been discussed in Chapter 2, i.e. CSONs, BSONs, and TSONs. This chapter is organised as follows. In Section 5.2, we present the functionalities regarding SONs concept. Section 5.3 presents the implementation of timed-SONs and its related analysis methods. Section 5.4 describes the tool architecture, and the way in which SONCraft integrates with the Workcraft framework. Section 5.5 provides installation information, and Section 5.6 concludes the chapter.

5.2 SON-based Functionality

This section presents an overview of the major SON-based features provided by SONCraft.

5.2.1 SONCraft Overview

The graphical interface of SONCraft is depicted in Figure 5.1. The *Main menu* provides the functions to manage, edit and analyse models. For example, the *Tools* menu provides a set of user-friendly analysis tools for checking models; and there is a vector graphics export function in the *File* menu (all SON models shown in this thesis were imported directly from SONCraft with minimal modifications). The *Editor tabs* line shows the names of all of the opened models and allows the user to choose which one is to be displayed in the *Editor window*. The latter is the place for the user to create, edit and simulate a SON model.

SONCraft defines several kinds of graphical nodes and connection types. These are displayed in the *Editor tools* panel that allows the user to create and edit SON-based models. The *Property editor* panel at the top-right hand side of Figure 5.1 is used to support various visual node editing operations, e.g. to change the label, colour, or position of a node. The *Tool controls* panel provides access to

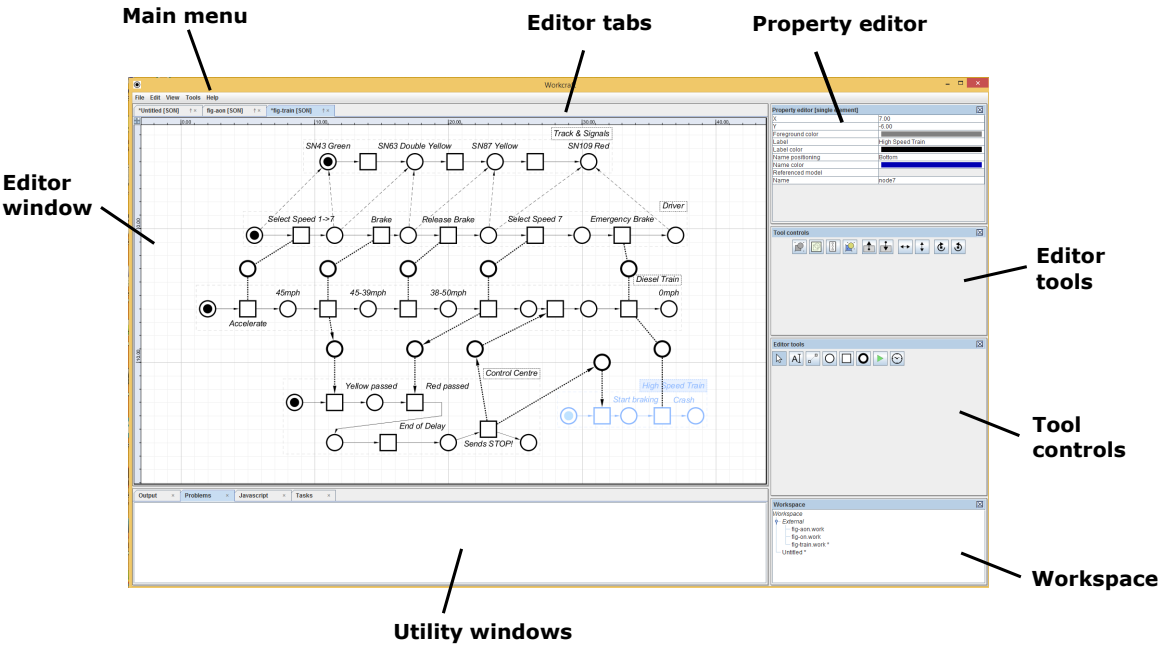


Figure 5.1: SONCRAFT interface

the extended functionality of a selected tool. For example, when the connection tool is activated, the user is able to switch between various connections in order to construct different types of SON abstractions. The *Workspace window* lists opened or imported work files. One can also operate on a work file (delete, save, etc). The *Utility window* is used for showing additional information concerning the progress of currently executed tasks, verification results, and information about any errors that may have occurred during execution.

5.2.2 Editor Tools

SONCRAFT offers a set of editor tools for constructing and editing SON models. Some generic tools for editing models are directly inherited from the WORKCRAFT framework, including selection, text note, flip horizontal, flip vertical, rotate clockwise, and counter-clockwise functions. Others tools for constructing SON models, defined specifically in SONCRAFT, are:

- The *SON component toolkit* contains a group of buttons for creating SON-based components in the editor window. The toolkit contains condition, event and channel place creators, where the first two creators are used for constructing ON, and the latter is for CSON construction.

- The *Connection tool* is used for creating relations between nodes. The tool provides several connection types that can be chosen to construct various different abstractions (i.e. causal relations F, weak causal relations W, and behavioural relations β). The tool also offers a basic relation validation facility, based on the SON definition. Any invalid user operation, e.g. connecting two conditions, will trigger an error message. (However not all user errors are detected at this stage, and those that cannot be immediately detected can be checked using the verification tool âĂŞ– see the section below).

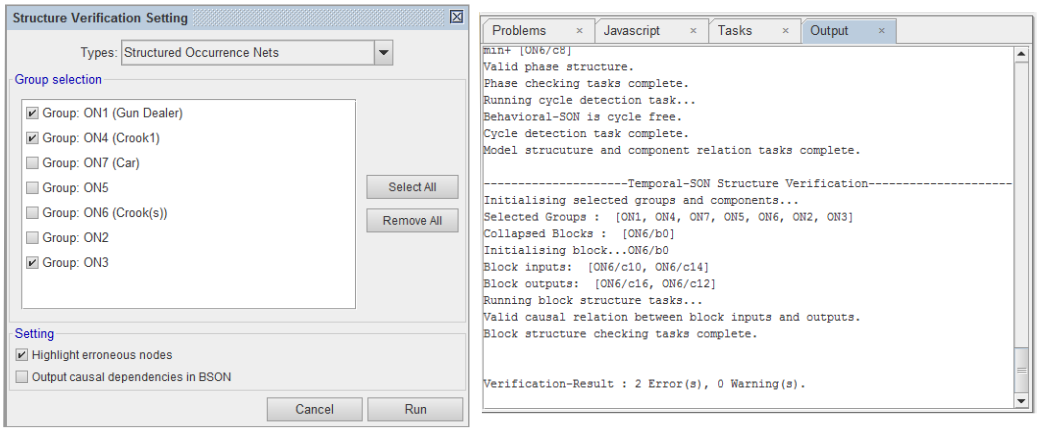
- The *Group tool* allows the user to combine a set of nodes (in particular, conditions, events, and causal relations) into a group. In SONCraft, an ON is not recognised until it has been delineated as a group. Thus, a SON model in SONCraft is generally composed of a set of groups representing component ONs, and the relations (abstractions) between the groups' components.

- The *Block tool* creates atomic actions in TSONs. Similarly to the group tool, a block in SONCraft is implemented as a container holding a set of user selected components. It can be collapsed into a single node causing its components to be hidden.

5.2.3 Structural Analysis Tool

The structural analysis tool provides the user with a set of 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. The verification criteria follow from the formal definitions and properties introduced in Chapter 2. The tool consists of two sub-checkers:

The *relation property checker* deals with the relation-based correctness of a SON model. The checking includes conflict-freeness, phase decomposition, component ONs disjointness, and block causality (see Section 2.6.1). The *acyclic property checker* implements Algorithm 2 and focuses on the acyclicity condition of SONs. The verification of such a property comes down in practice to searching strongly connected components (SCC) in a SON model. The checker applies Tarjan's algorithm as a core engine to compute maximal SCCs. A specific filter will be invoked at the end stage in order to obtain the desired results.



(a) *(b)*

Figure 5.2: (a) Structural verification setting dialog; and (b) a verification report.

The verification setting dialog (Figure 5.2(a)) allows users to specify the particular SON's model type as well as which groups (ONs) in this SON they would like to verify. Such partial checking works efficiently for a SON consisting of a large set of ON fragments and relations [15] (e.g. representing a major accident or a large scale criminal activity). The results of the verification are detailed in a verification report (Figure 5.2(b)), and are shown by colouring any offending parts of the SON model.

5.2.4 Simulator

SONCraft offers a built-in simulator for SONs simulation. The underlying semantics of SON-based simulation follows the firing rules presented in Chapter 2. The simulation function in SONCraft can be activated by clicking on the simulation button in the editor tools panel. The initial marking will be automatically set, and all enabled events will be highlighted.

The simulation can then be conducted either manually or automatically, by firing a succession of enabled events, causing tokens to move, event highlighting to be updated, and the simulation record augmented.

The simulation tool controls provide the means to analyse and navigate a previously recorded simulation. There are two sources of data for a simulation record: a 'branch' records the firing sequence of events that were executed by explicitly clicking the enabled nodes of the model, and 'traces' are automatically generated from other tools, e.g. the reachability tool. Thus one is able to generate simulation records from different executions in order to perform a comparison. The panel also provides access to several additional simulation functions, most of which relate to the simulation traces or branches (see Figure 5.3). Some of the main features are listed below:

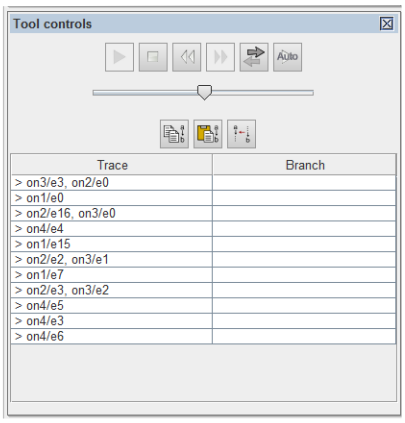


Figure 5.3: Simulation control panel

- *Playback* - to automatically play back an existing trace or branch, at a selected speed.

- *Step forward/backward* - to step forward/backward through an existing trace or branch, highlighting the record reached.
- *Reverse/Forward simulation* - to change the simulation directions.
- *Automatic simulation* - to causes simulation to occur, using maximum parallelism through to the end.
- *Copy/Paste* - to copy a trace or branch to the clipboard, and reload the stored trace or branch from the clipboard.

The SON simulator provides a failure analysis function, called error tracing. When the failure analysis function is turned on, each event has an associated *fault bit*, which is a '1' or a '0'. This bit can be used to indicate whether one wishes to regard the event as a faulty one, with '1' indicating a simulated fault. An error count is also shown in the editor window below each condition, and is set initially to '0'. This count cannot be changed manually by the user. Rather, it is automatically calculated during simulation to indicate for each condition the number of faults that have been passed on the forward route to that condition.

5.2.5 Reachability Tool

Once a SON model is complete and its structure is valid, the user can perform model checking. SONCraft provides a reachability checker which implements Algorithm 3 for verifying reachability. This is used to analyse whether or not a given marking can be reached from the initial marking. If the marking is reachable, a request can be made for the trace that leads to the marking to be passed to the simulation tool for playback or further analysis.

5.3 Time-based Functionality

We have implemented timed-SONs and their analysis algorithms introduced in Chapter 4 in SONCraft. This section presents the implementation.

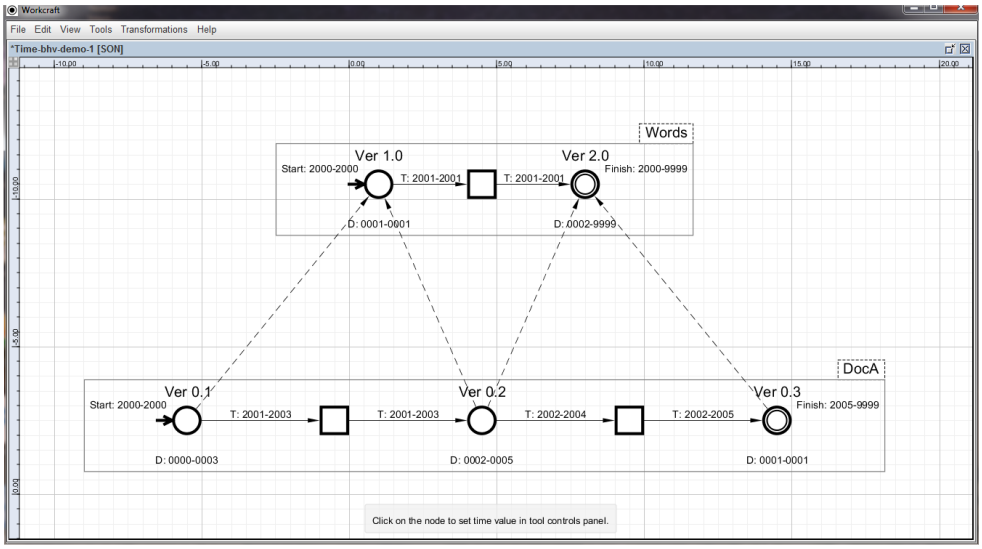


Figure 5.4: Time SONs visualisation.

5.3.1 Visualisation

The *time mode* in SONCraft enables the display of the time information in a SON model, as shown in the screenshot in Figure 5.4. 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 small thick 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 displaying 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 5.4, the start time

interval $[2000, 2000]$ of the initial condition 'Ver 1.0' is displayed directly next to the node.

5.3.2 Time Property Setting Tool

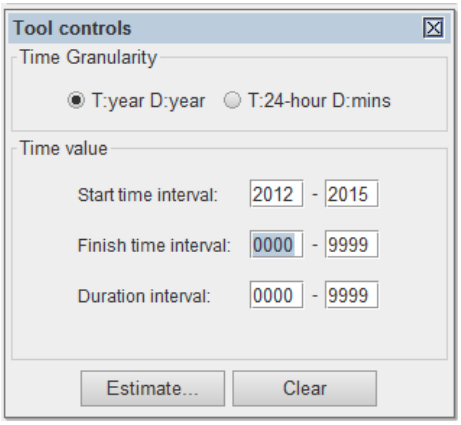
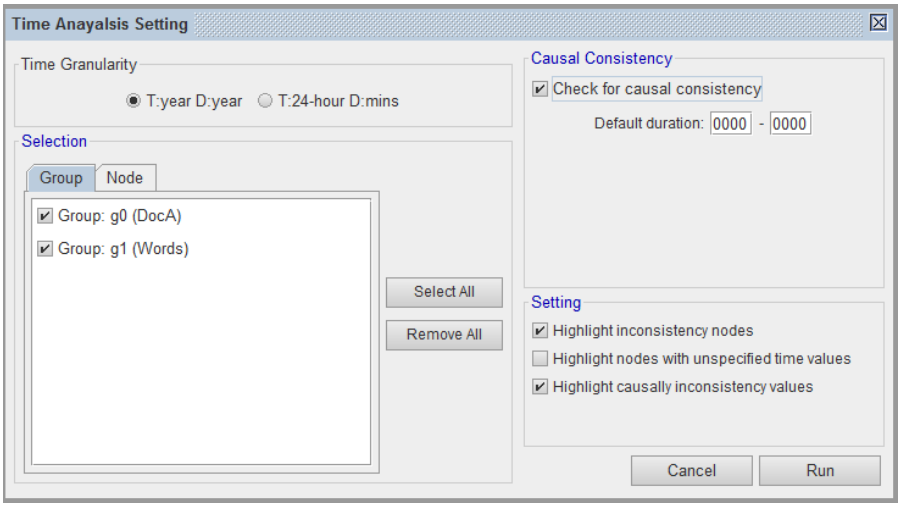


Figure 5.5: Time property setter

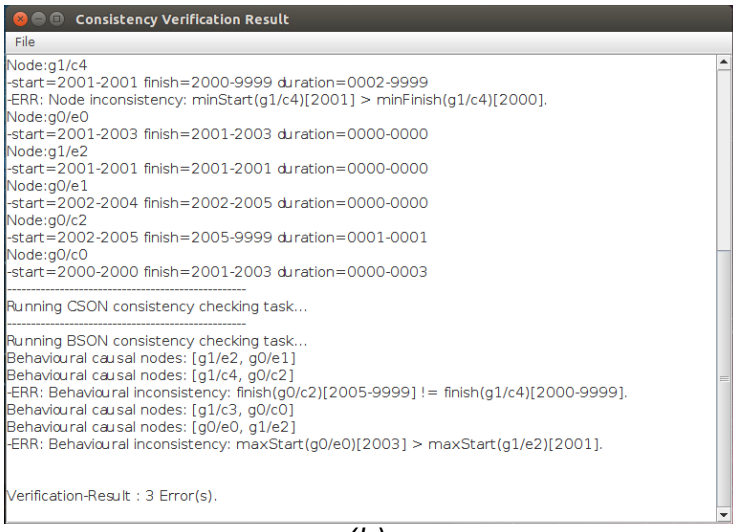
The *time property setting tool* shown in Figure 5.5 is an interface for specifying the time information of a given node in a SON model. The *time granularity* panel at the top of the interface currently offers two granularities: year/year and 24-hour-clock/mins. Different granularities have their own 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 (4.1) – (4.4). Moreover, for each input interval, the checking of its time or duration bounds is performed according to its time granularity. The tool also provides time estimation for nodes with unspecified time intervals. Depending on the user selection, different algorithms (e.g. Algorithm 5 or the entire SON estimation) will be called for the computation.

5.3.3 Consistency Checking Tool



(a)



(b)

Figure 5.6: (a) Time consistency tool setting dialog; and (b) part of the consistency checking result of the SON in Figure 5.4.

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 4.3, and provides a user interface for additional settings. Figure 5.6

(a) shows the tool interface. The interface is divided into the following four (sub-)panels.

Two granularities are present in the *time granularity* panel. The *selection* panel allows the user to perform a consistency checking in two ways. The group selection lists all component occurrence nets of a SON. The user can either choose to verify a partial SON (i.e. number of its occurrence nets together with their related CSON and BSON relations) or a complete SON (i.e,. all occurrence nets together with the relations). While the node selection supports the verification of the consistency of any selected node. Causal consistency checking can be activated using the *causal consistency* panel. The facility implements Algorithm 13 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 highlighting of time inconsistent nodes (if any) in the editor window after verification.

Figure 5.6 shows a consistency verification result of the SON displayed in Figure 5.4. 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.

5.4 Tool Architecture

SONCraft is written in JAVA, making it accessible on all platforms for which there exists a JVM. The architecture depicted in Figure 5.7 shows a detailed view of the integration between the Workcraft framework and SONCraft.

5.4.1 Workcraft Architecture

The Workcraft framework consists of the following three parts:

The *Core framework* is in charge of the initialisation of Workcraft, managing plug-ins and the provision of common services to the plug-ins. When the program starts up, services such as the configuration manager and the frame-

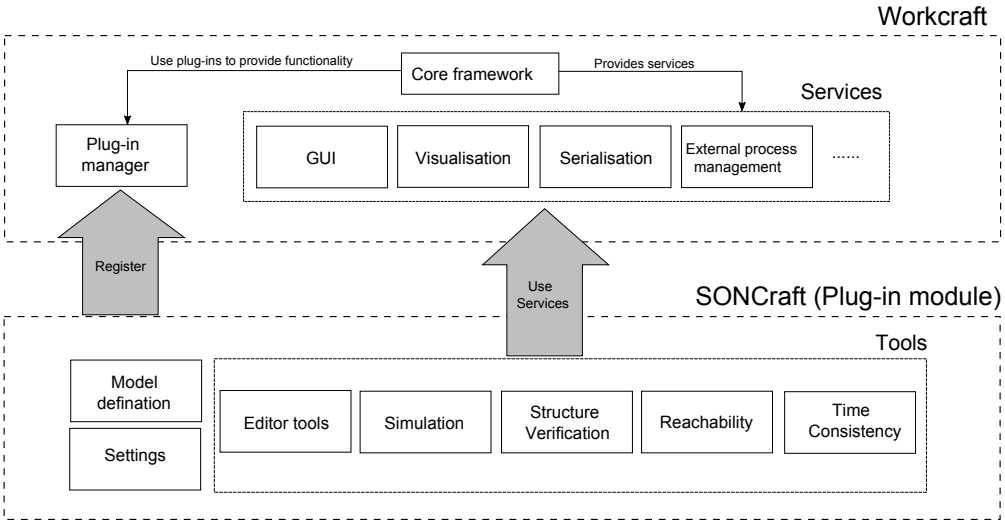


Figure 5.7: Tool architecture.

work GUI are initialised. This is followed by the initialising of the plug-in manager, which provides the facility for loading all existing plug-ins. On shut-down, WORKCRAFT saves the configuration of the framework; it restores it on the next start-up.

The *Plug-in manager* is responsible for scanning and loading all plug-in modules which have been registered in the manager. A plug-in module is a related collections of plug-ins that together implement a specific functionality, for instance the SONs module. For each plug-in module, the manager also maintains a list of its internal facilities. During the initialisation the plug-in manager uses the list to load the contents of plug-ins instead of scanning the plug-ins directory every time.

The *Services* are fully managed by WORKCRAFT and accessible to the plug-ins. The GUI service provides the facilities for creating editor, tool and information windows. A number of advanced GUI capabilities, such as the multiple document interface and full-screen mode, are also supported. The Visualisation service facilities provide editing functions for the node types defined by any model, for instance, drawing, transformation and auxiliary editing operations. The Task management service is responsible for executing all external process tasks – it maintains the list of all running tasks and uses a separate thread for the parallel

execution.

5.4.2 SONCraft Integration

SONCraft is deployed in the Workcraft framework as an individual plug-in module. There are three main components inside the module:

The *Model definition* component describes the basic features of a SON model. The component is divided into mathematical and visual levels in order to avoid mixing unrelated responsibilities. The mathematical model describes all the semantics concerning model integrity — it keeps information such as connection types and node names. The visual model is a manageable interface between the user and both the mathematical and the visual models. The visual model defines how to draw/present SON models as well as maintaining visual information, such as colour, position and label.

The *Settings* component records the default properties of a SON model and stores them in a configuration XML file. The Workcraft start-up process loads the stored settings and allows other components to read their configuration variables.

The *Tools* component manages all the external and built-in tools in SONCraft. The implementation of each component tool uses the services provided by the Workcraft framework. For example, the editor tools and simulation facilities rely on the GUI and visualisation services for node placement, trace table creation, and so on. The structural verification, reachability checking and time consistency tools invoke external process management for monitoring and managing the tasks.

5.5 Installation

The latest version of SONCraft is available from [2]. It is necessary to have a compatible Java Runtime Environment (JRE) version 7 or higher in order to run SONCraft. The standard JRE can be downloaded from [1]. There is no automatic installer for SONCraft; to install it, the files from the link archive need to be extracted manually. A comprehensive user manual can be found in [12].

In [3], we also provide a tutorial showing how to use SONCraft for modelling and analysing crime and accident scenes.

5.6 Conclusion

In this chapter, we introduced the SONCraft toolkit for construction, simulation and verification of SONs. SONCraft provides a user-friendly graphical interface and a set of editor tools enabling the user to construct models easily and quickly. The simulator tool implements the firing rules described in Chapter 2 for SON-based model simulation. The structural analysis tool provides a number of algorithms for validating the structural correctness of a SON model. The reachability tool is used to perform reachability checking for a given marking.

In addition to the SON-based functionality, we have also implemented timing properties and their related analysis algorithms introduced in Chapter 4 to SONCraft. The time property setting tool can visually set time values for any selected node. The time consistency checking tool is used to verify the consistency of the user specified time values.

In the next chapter, we discuss a generator net of SONs and its unfolding.

Chapter 6

Unfolding CSPT-nets

6.1 Introduction

The structured occurrence nets concept introduced in Chapter 2 are directed acyclic graphs used to record execution histories of complex evolving systems. Similarly to the occurrence nets, one can in fact derive a SON in two different ways: (i) as a direct representation of an actual or imagined system's execution history; or (ii) as a process underpinning a run of some generator nets. The generator net in the second approach can be regarded as a specification of a system's design which describes complete system behaviour and is generally represented as cyclic structures. In [28], the authors investigate a generator net of CSONs called *Communication Structured Place Transition nets* (CSPT-nets). The nets are built out of the place/transition nets (PT-nets), which are connected by channel places allowing both synchronous and asynchronous communication. Figure 6.1(a) and (b) shows two CSPT-nets which consist of two interacting component PT-nets.

Once such a generator system model is constructed, suitable verification methods can be applied to formally check whether or not the model has the desired behaviour. The leading method is model checking [21], an automatic verification technique that is able to ascertain the correctness of a computing system. This technique takes a specification of desired properties as inputs, then operates over a finite-state model to automatically verify whether the properties are satisfied. Model checking has had tremendous impact on both academia and industry, and

its inventors were given the 2007 ACM Turing Award.

The main drawback of model checking is that it suffers from the state space explosion problem. That is, even a relatively small system model can (and often does) yield a very large state space. For example, consider a system composed by n processes, each having m states. Then, the asynchronous composition of these processes may have m^n states.

There are many methods have been proposed to alleviate the state space explosion problem, such as, symbolic model checking [34], unfoldings [24, 45] and partial order reductions [10]. Among them, the standard Petri nets unfoldings, introduced in [35], are a technique supporting effective verification of concurrent systems modelled by Petri nets (throughout this chapter, Petri net related concepts, such as configuration and unfolding, will be referred to as *standard*). The method relies on the concept of net unfolding which can be seen as the partial order behaviour of a concurrent system. The unfolding of a net is usually infinite, but for bounded Petri nets one can construct a finite complete prefix of the unfolding containing enough information to analyse important behavioural properties [26, 35]. More importantly, such prefix is often exponentially smaller than the corresponding reachability graphs, especially if the system at hand exhibits a lot of concurrency, as it is the case in our complex evolving system contexts.

In this chapter, we introduce branching processes of CSPT-nets (CSPT-net unfoldings). As in the standard net theory, CSPT branching processes act as a 'bridge' between CSPT-nets and their processes captured by CSONs (i.e. the branching processes of a CSPT-net contains a representation of all the possible single runs of the original net). In order to reduce the complexity of branching processes of CSPT-nets, we adapt the notion of occurrence depth which was originally developed for merged processes [59]. Moreover, we discuss several key properties of branching processes of CSPT-nets, and also present an algorithm for constructing CSPT-net unfoldings. The algorithm takes into account the occurrence depth of events, and fuses nodes which have same behaviours during the unfolding. In this way, the size of the resulting net can be significantly reduced when compared with the standard unfolding approach.

Consider again the CSPT-nets shown in Figure 6.1(a) and (b). In (a), m transitions are synchronous with n transitions between two PT-nets via two channel

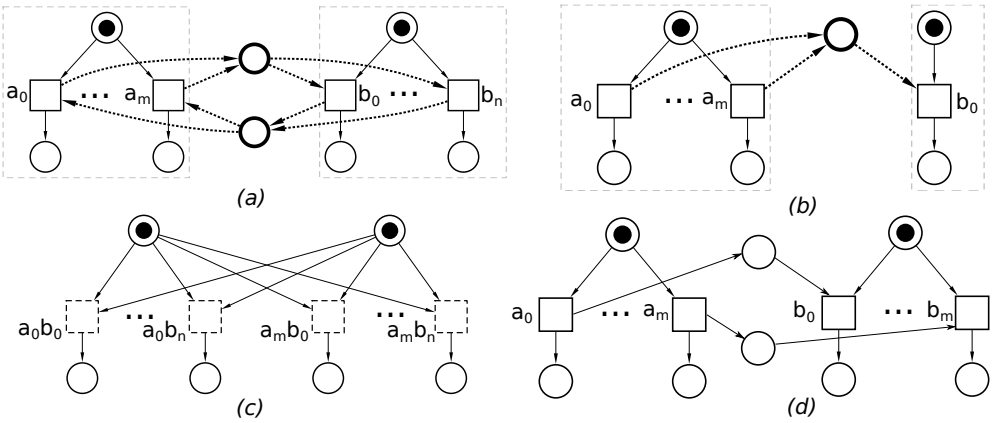


Figure 6.1: Two CSPT-nets (a) and (b); together with their respective standard unfoldings semantics after applying the Petri net encodings (c) and (d).

places. In (b), m transitions asynchronously communicate with b_0 via a single channel place. Their unfolding semantics are isomorphic to the original CSPT-nets (with the sizes of $m + n$ events in (a) and $m + 1$ events in (c)). If one was only interested in marking reachability, then one might attempt to encode a CSPT-net by replacing every asynchronous channel place by a standard place and 'glue' transitions forming a synchronous event into a single one. One would then be able to apply the standard unfolding to this Petri net based representation. However, the efficiency of such an approach would suffer from the introduction of many new transitions, as well as the loss of the merging on channel places which is due to the exploitation of occurrence depth. In this case, the 'glue' encoding for (a) yields $m \times n$ events in the corresponding unfolding (c). While the 'replace' encoding for (b) would yield $m + n$ events as shown in (d).

The chapter is organised as follows. Section 6.2 provides basic notions concerning Petri nets and their unfoldings. Section 6.3 presents the main concepts of communication structured net theory, including CSON-nets, CSPT-nets and CSPT branching processes. In section 6.4, we discuss finite and complete prefixes of CSPT branching processes and related properties. The CSPT unfolding algorithm is provided in Section 6.5. Section 6.6 concludes the chapter.

6.2 Basic Definitions

In this section, we recall the basic notions concerning Petri nets and their unfoldings. For the reader not familiar with these concepts, [24, 35, 62] cover the topics in depth. Throughout the chapter, a *multiset* over a set X is a function $\mu : X \to \mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, \ldots\}$. A multiset may be represented by explicitly listing its elements with repetitions. For example $\{a, a, b\}$ denotes the multiset such that $\mu(a) = 2$, $\mu(b) = 1$ and $\mu(x) = 0$ for $x \in X \backslash \{a, b\}$.

6.2.1 PT-nets

A *net* is a triple $N = (P, T, F)$ such that P and T are disjoint sets of respectively *places* and *transitions* (collectively referred to as *nodes*), and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow* relation. The *inputs* and *outputs* of a node x are defined as ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. Moreover, ${}^\bullet x^\bullet = {}^\bullet x \cup x^\bullet$. It is assumed that the inputs and outputs of a transition are non-empty sets.

Two nodes, x and x', are in *conflict* if there are distinct transitions, t and t', such that ${}^\bullet t \cap {}^\bullet t' \neq \varnothing$ and $(t, x) \in F^*$ and $(t', x') \in F^*$. We denote this by $x \# x'$[1]. A node x is in *self-conflict* if $x \# x$.

A *place transition net* (PT-net) is a tuple $\mathrm{PT} = (P, T, F, M_0)$, where (P, T, F) is a finite net, and $M_0 : P \to \mathbb{N}$ is the *initial marking* (in general, a marking is a multiset of places).

A *step* U is a non-empty multiset of transitions of PT. It is *enabled* at a marking M if

$$M(p) \geq \sum_{t \in p^\bullet} U(t) \ ,$$

for every place p. In such a case, the *execution* of U leads to a new marking M' given by

$$M'(p) = M(p) + \sum_{t \in {}^\bullet p} U(t) - \sum_{t \in p^\bullet} U(t) \ ,$$

for every $p \in P$. We denote this by $M[U\rangle M'$. A *step sequence* of PT is a

[1] The conflict relation and the alternative relation introduced in Chapter 3 are essentially identical except for that the former one describes a relationship between transitions while the latter one is for events. Therefore we use the same symbol # to indicate both relations.

sequence $\lambda = U_1 \dots U_n$ ($n \geq 0$) of steps such that there exist markings $M_1, \dots, M_n$ satisfying:

$$M_0[U_1\rangle M_1, \dots, M_{n-1}[U_n\rangle M_n$$

The *reachable markings* of PT are defined as the smallest (w.r.t. $\subseteq$) set *reach*(PT) containing M_0 and such that if there is a marking $M \in$ *reach*(PT) and $M[U\rangle M'$, for a step U and a marking M', then $M' \in$ *reach*(PT).

PT is *k-bounded* if, for every reachable marking M and every place $p \in P$, $M(p) \leq k$, and *safe* if it is 1-bounded. Markings of a safe PT-net can be treated as sets of places.

6.2.2 Branching Processes of PT-nets

A net $O = (P, T, F)$, with places and transitions called respectively *conditions* and *events*, is a *branching occurrence net* if the following hold:

- F is acyclic and no transition $t \in T$ is in self-conflict;
- $|{}^\bullet p| \leq 1$, for all $p \in P$; and
- for every node x, there are finitely many y such that $(y, x) \in F^*$.

The set of all places p with no inputs (i.e. ${}^\bullet p = \varnothing$) is the default initial state of O, denoted by M_O. In general, a *state* is any set of places.

If $|p^\bullet| \leq 1$, for all $p \in P$, then O is a *non-branching* occurrence net. Note that unlike alternative occurrence nets, in a branching occurrence net, two paths outgoing from a place will never meet again by coming to the same place (the inputs of places are at most singleton sets) nor the same transition (transitions cannot be in self-conflict).

A *branching process* of a PT-net PT $= (P, T, F, M_0)$ is a pair $\Pi = (O, h)$, where $O = (P', T', F')$ is a branching occurrence net and $h : P' \cup T' \to P \cup T$ is a mapping, such that the following hold:

- $h(P') \subseteq P$ and $h(T') \subseteq T$;
- for every $e \in T'$, the restriction of h to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet h(e)$, and similarly for $e^\bullet$ and $h(e)^\bullet$;

- the restriction of h to M_O is a bijection between M_O and M_0; and
- for all $e, f \in T'$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$.

Two branching processes $\Pi' = (O', h')$ and $\Pi = (O, h)$ of a PT-net are *isomorphic* if there is a bijective homomorphism h from Π' to Π such that $\Pi' \circ h = \Pi$. It is shown in [24] that a PT-net has a unique (up to isomorphism) maximal (w.r.t. prefix relation) branching process, called the *unfolding* of PT.

6.2.3 Configurations and Cuts of a Branching Process

Let $\Pi = (O, h)$ be a branching process of a PT-net PT, and $O = (P', T', F')$.

A *configuration* of Π is a set of events $\mathbb{C} \subseteq T'$ such that $\neg(e \# e')$, for all $e, e' \in \mathbb{C}$, and $(e', e) \in F'^{+} \implies e' \in \mathbb{C}$, for every $e \in \mathbb{C}$. In particular, the *local configuration* of an event e, denoted by $[e]$, is the set of all the events e' such that $(e', e) \in F'^{*}$. The notion of a configuration captures the idea of a possible history of a concurrent system, which must be conflict-free, and causally closed, i.e. all the predecessors of a given event must have occurred.

A *co-set* of Π is a set of conditions $B \subseteq P'$ such that, for all distinct $b, b' \in B$, $(b, b') \notin F'^{+}$. Moreover, a *cut* of Π is any maximal (w.r.t. $\subseteq$) co-set B.

Finite configurations and cuts of Π are closely related:

- if $\mathbb{C}$ is a finite configuration of Π, then $Cut(\mathbb{C}) = (M_O \cup \mathbb{C}^\bullet) \setminus {}^\bullet\mathbb{C}$ is a cut and $Mark(\mathbb{C}) = h(Cut(\mathbb{C}))$ is a reachable marking of PT; and
- if M is a reachable marking of PT, then there is a finite configuration $\mathbb{C}$ of Π_{PT} such that $Mark(\mathbb{C}) = M$.

Hence every marking represented in the unfolding Π_{PT} is reachable in PT, and every reachable marking of PT is represented in Π_{PT}.

6.3 Structuring PT-nets

In this section we address the formal definitions concerning communication structured nets theory, including CSPT-nets, its branching processes (BCSO-nets) and single runs (CSOs).

In general, a CSO represents a single run of interacting systems due to its conflict-freeness and acyclicity. CSPT-nets are generator nets of CSOs that represent complete systems behaviours, and similarly as in the case of PT-nets, its branching processes will capture full execution information of the corresponding CSPT-nets.

6.3.1 CSPT-nets

By generalising the definition of [28], we first introduce an extension of PT-nets which combines several such nets into one model using channel places.

Definition 6.1 (CSPT-net)**.** *A* communication structured place transition net *(or* CSPT*-net) is a tuple*

$$\text{CSPT} = (\text{PT}_1, \ldots, \text{PT}_k, Q, W, M_0) \quad (k \geq 1)$$

such that each $\text{PT}_i = (P_i, T_i, F_i, M_i)$ *is a safe (component)* PT*-net;* Q *is a finite set of* channel *places;* $M_0 : Q \to \mathbb{N}$ *is the initial marking of the channel places; and* $W \subseteq (T \times Q) \cup (Q \times T)$*, where* $T = \bigcup T_i$*, is the flow relation for the channel places.*

It is assumed that the following are satisfied:

1. *The* PT_i*'s and* Q *are pairwise disjoint.*

2. *For every channel place* $q \in Q$*,*

   - *the sets of inputs and outputs of* q*,*

     $$^\bullet q = \{t \in T \mid (t, q) \in W\} \quad \textit{and} \quad q^\bullet = \{t \in T \mid (q, t) \in W\}\,,$$

     are both nonempty and, for some $i \neq j$*,* $^\bullet q \subseteq T_i$ *and* $q^\bullet \subseteq T_j$*; and*

   - *if* $^\bullet q^\bullet \cap T_i \neq \varnothing$ *then there is no reachable marking of* PT_i *which enables a step comprising two distinct transitions in* $^\bullet q^\bullet$*.* ◇

The initial marking M_{CSPT} of CSPT is the multiset sum of the M_i's ($i = 0, 1, \ldots, k$), and a marking is in general a multiset of places, including the channel places.

To simplify the presentation, in the rest of this paper we will assume that the channel places in the initial states of CSPT-nets are *empty*.

The execution semantics of CSPT is defined as for a PT-net except that a step of transitions U is *enabled* at a marking M if, for every non-channel place p,

$$M(p) \geq \sum_{t \in p^\bullet} U(t) \ ,$$

and, for every channel place q,

$$M(q) + \sum_{t \in {}^\bullet q} U(t) \geq \sum_{t \in q^\bullet} U(t) \ . \qquad (*)$$

The *step sequence, reachable marking* and *safeness* of CSPT-nets are then defined in the same way with the PT-nets.

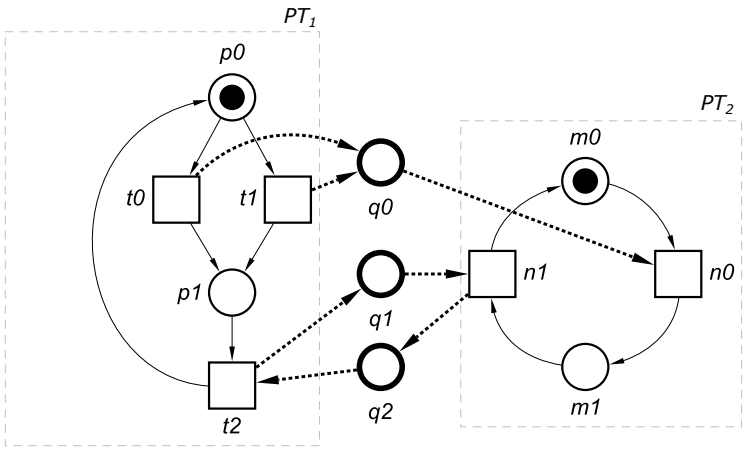


Figure 6.2: A CSPT-net with two component PT-nets.

Definition 6.1(2) means that the occurrences of transitions in ${}^\bullet q$ (as well as those in $q^\bullet$) are totally ordered in any execution of the corresponding component net PT_i. In other words, we assume that both the output access and the input access to the channel places is *sequential*. This will allow us to introduce the notion of depth at which an event which accessed a channel place has occurred.

Figure 6.2 shows a CSPT-net which consists of two component PT-nets connected by a set of channel places (represented by circles with thick borders). To improve readability, the thick dashed lines indicate the flow relation W. Transitions t_2 and n_1 are connected by a pair of empty channel places, q_1 and q_2, forming a cycle. This indicates that these two transitions can only be executed synchronously. These places will be filled and emptied synchronously when both

t_2 and n_1 participate in an enabled step. On the other hand, the execution of transitions t_0 and n_0 can be either asynchronous (t_0 occurs before n_0), or synchronous (both of them occur simultaneously). A possible step sequence of Figure 6.2 is $\lambda = \{t_0\}\{n_0\}\{t_2, n_1\}$, where t_0 and n_0 perform an asynchronous communication. Another step sequence $\lambda' = \{t_0, n_0\}\{t_2, n_1\}$ shows that t_0 and n_0 can be also executed synchronously.

Given a branching process derived for a component PT-net of a CSPT-net, consider an event e such that its corresponding transition is an input (or output) of a channel place q in the CSPT-net. Then the occurrence depth of such event w.r.t. the channel place q is the number of events such that they all causally precede e and their corresponding transitions are also inputs (or outputs) of the channel place q. Hence the tokens flowing through channel places are based on the FIFO (First In, First Out) policy. The occurrence depth intuitively represents the number of tokens which have entered (or left) the channel place q before the occurrence of e.

Definition 6.2 (occurrence depth)**.** *Let* CSPT *be as in Definition 6.1, and* $q \in Q$ *and* PT_i *be such that* ${}^\bullet q^\bullet \cap T_i \neq \varnothing$*. Moreover, let* $\Pi = (O, h)$ *be a branching process of* PT_i*, and* e *be an event of* $O = (P', T', F')$ *such that* $h(e) \in {}^\bullet q^\bullet$*.*

The depth *of* e *in* Π *w.r.t. the channel place* q *is given by:*

$$depth_q^\Pi(e) = |\{f \in T' \mid h(f) \in {}^\bullet q^\bullet \wedge (f, e) \in F'^*\}| \,.$$

Moreover, if the process Π *is clear from the context, we will write* $depth_q(e)$ *instead of* $depth_q^\Pi(e)$*.* ◇

Proposition 6.1. *Let* Π *and* $q \in Q$ *be as in Definition 6.2. Moreover, let* e *and* f *be two distinct events of* Π *satisfying* $\neg(e \# f)$ *and* $h(e), h(f) \in {}^\bullet q^\bullet$*. Then* $depth_q(e) \neq depth_q(f)$*.*

Proof. By $\neg(e \# f)$ and Definition 6.1(2), we have that either eF'^+f or fF'^+e holds. Hence $depth_q(e) < depth_q(f)$ or $depth_q(e) > depth_q(f)$, respectively. □

The nets in the dashed line boxes in Figure 6.3(b) are two component branching processes derived from the component PT-nets of the CSPT-net in Figure 6.3(a). The labels are shown alongside each node, and the occurrence depth of each event

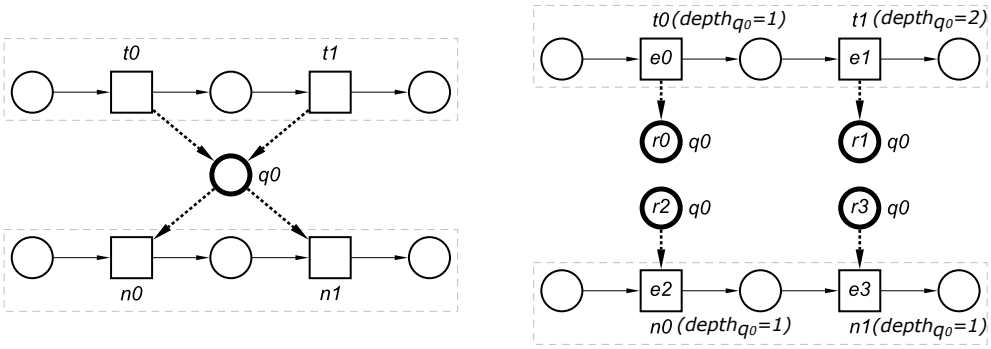


Figure 6.3: (a) A CSPT-net, and (b) its branching process (event labels are shown alongside the nodes and the occurrence depths are shown in brackets).

connected to a (unique, in this case) channel place is shown in brackets. Let us consider event e_1. Its corresponding transition t_1 is the input of channel place q_0. When searching the directed path starting at the initial state and terminating at e_1, we can find another event (viz. e_0) such that its corresponding transition is also the input of q_0. Therefore the occurrence depth of e_1, w.r.t. q_0, is $depth_{q_0}(e_1) = 2$. It intuitively represents transition t_1 passing the second token to the channel.

6.3.2 Non-branching Processes of CSPT-nets

Similarly to the way in which CSPT-nets are extensions of PT-nets, non-branching processes of CSPT-nets are extensions of non-branching occurrence nets.

Definition 6.3 (non-branching process of CSPT-net)**.** *Let* CSPT *be as in Definition 6.1 with M_0 being empty.*

A non-branching process *of* CSPT *is a tuple:*

$$\text{CSO} = (\Pi_1, \ldots, \Pi_k, Q', W', h')$$

such that each $\Pi_i = (O_i, h_i)$ is a non-branching process of PT$_i$ *with $O_i = (P_i', T_i', F_i')$; Q' is a set of channel places; $W' \subseteq (T' \times Q') \cup (Q' \times T')$ where $T' = \bigcup T_i'$; and $h' : Q' \to Q$.*

It is assumed that the following hold, where $h = h' \cup \bigcup h_i$ and $F' = \bigcup F_i'$:

1. *The O_i's and Q' are pairwise disjoint.*

2. *For every $r \in Q'$,*

   - $|{}^\bullet r| = 1$ *and* $|r^\bullet| \leq 1$*; and*
   - *if* $e, f \in {}^\bullet r^\bullet$*, then* $depth_{h(r)}(e) = depth_{h(r)}(f)$*.*

3. *For every $e \in T'$, the restriction of h to ${}^\bullet e \cap Q'$ is a bijection between ${}^\bullet e \cap Q'$ and ${}^\bullet h(e) \cap Q$, and similarly for $e^\bullet \cap Q'$ and $h(e)^\bullet \cap Q$.*

4. *The relation*

$$(\sqsubset \cup \prec)^* \circ \prec \circ (\prec \cup \sqsubset)^*$$

   over T' is irreflexive, where:

   - $e \prec f$ *if there is* $p \in \bigcup P'_i$ *with* $p \in e^\bullet \cap {}^\bullet f$*; and*
   - $e \sqsubset f$ *if there is* $r \in Q'$ *with* $r \in e^\bullet \cap {}^\bullet f$*.*

5. *$h(M_{\text{CSO}}) = M_{\text{CSPT}}$, where M_{CSO} is the default initial state of* CSO *defined as* $\bigcup M_{O_i}$*.* ◇

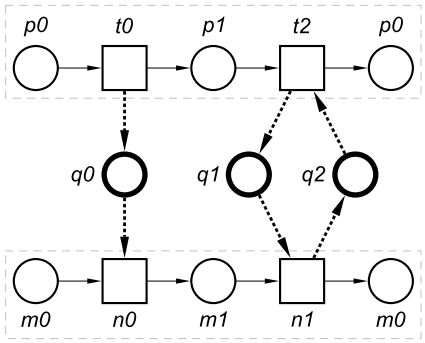


Figure 6.4: A CSO-net which is a possible single run of the CSPT-net of Figure 6.2.

The above definition extends that CSON in Chapter 2 by allowing an infinite number of nodes, and therefore provides a general meaning of a single run of a CSPT-net. In particular, Definition 6.3(2) implies that each channel place has exactly one input event and at most one output event. Moreover, the condition

involving the depth of two events accessing the same channel place means that the tokens flowing through channel places are based on the FIFO policy, so that the size of the subsequent full representation of the behaviours of a CSPT-net is kept low. For example, consider again the CSPT-net in Figure 6.3(a). Using the notion of occurrence depth, we know that n_0 can only consume the token generated from t_0, and n_1 can only consume the token generated from t_1. Hence, there is a unique CSO with $M_{\text{CSO}} = M_{\text{CSPT}}$ and finial state corresponds to the final marking of CSPT. Definition 6.3(3) restricts the mapping h' to a bijection between the channel places in CSPT-net and in CSON. Definition 6.3(4) indicates the acyclicity in CSOs, and Definition 6.3(5) address the consistency of the initial markings in CSPT-net and CSO.

The CSO in Figure 6.4 shows a non-branching processes with the labels (alongside the nodes) coming from the CSPT-net shown in Figure 6.2. It corresponds, e.g. to the step sequence $\lambda = \{t_0\}\{n_0\}\{t_2, n_1\}$ in the original CSPT-net.

6.3.3 Branching Processes of CSPT-nets

We have described two classes of structured nets, i.e. CSPT-nets and CSOs. The former is a system-level class of nets providing representations of entire systems, whereas the latter is a behaviour-level class of nets representing single runs of such systems. In this section, we will introduce a new class of branching nets which can capture the complete behaviours of CSPT-nets.

Definition 6.4 (branching process of CSPT-net)**.** *Let* CSPT *be as in Definition 6.1 with* M_0 *being empty.*

A branching process *of* CSPT *is a tuple:*

$$\text{BCSO} = (\Pi_1, \ldots, \Pi_k, Q', W', h')$$

such that each $\Pi_i = (O_i, h_i)$ *is a branching process of* PT_i *with* $O_i = (P'_i, T'_i, F'_i)$*;* Q' *is a set of channel places;* $W' \subseteq (T' \times Q') \cup (Q' \times T')$ *where* $T' = \bigcup T'_i$*; and* $h' : Q' \to Q$*.*

It is assumed that the following hold, where $h = h' \cup \bigcup h_i$ *and* $F' = \bigcup F'_i$*:*

1. *The* O_i*'s and* Q' *are pairwise disjoint.*

2. *For all* $r, r' \in Q'$ *with* $h(r) = h(r')$*, as well as for all* $e \in {}^\bullet r^\bullet$ *and* $f \in {}^\bullet r'^\bullet$*,*

$$depth_{h(r)}(e) = depth_{h(r')}(f) \iff r = r' \ .$$

3. BCSO *is covered in the graph-theoretic sense by a set of non-branching processes* CSO *of* CSPT *satisfying* $M_{\text{CSO}} = M_{\text{BCSO}}$*, where the default initial state* M_{BCSO} *of* BCSO *is defined as* $\bigcup M_{O_i}$*.* ◇

Two branching processes BCSO′ and BCSO of a CSPT-net are *isomorphic* if there is a bijective homomorphism h from BCSO′ to BCSO such that BCSO′ $\circ\, h =$ BCSO. Using arguments similar to those used in the case of the standard net unfoldings, one can show that there is a unique (up to isomorphism) maximal branching process $\text{BCSO}_{\text{CSPT}}$, called the *unfolding* of CSPT.

A branching process of a CSPT-net consists of branching processes obtained from each component PT-net and a set of channel places. The default initial state M_{BCSO} consists of the initial states in the component branching processes. In addition, Definition 6.4(1) means that the component branching processes are independent, and all the interactions between them must be via the channel places. In particular, there is no direct flow of tokens between any pair of the component branching processes. Definition 6.4(2) implies that the occurrence depths of events inserting tokens to a channel place are the same, and are equal to the occurrence depths of events removing the tokens. Moreover, channel places at the same depth correspond to different channel places in the original CSPT-net. Finally, Definition 6.4(3) specifies that the label of every input and output event of a channel place in BCSO matches a corresponding transition in the original CSPT-net. In general, every node and arc in the branching process belongs to at least one non-branching process of CSPT-net (CSO). This ensures that every event in the BCSO is *executable* from the default initial state M_{BCSO} (i.e. it belongs to a step enabled at some reachable marking), and every condition and channel place is reachable (i.e. it belongs to the initial state or to the post-set of some executable events).

Proposition 6.2 (safeness)**.** *Let* BCSO *be as in Definition 6.4. Then* BCSO *is safe when executed from the default initial state* M_{BCSO}*.*
Note: This means that we treat BCSO *as a* CSPT*-net with the initial marking*

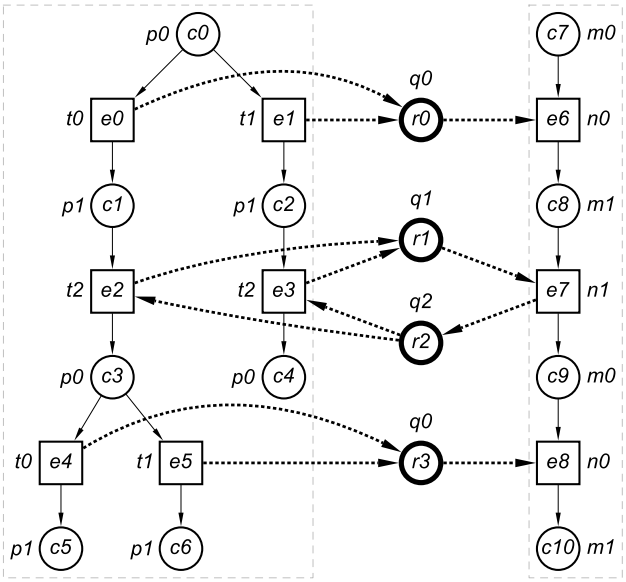


Figure 6.5: A branching process of the CSPT-net of Figure 6.2.

obtained by inserting a single token in each condition belonging to M_{BCSO}*, and safety means that no reachable marking contains more than one token in any condition, including the channel places.*

Proof. For the conditions which are not channel places, this follows from the general properties of the branching processes of PT-nets. For the channel places, this follows from Proposition 6.1 and the fact that no event in a branching process of a PT-net can be executed more than once from the default initial marking. □

The net in Figure 6.5 is one of the branching processes of the CSPT-net showing in Figure 6.2. We can observe that every input and output event of a channel place has the same occurrence depth which represents the token flow sequence during communication between different PT-nets. For instance, the occurrence depths of e_4 and e_8, w.r.t. q_0 are $depth_{q_0}(e_4) = depth_{q_0}(e_8) = 2$. This means of that the transitions t_0 and n_0 were involved in a second asynchronous communication. For the CSPT-net in Figure 6.3(a), its branching process is the nets shown in Figure 6.3(b) with the merge of the channel place pair r_0 and r_2, and the pair r_1

and r_3. So each of the merged channel places is connected with the events with the same occurrence depth, and none of them can be marked more than once.

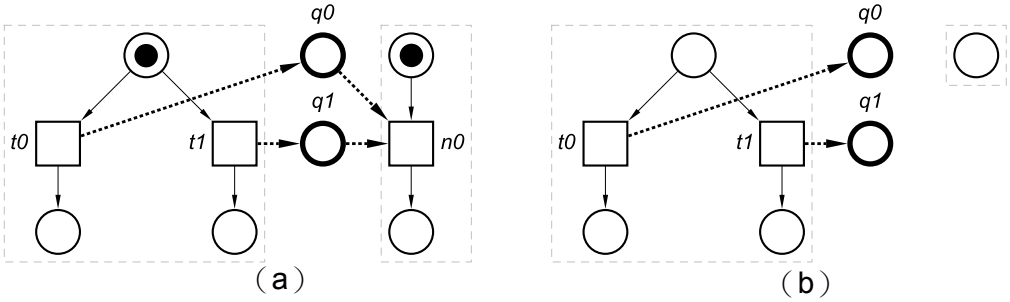


Figure 6.6: A CSPT-net (a), and its branching process (b).

Remark 6.1. *A* BCSO *cannot, in general, be obtained by simply unfolding every component* PT*-net independently and appending the necessary channel places afterwards. Proceeding in such a way can lead to a net violating Definition 6.4(3). This is so because an executable transition in a component* PT*-net does not have to be executable within the context of the* CSPT*-net. For example, Figure 6.6(b) shows a valid branching process of the* CSPT*-net of Figure 6.6(a). One can observe that there is no n_0-labelled event in the* BCSO*, since transition n_0 in* PT_2 *can never be executed since t_0 and t_1 are in conflict, and the system is acyclic.* $\diamond$

6.4 Completeness of Branching Processes

The unfolding of a CSPT-net contains a full reachability information about the original net. It is in general infinite and therefore of limited use for the verification. For bounded CSPT-nets, however, there always exist a truncated part of possibly infinite unfolding that contains full reachability information about the original net.

In this section, we introduce the concept of a complete prefix of the unfolding of a CSPT-net. The idea is to consider global configurations of the unfolding taking into account single runs across different component PT-nets. Then we show that the final states of all the finite global configurations correspond to the reachable markings of original CSPT-net. Using this result, it is possible to consider a finite truncation which is sufficient to represent all reachable markings.

6.4.1 Global Configurations

A global configuration of a BCSO consists of a set of (standard) configurations, each coming from a different component branching process, joined together by channel places.

Definition 6.5 (global configuration)**.** *Let* BCSO *be as in Definition 6.4.*

A global configuration *of* BCSO *is a set of events*

$$\mathbb{C} = \mathbb{C}_1 \cup \cdots \cup \mathbb{C}_k$$

such that each $\mathbb{C}_i$ *is a finite configuration of the process* Π_i*, and the following hold:*

1. ${}^\bullet\mathbb{C} \cap Q' \subseteq \mathbb{C}^\bullet$.
2. *The relation*

$$(\sqsubset \cup \prec)^* \circ \prec \circ (\prec \cup \sqsubset)^*$$

over $\mathbb{C}$ *is irreflexive, where:*

- $e \prec f$ *if there is* $p \in \bigcup P'_i$ *with* $p \in e^\bullet \cap {}^\bullet f$*; and*
- $e \sqsubset f$ *if there is* $r \in Q'$ *with* $r \in e^\bullet \cap {}^\bullet f$.

Moreover we define that $Fin(\mathbb{C}) = (M_{\text{BCSO}} \cup \mathbb{C}^\bullet) \setminus {}^\bullet\mathbb{C}$ *is the* final *state of* $\mathbb{C}$.

The set of all global configurations of BCSO *will be denoted by* $Conf_{\text{BCSO}}$. ◇

Definition 6.5(1) reflects the nature of a/synchronous communication between component (standard) configurations. Intuitively, if we start with an event of the global configuration which is an output event of a channel place, then there exists an input event of the same channel place that also belongs to the global configuration. Moreover, Definition 6.5(2) states that there are no asynchronous cycles in a global configuration.

Proposition 6.3 (configuration is non-branching)**.** *Let* $\mathbb{C}$ *be a configuration as in Definition 6.5. Then, for all distinct* $e, f \in \mathbb{C}$, ${}^\bullet e \cap {}^\bullet f = e^\bullet \cap f^\bullet = \varnothing$.

Proof. Suppose that ${}^\bullet e \cap {}^\bullet f \neq \varnothing$. Then by Definitions 6.4(1) and 6.5 and the definition of a configuration of a branching occurrence net, e, f belong to the same net Π_i and there is $r \in Q'$ such that $r \in {}^\bullet e \cap {}^\bullet f$. This, however, contradicts Proposition 6.1. As a result, ${}^\bullet e \cap {}^\bullet f = \varnothing$. The proof of $e^\bullet \cap f^\bullet = \varnothing$ is similar. □

Proposition 6.4 (configuration is causally closed)**.** *Let $\mathbb{C}$ be a configuration as in Definition 6.5. Then, for every $e \in \mathbb{C}$, $p \in \bigcup P'_i$ and $p \in e^\bullet \cap {}^\bullet f$ imply $f \in \mathbb{C}$. Moreover, if $r \in Q' \cap {}^\bullet e$ then there is $f \in \mathbb{C}$ such that $r \in f^\bullet$.*

Proof. Follows from the definition of a configuration of a PT-net, Proposition 6.1 and Definition 6.5. □

Since in BCSO we use the *merging* technique in the case of channel places (i.e. different events with same occurrence depth and label will link with same instance of channel place), it is possible for a channel place to have multiple inputs or outputs. Propositions 6.3 and 6.4 imply that global configuration are guaranteed to be non-branching and causally closed w.r.t. the flow relations F' and W'. Indeed, if a channel place has more than one input (or output) events, these events are in conflict w.r.t. the flow relation F'. Hence the events belong to different configurations, and each channel place in global configuration has exactly one input and no more than one output. As a result, a global configuration retains key properties of the standard configurations, and it represents a valid execution of transitions of the original CSPT-net.

Consider again the branching process in Figure 6.5. It has a configuration $\mathbb{C} = \{e_0, e_2, e_6, e_7\}$ which consists of two configurations $\mathbb{C}_1 = \{e_0, e_2\}$ and $\mathbb{C}_2 = \{e_6, e_7\}$ coming from two component branch processes, whereas $\mathbb{C}' = \{e_0, e_6, e_7\}$ and $\mathbb{C}'' = \{e_0, e_1\}$ are not valid configurations ($\mathbb{C}'$ has no input event for the channel place r_3, while $\mathbb{C}''$ includes two events in conflict).

Each finite configuration $\mathbb{C}$ has a well-defined final state determined by the outputs of the events in $\mathbb{C}$. Intuitively, such a state comprises the conditions and channel places on the frontier between the events of $\mathbb{C}$ and events outside $\mathbb{C}$. Note that a final state may contain channel places which were involved in asynchronous communications. No channel place involved in a synchronous communications can appear in $Fin(\mathbb{C})$, as such channel place must provide input for another event. For instance, the final state of the global configuration example above is $Fin(\mathbb{C}) =$

$\{c_3, c_7\}$, whereas the final state of another global configuration $\mathbb{C}''' = \{e_0\}$ is $Fin(\mathbb{C}''') = \{r_0, c_1\}$ which contains an asynchronous channel place.

The next result shows that a global configuration together with their outputs and the initial state form a CSO representing a non-branching process of the original CSPT-nets. And, similarly, the events of a non-branching process included in a branching one form a global configuration.

Proposition 6.5. *Let* BCSO *be as in Definition 6.4.*

1. *Let* $\mathbb{C}$ *be a global configuration as in Definition 6.5. Then* $M_{\text{BCSO}} \cup \mathbb{C} \cup \mathbb{C}^\bullet$ *are the nodes of a non-branching process of* CSPT *included in* BCSO.

2. *The events of any non-branching process* CSO *included in* BCSO *and satisfying* $M_{\text{CSO}} = M_{\text{BCSO}}$ *form a global configuration.*

Proof. (1) Let $\mathbb{C} = \mathbb{C}_1 \cup \cdots \cup \mathbb{C}_k$ be as in Definition 6.5. From the standard theory we know that, for each i, $M_{O_i} \cup \mathbb{C}_i \cup \mathbb{C}_i^\bullet$ form the nodes of a non-branching process Π_i' of PT_i included in Π_i and satisfying $M_{\Pi_i'} = M_{\Pi_i}$. Define CSO as composed of $\Pi_1', \ldots, \Pi_k'$, the channel places in $\mathbb{C}^\bullet$ and the connecting arrows. We need to show that CSO satisfies Definition 6.3.
We then observe that: Definition 6.3(2) follows from Proposition 6.3 and Definitions 6.5(1) and 6.4(2); Definition 6.3(3) follows from Definition 6.4(3); Definition 6.3(4) follows from Definition 6.5(2); and Definition 6.3(5) follows from Definition 6.5(2) and $M_{\Pi_i'} = M_{\Pi_i}$.

(2) Follows from Definition 6.3 and an argument reversing that carried out in part (1). □

Proposition 6.6. *Let* $\mathbb{C}$ *be a global configuration as in Definition 6.5. Then* $h(Fin(\mathbb{C}))$ *is a reachable marking in the original* CSPT*-net.*

Proof. Follows from Proposition 6.5(1) and the properties of non-branching processes of CSPT-nets. □

By combining Propositions 6.5 and 6.6, we obtain that global configurations provide a faithful representation of all the reachable marking of the original CSPT-net.

Theorem 6.1. *Let* $\text{BCSO}_{\text{CSPT}}$ *be the unfolding of* CSPT*. Then* M *is a reachable marking of* CSPT *if and only if* $M = h(Fin(\mathbb{C}))$*, for some global configuration* $\mathbb{C}$ *of* $\text{BCSO}_{\text{CSPT}}$*.*

6.4.2 Complete Prefixes of CSPT-nets

A complete prefix of the unfolding of a CSPT-net contains full reachability information about the original net.

The semantical meaning of completeness has been first addressed in McMillan's seminal work in order to avoid the state explosion problem in the verification of systems modelled with Petri nets [35]. The completeness criteria ensures which information is to be preserved in the prefix. Briefly, the unfolding of a net N is *(marking-)complete* if any marking reachable in N is in the image of some cut of the unfolding. The property has been further discussed in [60], which extended it to more general properties. We can adapt the resulting notion to the current context as follows.

Definition 6.6 (completeness)**.** *Let* BCSO *be as in Definition 6.4, and* E_{cut} *be a set of events of* BCSO*. Then* BCSO *is complete w.r.t.* E_{cut} *if the following hold:*

- *for every reachable marking* M *of* CSPT*, there is a finite global configuration* $\mathbb{C}$ *such that* $\mathbb{C} \cap E_{cut} = \varnothing$ *and* $Fin(\mathbb{C}) = M$*; and*
- *for each global configuration* $\mathbb{C}$ *of* $\text{BCSO}_{\text{CSPT}}$ *such that* $\mathbb{C} \cap E_{cut} = \varnothing$ *and, for each event* $e \notin \mathbb{C}$ *of* $\text{BCSO}_{\text{CSPT}}$ *such that* $\mathbb{C} \cup \{e\}$ *is a global configuration of* $\text{BCSO}_{\text{CSPT}}$*,* e *belongs to* BCSO*.*

BCSO *is* marking complete *if it satisfies the first condition.* ◇

Figure 6.7 is a complete unfolding prefix of the CSPT-net in Figure 6.2. $E_{cut} = \{e_1, e_2, e_4\}$ are graphically marked by using double lines.

6.5 Construction of full CSPT-net unfolding

We will now describe an algorithm for constructing the full unfolding of a CSPT-net. A key notion used by the algorithm is that of an executable event (i.e. an

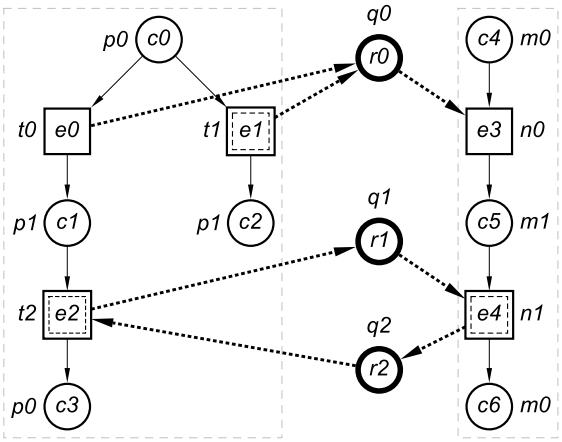


Figure 6.7: A complete prefix of the CSPT in Figure 6.2.

event which is able to fire during some execution from the default initial state) as well as that of an reachable condition or channel place (i.e. one produced by an executable event). Note that whether an event is executable in a CSPT-net is not only determined by the corresponding PT-net, but also by the behaviours of other PT-nets. This means that a component branching process in CSPT unfolding may lose some nodes in the overall unfolding (see Remark 6.1 and Figure 6.6). In other words, there may exist events which are valid extensions in the unfolding process of a component PT-net, but become invalid when considering communication.

In particular, due to synchronous communication, it may be difficult to make sure that every extension is executable before appending it to the unfolding. Unlike the standard unfolding methods, an algorithm for CSPT-net cannot simply unfold the component branching processes adding one event at a time, and connecting it to already existing channel places. This is because a synchronous communication in CSPT unfolding forms a cycle. It is therefore impossible to add only one of the synchronised events and guarantee its executability at the same time. Similarly, adding a synchronous event set together with all related channel places in one step may also be difficult to achieve since the use of merging may produce infinitely many events which are connected to the same channel place.

Instead, our idea is to design an algorithm which will sometimes generate non-executable events requiring tokens from channel places which have not yet been

generated, in the anticipation that later on a suitable (possibly synchronous) events will provide such tokens. Basically, the algorithm maintains two data structures *Unf* and *auxUnf*. The former is used to keep the resulting unfolding, whereas the latter keeps an 'over-approximating unfolding' which may contains non-executable events. Each possible extension together with its output conditions are firstly appended to *auxUnf*. Then the algorithm performs an executability check for the new event after constructing its a/synchronous communications. Only executable events and their related conditions/channel places can be added to *Unf*.

Before providing the details of the algorithm, we introduce some auxiliary notions. In what follows, we assume that CSPT is as in Definition 6.1.

Definition 6.7 (local CSPT configuration)**.** *Let $e \in \mathbb{C}$, where $\mathbb{C}$ is a global configuration of* BCSO *as in Definition 6.5. Then the local CSPT configuration of e in $\mathbb{C}$, denoted by $\mathbb{C}[e]$, is defined as*

$$\mathbb{C}[e] = \{f \in C \mid (f, e) \in (\prec \cup \sqsubset)^*\} ,$$

where the relations $\prec$ and $\sqsubset$ are as in Definition 6.5. Moreover,

$$\mathit{Conf}(e) = \{\mathbb{C}[e] \mid \mathbb{C} \in \mathit{Conf}_{\mathit{BCSO}} \wedge e \in \mathbb{C}\}$$

is the set of all CSPT local configurations of e. ◇

The CSPT local configuration of an event e in $\mathbb{C}$ is the set of events that are executed before (or together with) e. In general, it consists of a configuration comprising the standard local configuration of e together with a set of standard configurations coming from other branching processes. Note that an event may have different local CSPT configurations, e.g. if one of its inputs is a channel place which has multiple input events. Each such local configuration belongs to a different non-branching process. For instance, consider a global configuration $\mathbb{C} = \{e_0, e_3, e_4\}$ in Figure 6.8(b). The CSPT local configuration of event e_4 in $\mathbb{C}$ is $\mathbb{C}[e_4] = \{e_4, e_3, e_0\}$ (which in this case is identical to $\mathbb{C}$). It involves two standard local configurations, $[e_3]$ and $[e_4]$. Moreover, we can observe that $\mathbb{C}[e_4]$ is not the unique local configuration of e_4, as another one is $\mathbb{C}'[e_4] = \{e_1, e_4\}$.

An event may even have infinitely many local configurations. If we continue to unfold the net, we will construct infinitely many n_0 and t_1 labelled events with

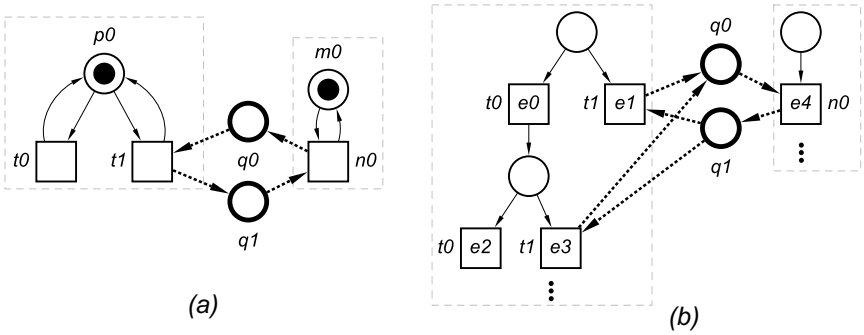


Figure 6.8: (a)A CSPT-net; and (b) its unfolding.

occurrence depths equal to 1 (w.r.t. q_0 and q_1). All of them are synchronised with e_4 via the same channel places. As the result, e_4 would have infinitely many local configurations belonging to different runs.

6.5.1 Computing Local Configuration

In order to improve the efficiency of unfolding procedure, checking for the existence of a local CSPT configuration of an event can be reduced to the problem of exploring the causal dependencies between channel places.

Below we assume that if $\mathbb{C}_i$ is a configuration of the unfolding of the i-th component PT-net, and $e \in \mathbb{C}_i$ and $q \in Q$ are such that $(h(e), q) \in W$ (or $(q, h(e)) \in W$), then $r = (q, \mathit{depth}_q(e))$ belongs to the set of implicit channel places $Q_{\mathbb{C}_i}$ connected to $\mathbb{C}_i$. Moreover, the label of r is q, and $(e, r) \in W_{\mathbb{C}_i}$ (resp. $(r, e) \in W_{\mathbb{C}_i}$) is the corresponding implicit arc.

Definition 6.8 (a/sync graph)**.** *Let $\mathbb{C}_i$ be a configuration of the unfolding of the i-th component* PT*-net.*

Then the a/sync graph *of $\mathbb{C}_i$ is defined as:*

$$\mathcal{G}(\mathbb{C}_i) = (Q_{\mathbb{C}_i}, \widehat{\prec}_{\mathbb{C}_i}, \widehat{\sqsubset}_{\mathbb{C}_i})$$

where $\widehat{\prec}_{\mathbb{C}_i}, \widehat{\sqsubset}_{\mathbb{C}_i}$ are two binary relations over $Q_{\mathbb{C}_i}$ such that, for every $r, r' \in Q_{\mathbb{C}_i}$:

- *$r \;\widehat{\prec}_{\mathbb{C}_i}\; r'$ if there are two distinct $e, f \in \mathbb{C}_i$ such that $(r, e), (f, r') \in W_{\mathbb{C}_i}$, and e precedes f within $\mathbb{C}$; and*
- *$r \;\widehat{\sqsubset}_{\mathbb{C}_i}\; r'$ if there is $e \in \mathbb{C}_i$ with $(r, e), (e, r') \in W_{\mathbb{C}_i}$.* ◇

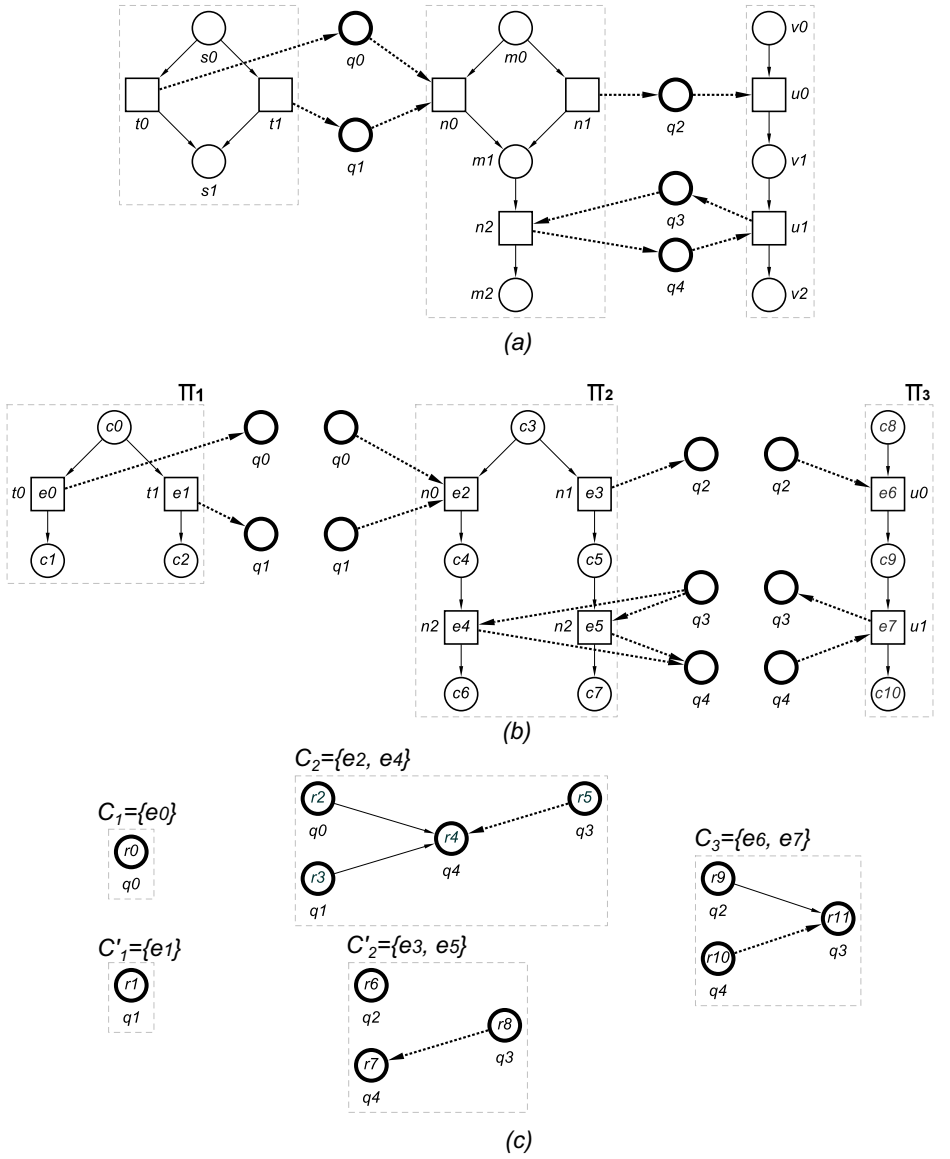


Figure 6.9: (a) a CSPT-net, (b) unfoldings of three component PT-nets of (a) (together with their implicit channel places), and (b) a/sync graphs of configurations derived from these unfoldings.

$\mathcal{G}(\mathbb{C}_i)$ captures relationships between the input and output channel places of a configuration of the unfolding of an individual component system. Its nodes are the channel places involved in $\mathbb{C}_i$. Moreover, $r \mathrel{\widehat{\prec}}_{\mathbb{C}_i} r'$ if there is a path from r to r' involving more than one event of $\mathbb{C}_i$, and $r \mathrel{\widehat{\sqsubset}}_{\mathbb{C}_i} r'$ if r is an input and r'

an output of some event in $\mathbb{C}_i$.

Figure 6.9(b) shows the unfolding of each component PT-net of (a) together with their input and output channel places. By exploring the relations between those channel places, we are able to generate a/sync graph for any configuration. For example, Figure 6.9(c) shows five a/sync graphs of the configurations derived from (b), where the relations $\widehat{\prec}_{\mathbb{C}_i}$ and $\widehat{\sqsubset}_{\mathbb{C}_i}$ are represented by solid arcs and thick dashed arcs, respectively. For the left-hand side PT-net Π_1, we have that:

$$\mathcal{G}(\mathbb{C}_1) = (\{r_0\}, \varnothing, \varnothing) \qquad \mathcal{G}(\mathbb{C}'_1) = (\{r_1\}, \varnothing, \varnothing)$$

The a/sync graphs of the configurations in Π_2 are:

$$\mathcal{G}(\mathbb{C}_2) = (\{r_2, r_3, r_4, r_5\}, \{(r_2, r_4), (r_3, r_4)\}, \{(r_5, r_4)\})$$
$$\mathcal{G}(\mathbb{C}'_2) = (\{r_6, r_7, r_8\}, \varnothing, \{(r_8, r_7)\}$$

and for the right-hand side PT-net Π_3, we have that:

$$\mathcal{G}(\mathbb{C}_3) = (\{r_9, r_{10}, r_{11}\}, \{(r_9, r_{11})\}, \{(r_{10}, r_{11})\})$$

Given a set of a/sync graphs $\mathcal{G}(\mathbb{C}_1), \ldots, \mathcal{G}(\mathbb{C}_k)$ extracted for the k component systems, we call these graphs compatible if all inputs are produced and there is no cycle involving $\widehat{\prec}$.

Definition 6.9 (compatibility of a/sync graphs)**.** *Let $\mathbb{C}_i$ $(i = 1, \ldots, k)$ be a configuration of the unfolding of the i-th component* PT*-net, and $\mathcal{G}(\mathbb{C}_i) = (Q_{\mathbb{C}_i}, \widehat{\prec}_{\mathbb{C}_i}, \widehat{\sqsubset}_{\mathbb{C}_i})$.*

Then $\mathbb{C}_1, \ldots, \mathbb{C}_k$ are compatible *configurations if the following hold:*

1. *if $(r, e) \in W_{\mathbb{C}_i}$ then that there is $j \neq i$ such that $r \in Q_{\mathbb{C}_j}$; and*
2. *the relation*

$$(\widehat{\sqsubset} \cup \widehat{\prec})^* \circ \widehat{\prec} \circ (\widehat{\prec} \cup \widehat{\sqsubset})^*$$

is irreflexive, where $\widehat{\prec} = \bigcup \widehat{\prec}_{\mathbb{C}_i}$ and $\widehat{\sqsubset} = \bigcup \widehat{\sqsubset}_{\mathbb{C}_i}$. ◇

In Figure 6.9, configurations $\mathbb{C}_1, \mathbb{C}'_2, \mathbb{C}_3$ are compatible since the q_3-labelled input channel place r_8 in $\mathcal{G}(\mathbb{C}'_2)$ is present in $\mathcal{G}(\mathbb{C}_3)$ (i.e. r_{11}), and the input channel places r_9, r_{10} (labelled by q_2 and q_4 respectively) in $\mathcal{G}(\mathbb{C}_3)$ are all present in $\mathcal{G}(\mathbb{C}'_2)$. On the other hand, we can observe that there are no compatible configurations which involve $\mathbb{C}_2$, i.e. neither configurations $\mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_3$ nor $\mathbb{C}'_1, \mathbb{C}_2, \mathbb{C}_3$ are compatible. This is because the producers of r_2 and r_3 are in conflict in Π_1.

Theorem 6.2. *Let $\mathbb{C}_1, \ldots, \mathbb{C}_k$ be configurations of the unfoldings of the component* PT*-nets, and $\mathbb{C} = \mathbb{C}_1 \cup \cdots \cup \mathbb{C}_k$. Then $\mathbb{C}$ is a global configuration if and only if $\mathbb{C}_1, \ldots, \mathbb{C}_k$ are compatible.*

Proof. ($\Longrightarrow$) If $\mathbb{C}$ is a global configuration then, Proposition 6.4, every input channel place of a global configuration $\mathbb{C}$ is produced in $\mathbb{C}$, and from Definition 6.1(2), the producer e and consumer f belong to separate configurations. Hence Definition 6.9(1) holds. Moreover, Definition 6.9(2) follows from Definition 6.5(2).

($\Longleftarrow$) We observe that Definition 6.5(1) and Definition 6.5(2) respectively follow from Definition 6.9(1) and Definition 6.9(2).

□

Therefore, one can obtain the CSPT local configurations of an event e by checking whether there are compatible configurations $\mathbb{C}_1, \ldots, \mathbb{C}_k$ such that e belongs to one of them. Such a task can be made efficient by working with the graphs $\mathcal{G}(\mathbb{C}_1), \ldots, \mathcal{G}(\mathbb{C}_k)$. In fact, one can just check those configurations which have dependencies on e.

6.5.2 Unfolding Algorithm

The unfolding algorithm we are going to present significantly differs from the existing net unfolding algorithms. The key difference is that during the unfolding procedure we will be constructing nodes and connections which will not necessarily be the part of the final unfolding. This is due to the presence of synchronous communication within our model. More precisely, in the net being constructed there will be *executable* and *non-executable* events and conditions. The former will definitely be included in the resulting unfolding, whereas the latter cannot be yet included due to the absence of event(s) which are needed for communication. If, at some later stage, the missing events are generated, then the previously non-executable event and the conditions (and channel places) it produced become executable. In particular, we will call an event e *executable* if $Conf(e) \neq \varnothing$. This happens if we have generated enough events to find at least one local CSPT configuration of e in the unfolding.

The net generated by the algorithm may not strictly speaking be a branching process during its creation, therefore we do not put the non-executable events

into *Unf* immediately, but keep them in an auxiliary data structure *auxUnf*, and transfer them to *Unf* once they become executable. In this way, we make sure *Unf* converges to the unfolding.

Intuitively, an executable event is an event belonging to at least one run of a BCSO. For the example net in Figure 6.9(b), if we combine the channel places with the same label together, then e_6 is an executable event since there exists a local CSPT configuration of e_6: $\mathbb{C}[e_6] = \{e_0, e_3, e_6\}$, where $\mathbb{C} = \{e_0, e_3, e_6\}$. On the other hand, event e_2 is non-executable because it does not have any local configuration (we have seen the example of Figure 6.9(c) that there are no compatible configurations which involve e_2). Therefore, such a net is not a valid CSPT branching process since according to Definition 6.4(3) every event in BCSO is executable. If we remove e_2 together with its successors, then all events in the new net become executable indicating the net is a valid BCSO.

Proposition 6.7. *Let e be an executable event in* BCSO. *Then each event appearing in any configuration in $Conf(e)$ is executable.*

Proof. From Definition 6.7 it follows that for each event f in any configuration in $Conf(e)$ there exist a global configuration $\mathbb{C}$ such that $e, f \in \mathbb{C}$. Hence $Conf(f) \neq \varnothing$, and so the result follows from the definition of executable events. □

The procedure for constructing the unfolding of a CSPT-net is presented as Algorithm 7.

The first part of the algorithm adds conditions representing the initial marking of the CSPT-net being unfolded. It also adds possible extensions to the working set *pe*. A *possible extension* of *Unf* is a pair $e = (t, B)$ with $h(e) = t$ where t is a transition of CSPT, and B is a set of conditions of *Unf* such that:

- B is a co-set in one of the subnets of *Unf*
- $h(B)$ are all the input non-channel places of t; and
- $(t, B) \notin pe$ and *Unf* contains no t-labelled event with the non-channel place inputs B.

The pair (t, B) is an event used to extend BCSO without considering channel places. We use the standard condition of a possible extension to choose events

Algorithm 7 (unfolding of CSPT-net)

input: CSPT — CSPT-net
output: *Unf* — unfolding of BCSO

Unf ← the empty branching process
auxUnf ← the empty branching process // auxiliary data structure to keep branching process with non-executed events
pe ← ∅
add instances of the places in the initial marking of CSPT to *Unf* and *auxUnf*
add all possible extensions of *Unf* to *pe*

while $pe \neq \varnothing$ **do**
  remove e from *pe*
  $addConnections(e)$
  if $Conf(e) \neq \varnothing$ **then**
    for all events f in configurations of $Conf(e)$ **do**
      if f is not present in *Unf* **then**
        add f and its related places and arcs created in $addConnections(e)$ to *Unf*
  add all possible extensions of *Unf* to *pe*

procedure addConnections (**input:** $e = (t, B)$)
add e to *auxUnf*
create and add all the standard post-conditions of e to *auxUnf*
for all channel place $q \in {}^\bullet t^\bullet$ **do**
  let $r := (q, k)$ where $k := depth_q(e)$
  if there is no $r := (q, k)$ in *auxUnf* **then**
    add q-labelled channel place r to *auxUnf*
  add a corresponding arc between r and e

that can be added to a component branching process (i.e. $h(B) = {}^\bullet t \cap P'$), while constructing the related a/synchronous communications in a separate step. In such a way, the complexity of appending groups of synchronous events is significantly reduced. Note that a possible extension e has precisely determined channel place connections since the depth values are fully determined.

Procedure *addConnections* provides the details of appending a possible extension e to BCSO as well as constructing related channel place structure after removing e from *pe*. Each new extension and its output conditions are treated

as non-executable and kept in *auxUnf*. The conditions in *auxUnf* also indicate that they are unable to be used for deciding any further possible extension. In this way we can avoid any unnecessary extension and make sure the predecessors of every new event is executable.

The procedure then creates the a/synchronous communications of the input event if it is required. Given an event e, for every input or output channel place q of its corresponding transition $h(e)$ in the original CSPT-net, we search in *auxUnf* for the matching channel place (i.e. its label is q and its depth value equals to the occurrence depth of e). Then we create a direct connection if such a channel place exists. Otherwise, we add a new instance of the channel place together with the corresponding arc.

After adding the implicit channel places connected to e (or creating the connection for those which already existed) together with the corresponding arcs, we are able to obtain the local configuration of e by looking for compatible configurations $\mathbb{C}_1, \ldots, \mathbb{C}_k$ of the component nets (which may contain non-executable events) such that e belongs to one of the $\mathbb{C}_i$'s. If e is executable ($Conf(e) \neq \varnothing$), we make all non-executable events in $Conf(e)$ together with their post-conditions executable (see Proposition 6.7) by adding them to *Unf* which always present a valid branching process. After that, we generate new potential extensions (each such extension must use at least one of conditions which have just been made executable). Then another waiting potential extension (if any) is processed.

We now use the CSPT-net in Figure 6.9(a) as an example to illustrate the algorithm. For simplicity, we only show the net generation in *auxUnf*, and omit the step that transfer the nodes from *auxUnf* to *Unf*. The unfolding process is shown in Figure 6.10. It starts by appending instances of the initial marking s_0, m_0 and v_0 of the CSPT-net to *auxUnf*. Then we go to the first iteration which adds a possible extension $e_0 = (t_0, \{c_0\})$ together with its post-condition to *auxUnf*. It is not possible to confirm the executability of e_0 at this point due to the absence of a suitable channel place: to generate related communications, we search in the *auxUnf* for a q_0-labelled and $depth_{q_0}(e_0) = 1$ channel place. Apparently, there is no such place, and so we need to add a copy of q_0 to the net and connect it with e_0. After executing $addConnection(e_0)$ procedure, we find that there is a local configuration of e_0 (which is $\{e_0\}$ itself) by building a/sync

graph and verifying compatibility. This local configuration indicates $h(e_0)$ is executable in one of the single runs of the original CSPT-net, and hence is also an executable event in the corresponding unfolding. Therefore, we are allowed to add e_0 and its output condition c_1 to Unf; c_1 then can be used to generate further extensions after our first complete iteration.

Stage C shows the result after three iterations (i.e. appending e_0, e_1 and e_2). Notice that in the $AddConnection(e_2)$ procedure, it is possible to add only arcs from existing channel places r_0 and r_1 to the chosen event e_2. This is so because suitable channel places were already added in the previous iterations. Moreover, e_2 and c_4 remain to be non-executable after their own iteration since there is no local configuration containing e_2 (nodes treated as non-executable are shaded). Such a condition cannot be used for generating new possible extension until it becomes executable.

Stage D and E illustrate the way to generate synchronous communication between e_4 and e_6. After adding e_4 to the $auxUnf$, we first create 'half' of the synchronous communication, i.e. we add channel places r_3, r_4 and the relations (r_3, e_4), (e_4, r_4). It can be observed that there is no compatible configurations involving e_4 at the moment since the producer of r_3 is missing. Therefore, e_4 is a non-executable event. However, this non-executable event and its post-condition can become executable after adding another 'half' of the synchronous communication i.e. event e_6 and the related connections. More precisely, after $AddConnection(e_6)$, we have the non-executable nodes $\{e_2, c_4, e_4, c_6, e_6, c_9\}$. We can then find compatible configurations $\mathbb{C}_1 = \{e_1\}, \mathbb{C}_2 = \{e_3, e_4\}, \mathbb{C}_3 = \{e_5, e_6\}$ which contain both non-executable events e_4 and e_6. These two events and their output conditions can then be added to Unf.

After adding all possible extensions from pe to $auxUnf$ (in this example, the unfolding is finite since the original CSPT-net is acyclic), in the corresponding Unf, we obtain the correct unfolding which does not have the nodes e_2 and c_4 since they are still non-executable.

6.6 Conclusion

In this chapter, we introduced the concept of a branching process of a CSPT-net — a new class of branching nets which can capture complete behaviours of CSPT-nets. The model extends the standard branching processes by combining multiple component branching processes with channel places. We then formulated the property of completeness for BCSOs. The concept relies on the notion of a global configuration which describes a single run of transitions crossing different PT-nets.

Our investigation has led to an algorithm for constructing the unfolding of a CSPT-net which is its unique maximal BCSO. A central part of the unfolding algorithm is the test of whether a non-executable event can become executable. This is done by checking whether there are global configurations such that the event belongs to one of them. Moreover, only conditions marked as executable can be used for constructing new possible extensions, so that we always generate further extensions on the basis of executable elements.

The algorithm presented in this paper is based on standard unfolding method, which essentially works by appending possible extension one by one. A potentially very efficient approach to the construction of the unfolding could be to use the parallel unfolding technique [31]. One can, for example, unfold each component branching process in parallel, by temporarily ignoring any a/synchronous issues. The procedures of appending channel places as well as executability checking (removing unnecessary events) would proceed in a separate step. In this way, we might significantly improve the efficiency of the algorithm since different component net unfoldings can be constructed on multiple computer processors.

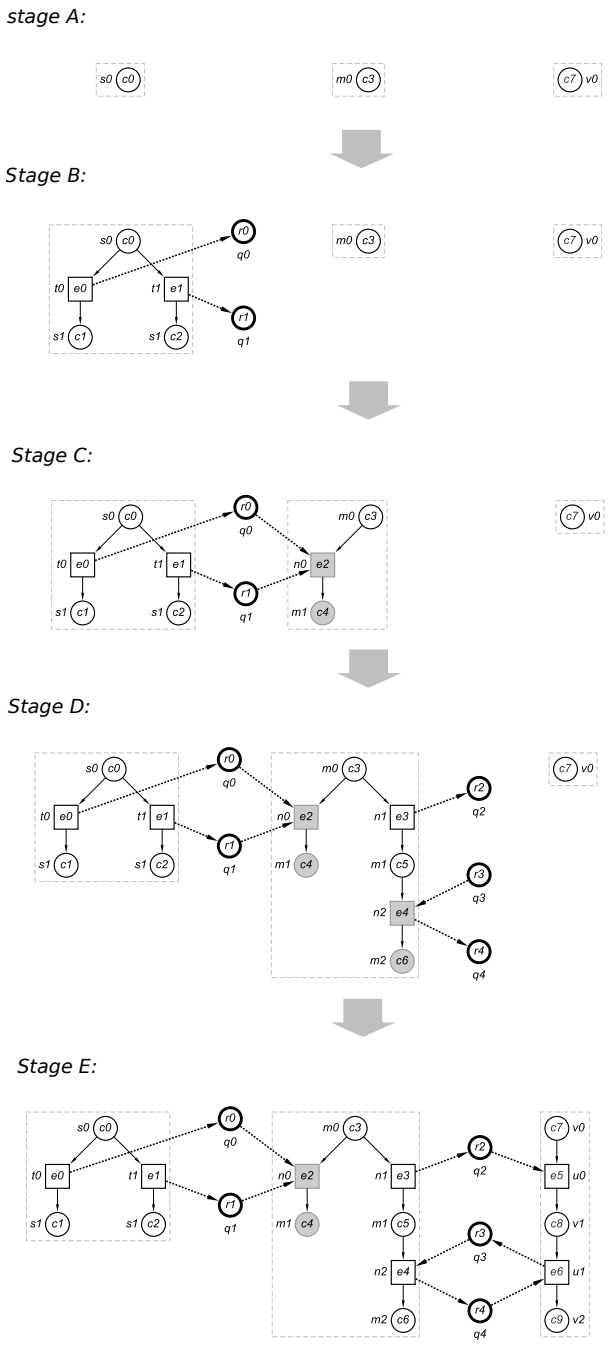


Figure 6.10: Unfolding the CSPT-net of Figure 6.9(a): (Stage A) starting point of the unfolding, (Stage B) after first complete iteration, (Stage C) chosen event e_2 is non-executable, (Stage D) chosen event e_4 is non-executable, (Stage E) e_4 becomes executable due to the missing event is generated.

Chapter 7

Conclusion and Future work

7.1 Conclusion

In this thesis, we have presented a framework for modelling and analysing the behaviour of complex evolving systems. In Chapter 2, the basic concept of structured occurrence nets was discussed. The concept was built on occurrence nets, and uses various relationships to provide a means of recording the activities of interacting and evolving systems. For each variant of SONs, we defined firing rules for step by step simulation, and showed several of the variant's properties for model analysis and verification. We discussed automated verification in SONs, and provided the relevant algorithms. In addition, a case study using SONs for modelling an accident scenario was presented.

In Chapter 3, we proposed an extension of SONs which supports the modelling of alternative behaviours of complex evolving systems. We defined the ASON firing rules used for the simulation. The rules take into account the execution semantics of alternative behaviours.

Chapter 4 introduced time information in SON-based models. The basic idea is to represent time bounds and duration ranges in each node. This given timing information can be uncertain, using a form of interval. We discussed how time information as is provided in a SON can be checked for consistency, and used in time estimation.

Chapter 5 reported on the implementation details of SONs and time-SON visu-

alisations, simulations and analysis. The toolkit named SONCRAFT is a plug-in within the Workcraft framework. SONCRAFT provides a powerful graphical user interface that facilitates SON-based models visualisation, editing and simulation. The verification and analysis algorithms introduced in Chapters 2 and 4 have been implemented as a set of external tools which are integrated with SONCRAFT.

Chapter 6 addressed the unfolding of CSON generator nets (CSPT-nets). The unfolding extends the standard branching processes. The algorithms we proposed for the construction of CSPT-unfolding uses the idea of non-executable event to cope with the synchronous cyclic structure formed by events and channel places.

7.2 Future Work

The following are some ideas for future research:

- The SONs concept has been used as a suitable formal grounding to express system behaviour in several practical areas, including the accident scenario introduced in Chapter 2, and the modelling of data provenance [47]. We also intend to evaluate the potential impact of SON-based methods for post-hoc system analysis, e.g., support for complex crime analysis. In particular, we will use ideas from [32] to design and implement a prototype for displaying SONs representing facts from a complex investigation's database together with means of annotating them with probabilities, as well as facilities for basic analyses and predictions.

- The ASONs discussed in Chapter 3 are used to construct SONs from incomplete and/or contradictory information. An interesting and practically important property of the ASON concept and its application is support for associating probability estimates. Such an idea has been addressed in, for example, [7, 41]. Briefly, we will annotate particular events, conditions, and relations with some form of probability estimates, which indicate the current degree of accuracy of their representation. This probability information can guide the analysis of the likelihood of different scenarios. Interfacing with a probabilistic analyser such as [42] is a possibility.

- In time SONs associating time information with SON execution semantics is left for future research. The enableness of an event is not only determined by marked conditions, but also by the consistency of the time information provided by the event and its input conditions. Furthermore, we hope to use timed SONs to model and analyse cyber crime, problems involving 'big data', neurological processes, dynamic reconfiguration of real-time software, and hardware design,in order to demonstrate its generality and exploit the full potential of the formalism.

- The ongoing implementation work includes the representation of Alternative SONs and the extension of existing verification and simulation facilities in SONCraft for such SONs. For example, several new algorithms have been added to the existing structural analysis tool for the verification of ASONs.

  To date, SONCraft has been assessed and experimented with using a variety of manually-entered artificial trial models, but we have yet to try to integrate the system with an existing large-scale database management system (DBMS)-based investigation support system, or with any automated activity monitoring systems (e.g. for network or telephone communications). Such integration will, we expect, provide the most practical means of assessing the effectiveness of the system's facilities for structuring and hence coping with realistic large and complex evolving models.

- We intend to explore the generation of finite complete prefixes of CSPT-nets, and use them for efficient model checking and system synthesis, which are based on standard Petri nets methods [30, 55]. We also intend to implement the CSPT model and its analysis tools in SONCraft.

Appendix A

In the appendix we provide algorithms for basic and casual time consistency checking which encode the equations and conditions presented in Section 4.3. To make the algorithms easy to read, we will use $n.start$ to indicate the start time of a node n (I_s^n); $n.finish$ to indicate the finish time of n (I_f^n); and $n.duration$ to indicate the duration of n (I_d^n).

.1 Algorithms for Basic Consistency Checking

nodeConsistency is the basic function that implements Conditions (4.6), (4.7), and (4.8) to verify the consistency of a given node with specified start, finish, and duration intervals. The structure of the function is given by Algorithm 8. 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 8 (Node consistency)

1: **function** Boolean nodeConsistency(Node n)
2: $\tilde{I}_f^n := n.start + n.duration$ // Equation (4.26)
3: **if** $\tilde{I}_f^n \cap n.finish = \varnothing$ **then** // Condition (4.6)
4: **return** FALSE
5: $\tilde{I}_s^n := n.finish - n.duration$ // Equation (4.25)
6: **if** $\tilde{I}_s^n \cap n.start = \varnothing$ **then** // Condition (4.7)
7: **return** FALSE
8: $\tilde{I}_d^n := n.finish - n.start$ // Equation (4.27)
9: **if** $\tilde{I}_d^n \cap n.duration = \varnothing$ **then** // Condition (4.8)
10: **return** FALSE
11: **return** TRUE

The *concurConsistency* function implements Conditions (4.9), (4.10), and (4.11), 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 9. 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 9 (Concurrent consistency)

1: **function** Boolean concurConsistency(Event e)
2: **for** $c \in {}^\bullet e$ **do**
3: **if** $c.finish \neq e.start$ **then** // Condition (4.9)
4: **return** FALSE
5: **for** $c \in e^\bullet$ **do**
6: **if** $c.start \neq e.finish$ **then** // Condition (4.10)
7: **return** FALSE
8: **return** nodeConsistency(e) // Condition (4.11)

alterConsistency is called when one attempts to verify the consistency of a state. The algorithm is presented in Algorithm 10. Lines 3-8 and lines 10-15

implement Equations 4.12 and 4.13 respectively, which aim to find out at least one scenario the checked node belongs to.

Algorithm 10 (Alternative consistency)

```
1: function Boolean alterConsistency(State c)
2: b := false
3: for e_2 ∈ •c do
4:    if e_2.finish = c.start then                    // Equation 4.12
5:       b := true
6:       break
7: if ¬b then
8:    return FALSE
9: b := false
10: for e_1 ∈ c• do
11:    if e_1.start = c.finish then                   // Equation 4.13
12:       b := true
13:       break
14: if ¬b then
15:    return FALSE
16: return nodeConsistency(c)                        // Equation 4.14
```

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 11. The function applies Conditions (4.15) and (4.16) for the asynchronous and synchronous-based checking, then invokes *nodeConsistency* for the basic consistency checking of the channel place itself.

Algorithm 11 (A/Synchronous consistency)

```
1: function Boolean asynConsistency(Channel place q)
2: if q.input.finish ≠ q.start ∨ q.output.start ≠ q.finish then
   // Cnds. (4.15), (4.16)
3:    return FALSE
4: return nodeConsistency(q)                         // Condition (4.17)
```

The *bhvConsistency* function is used to verify the time consistency of a BSON. The structure of the function is given by Algorithm 12. The function first verifies the consistency of all binary relations in *causalU* using Condition (4.18), then uses Conditions (4.19) and (4.20) 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 12 (Behavioural consistency)

```
1: function bhvConsistency(Relation causalU)
2: Result := ∅                                   // behaviourally inconsistent nodes
3: for (e1, e2) ∈ causalU do
4:    if e1.start.lower > e2.start.lower ∨
         e1.start.upper > e2.start.upper then     // Condition (4.18)
5:       add e1, e2 to Result
6: for c ∈ C do
7:    if •c = ∅ ∧ c.start ≠ β(c).start then     // Condition (4.19)
8:       add c to Result
9:    else if c• = ∅ ∧ c.finish ≠ β(c).finish then   // Condition (4.20)
10:      add c to Result
11: return Result
```

Algorithm 13 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 13 (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_d^{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 bhvConsistency(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
```

References

[1] *JRE downloads*. http://www.oracle.com/technetwork/java/javase/downloads/. 84

[2] *SONCraft homepage*. https://soncraft.codeplex.com/. 84

[3] *SONCraft tutorial*. http://www.workcraft.org/tutorial/modelling/son/start. 85

[4] *Workcraft homepage*. http://workcraft.org. 73

[5] A.Aghasaryan, E.Fabre, A.Benveniste, R.Boubour, and C.Jard. Fault detection and diagnosis in distributed systems: an approach by partially stochastic Petri nets. *Discrete Event Dynamic Systems*, **8**[2]:203–231, 1998. 3

[6] A.Algirdas, L.Jean-Claude, R.Brian, and L.Carl. Basic concepts and taxonomy of dependable and secure computing. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, **1**[1]:11–33, 2004. 3

[7] A.Bouillard, S.Haar, and S.Rosario. Critical paths in the partial order unfolding of a stochastic Petri net. In JoÃńl Ouaknine and FritsW. Vaandrager, editors, *Formal modeling and analysis of timed systems*, **5813** of *Lecture Notes in Computer Science*, pages 43–57. Springer Berlin Heidelberg, 2009. 118

[8] A.Cheng, J.Esparza, and J.Palsberg. Complexity results for 1-safe nets. In R. K. Shyamasundar, editor, *FSTTCS*, **761** of *Lecture Notes in Computer Science*, pages 326–337. Springer, 1993. 32

[9] A.Mokhov and A.Yakovlev. Conditional partial order graphs: model, synthesis, and application. *Computers, IEEE Transactions on*, **59**[11]:1480–1493, Nov 2010. 73

[10] A.Valmari. *Stubborn sets for reduced state space generation*, pages 491–515. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. 87

[11] A.Yakovlev, L.Lavagno, and A.Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '92, pages 104–111, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 73

[12] B.Li and B.Randell. Soncraft user manual. Technical Report CS-TR-1448, School of Computing Science, Newcastle University, February 2015. 84

[13] B.Li, M.Koutny, and B.Randell. A tool for construction, simulation and verification of SONs. to appear, 2015. 22

[14] B.Randell. Occurrence nets then and now: the path to structured occurrence nets. In *Applications and Theory of Petri Nets*, pages 1–16. Springer Berlin Heidelberg, June 2011. 3

[15] B.Randell. Incremental construction of structured occurrence nets. to appear, 2013. 9, 76

[16] B.Randell and M.Koutny. Failure: their definition, modelling and analysis. In *Theoretical Aspects of Computing–ICTAC 2007*, pages 260–274. Springer, September 2007. 3

[17] C.Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, Cambridge, MA, USA, 1974. 54

[18] E.Best and B.Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, **16**[1]:93–124, 1981. 27, 29

[19] E.Best and C.Fernández. *Nonsequential processes: a Petri net view*, **13** of *EATCS monographs on Theoretical Computer Science*. 2, 9

[20] E.Best and R.Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, **55**[1]:87 – 136, 1987. 2

[21] E.M.Clarke., O.Grumberg, and D.A.Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. 86

[22] I.Poliakov. *Interpreted graph models*. PhD thesis, School of Electrical, Electronic and Computer Engineering, Newcastle University, 2011. 5, 72

[23] I.Poliakov, V.Khomenko, and A.Yakovlev. Workcraft–a framework for interpreted graph models. In *Applications and Theory of Petri Nets*, pages 333–342. Springer Berlin Heidelberg, June 2009. 5, 72

[24] J.Engelfriet. Branching processes of Petri nets. *Acta Informatica*, **28**[6]:575–591, 1991. 39, 87, 89, 91

[25] J.Esparza and J.Heljanko. *Unfoldings: a partial-order approach to model checking*. Springer Publishing Company, Incorporated, 1 edition, 2008. 3

[26] J.Esparza, S.Römer, and W.Vogler. An improvement of McMillan's unfolding algorithm. In *Formal Methods in System Design*, pages 87–106. Springer-Verlag, 1996. 87

[27] J.F.Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, **26**[11]:832–843, 1983. 71

[28] J.Kleijn and M.Koutny. Causality in structured occurrence nets. In *Dependable and Historic Computing*, **6875**, pages 283–297. Springer Berlin Heidelberg, 2011. 3, 5, 13, 86, 92

[29] J.Kleijn, M.Koutny, and G.Rozenberg. Towards a Petri net semantics for membrane systems. In Rudolf Freund, Gheorghe PĂČun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*,

3850 of *Lecture Notes in Computer Science*, pages 292–309. Springer Berlin Heidelberg, 2006. 39

[30] K.Heljanko. *Deadlock and reachability checking with finite complete prefixes*. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, F.Espoo, December 1999. Also available as: Series A: Research Report 56, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science. 32, 119

[31] K.Heljanko, V.Khomenko, and M.Koutny. Parallelisation of the Petri net unfolding algorithm. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 371–385, London, UK, UK, 2002. Springer-Verlag. 115

[32] K.Jeroen and S.Burkhard. Knowledge based crime scenario modelling. *Expert Syst. Appl.*, **30**[2]:203–222, 2006. 118

[33] H.C.M. Kleijn and M. Koutny. Infinite process semantics of inhibitor nets. In S.Donatelli and P.Thiagarajan, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2006*, **4024** of *Lecture Notes in Computer Science*, pages 282–301. Springer Berlin Heidelberg, 2006. 14

[34] K.L.McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. 87

[35] K.L.McMillan. and DK.Probst. A technique of state space search based on unfolding. *Formal Methods in System Design*, **6**[1]:45–65, January 1995. 3, 87, 89, 104

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

[37] M.A.Marsan, G.Balbo., G.Conte, S.Donatelli, and G.Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994. 54

[38] M.Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, **335**[2-3]:131–146, May 2005. 2

[39] M.Diaz and Patrick Sénac. *Application and Theory of Petri Nets 1994: 15th International Conference Zaragoza, Spain, June 20–24, 1994 Proceedings*, chapter Time stream Petri nets a model for timed multimedia information, pages 219–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. 54

[40] M.Koutny and B.Randell. Structured occurrence nets: a formalism for aiding system failure prevention and analysis techniques. *Fundamenta Informaticae*, **97**[1]:41–91, January 2009. 3, 9, 11, 13, 16, 21, 26, 28, 29, 37

[41] M.Koutny and B.Randell. Structured occurrence nets: incomplete, contradictory and uncertain failure evidence. Technical Report CS-TR-1170, School of Computing Science, Newcastle University, 2009. 3, 4, 38, 39, 40, 118

[42] M.Z.Kwiatkowska, G.Norman, and D.Parker. PRISM 4.0: verification of probabilistic real-time systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 585–591, 2011. 118

[43] N.Bobak and M.Gastpar. Computation over multiple-access channels. *Information Theory, IEEE Transactions on*, **53**[10]:3498–3516, 2007. 70

[44] P.Baldan, T.Chatain, S.Haar, and B.KÃűnig. Unfolding-based diagnosis of systems with an evolving topology. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory*, **5201** of *Lecture Notes in Computer Science*, pages 203–217. Springer Berlin Heidelberg, 2008. 3

[45] P.Bouyer, S.Haddad, and P.Reynier. *Timed unfoldings for networks of timed automata*, pages 292–306. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 87

[46] P.Darondeau, S.Demri, R.Meyer, and C.Morvan. Petri net reachability graphs: decidability status of FO properties. In Supratik Chakraborty and Amit Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011)*, **13** of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 140–151, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 32

[47] P.Missier, B.Randell, and M.Koutny. Modelling provenance using structured occurrence networks. In *Provenance and Annotation of Data and Processes - 4th International Provenance and Annotation Workshop, IPAW 2012, Santa Barbara, CA, USA, June 19-21, 2012, Revised Selected Papers*, pages 183–197, 2012. 118

[48] R.Janicki and M.Koutny. Invariants and paradigms of concurrency theory. In E.Aarts, J.Leeuwen, and M.Rem, editors, *PARLE '91 Parallel Architectures and Languages Europe*, **506** of *Lecture Notes in Computer Science*, pages 59–74. Springer Berlin Heidelberg, 1991. 14

[49] R.Langerak and E.Brinksma. A complete finite prefix for process algebra. In N.Halbwachs and P.Doron, editors, *Computer Aided Verification*, **1633** of *Lecture Notes in Computer Science*, pages 184–195. Springer Berlin Heidelberg, 1999. 3

[50] A. Romanovsky. A study of atomic action schemes intended for standard Ada. *Journal of Systems and Software*, pages 29–44, 1998. 27

[51] R.Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972. 31

[52] R.Wilson. Date range propagation in genealogical databases. In *Family History Technology Workshop*, 2012. 71

[53] S.Bran. Using UML for modeling complex real-time systems. In *Languages, Compilers, and Tools for Embedded Systems*, pages 250–260. Springer, 1998. 2

[54] Computer Science and Technology Board. Scaling up: a research agenda for software engineering. *Communications of the ACM*, **33**:281–293, 1990. 2

[55] S.Melzer and S.Römer. *Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings*, chapter Deadlock checking using net unfoldings, pages 352–363. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. 119

[56] S.Neville and G.Walker. Exploring the psychological factors involved in the ladbroke grove rail accident. *Accident Analysis and Prevention*, **43**[3]:1117 – 1127, 2011. 35

[57] T.Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, **77**[4]:541–580, April 1989. 2, 30

[58] V.Brox. *Date estimation in lineage-linked databases*. BSc dissertation, Newcastle University School of Computing Science, 2000. 71

[59] V.Khomenko, A.Kondratyev, M.Koutny, and W.Vogler. Merged processes: a new condensed representation of Petri net behaviour. *Acta Informatica*, **43**[5]:307–330, 2006. 87

[60] V.Khomenko, M.Koutny, and W.Vogler. Canonical prefixes of Petri net unfoldings. *Acta Inf.*, **40**[2]:95–118, 2003. 104

[61] W.D.Cullen. The Ladbroke Grove rail enquiry: part 1 report. Technical report, London: HMSO, 2001. 35

[62] W.Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985. 32, 89