Computation of summaries using net unfoldings

Javier Esparza[1], Loïg Jezequel[2], and Stefan Schwoon[3]

1 **Institut für Informatik, Technische Universität München, Germany**
2 **ENS Cachan Bretagne, Rennes, France**
3 **LSV, ENS Cachan & CNRS, INRIA Saclay, France**

Abstract

We study the following *summarization problem*: given a parallel composition $\mathbf{A} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ of labelled transition systems communicating with the environment through a distinguished component $\mathcal{A}_i$, efficiently compute a *summary* $\mathcal{S}_i$ such that $\mathbf{E} \parallel \mathbf{A}$ and $\mathbf{E} \parallel \mathcal{S}_i$ are trace-equivalent for every environment $\mathbf{E}$. While $\mathcal{S}_i$ can be computed using elementary automata theory, the resulting algorithm suffers from the state-explosion problem. We present a new, simple but subtle algorithm based on net unfoldings, a partial-order semantics, give some experimental results using an implementation on top of MOLE, and show that our algorithm can handle *divergences* and compute *weighted summaries* with minor modifications.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, F.1.2 Modes of Computation

Keywords and phrases Net unfoldings, Concurrent systems, Petri nets

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

We address a fundamental problem in automatic compositional verification. Consider a parallel composition $\mathbf{A} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ of processes, modelled as labelled transition systems, which is itself part of a larger system $\mathbf{E} \parallel \mathbf{A}$ for some environment $\mathbf{E}$. Assume that $\mathcal{A}_i$ is the interface of $\mathbf{A}$ with the environment, i.e., $\mathbf{A}$ communicates with the outer world only through actions of $\mathcal{A}_i$. The task consists in computing a new interface $\mathcal{S}_i$ with the same set of actions as $\mathcal{A}_i$ such that $\mathbf{E} \parallel \mathbf{A}$ and $\mathbf{E} \parallel \mathcal{S}_i$ have the same behaviour. In other words, the environment E cannot distinguish between $\mathbf{A}$ and $\mathcal{S}_i$. Since $\mathcal{S}_i$ usually has a much smaller state space than $\mathbf{A}$ (making $\mathbf{E} \parallel \mathbf{A}$ easier to analyse) we call it a *summary*.

We study the problem in a CSP-like setting [13]: parallel composition is by rendez-vous, and the behaviour of a transition system is given by its trace semantics.

It is easy to compute $\mathcal{S}_i$ using elementary automata theory: we first compute the transition system of $\mathbf{A}$, whose states are tuples $(s_1, \ldots, s_n)$, where s_i is a state of $\mathcal{A}_i$. Then we hide all actions except those of the interface, i.e., we replace them by ε-transitions (τ-transitions in CSP terminology). We can then eliminate all ε-transitions using standard algorithms, and, if desired, compute the minimal summary by applying e.g. Hopcroft's algorithm. The problem of this approach is the state-space explosion: the number of states of $\mathbf{A}$ can grow exponentially in the number of sequential components. While this is unavoidable in the worst case (deciding whether $\mathcal{S}_i$ has an empty set of traces is a PSPACE-complete problem, and the minimal summary $\mathcal{S}_i$ may be exponentially larger than $\mathcal{A}_1, \ldots, \mathcal{A}_n$ in the worst case, see e.g. [11]) the combinatorial explosion happens already in trivial cases: if the components $\mathcal{A}_1, \ldots, \mathcal{A}_n$ do not communicate at all, we can obviously take $\mathcal{S}_i = \mathcal{A}_i$, but the algorithm we have just described will need exponential time and space.

We present a technique to palliate this problem based on net unfoldings (see e.g. [4]). Net unfoldings are a partial-order semantics for concurrent systems, closely related to event structures [22], that provides very compact representations of the state space for systems with a high degree of concurrency. Intuitively, an unfolding is the extension to parallel compositions of the notion of unfolding a transition system into a tree. The unfolding is usually infinite. We show how to algorithmically construct a finite prefix of it from which the summary can be easily extracted. The algorithm can be easily implemented re-using many components of existing unfolders like Punf and Mole. However, its correctness proof is surprisingly subtle. This proof is the main contribution of the paper. However, we also evaluate the algorithm on some classical benchmarks [2]. We then show that – with minor modifications – the algorithm can be extended so that the summary obtained contains information about the possible divergences, that is whether or not after a given finite trace of the interface $\mathcal{A}_i$ it is possible that $\mathbf{A}$ evolves silently forever (i.e. without using any action of $\mathcal{A}_i$). And finally, we show how to extend the algorithm to deal with weighted systems: $\mathcal{S}_i$ then also gives for each of its finite traces the minimum cost in $\mathbf{A}$ to execute this trace.

Related work. The summarization problem has been extensively studied in an interleaving setting (see e.g. [10, 21, 23]), in which one first constructs the transition system of $\mathbf{A}$ and then reduces it. We study it in a partial-order setting.

Net unfoldings, and in general partial-order semantics, have been used to solve many analysis problems: deadlock [18, 15], reachability and model-checking questions [6, 3, 14, 4, 1], diagnosis [7], and other specific applications [17, 12]. To the best of our knowledge we are the first to explicitly study the summarization problem.

Our problem can be solved with the help of Zielonka's algorithm [24, 19, 9], which yields an asynchronous automaton trace-equivalent to $\mathbf{A}$. The projection of this automaton onto the alphabet of $\mathcal{A}_i$ yields a summary $\mathcal{S}_i$. However, Zielonka's algorithm is notoriously complicated and, contrary to our algorithm, requires to store much additional information for each event [19]. In [8], the complete tuple $\mathcal{S}_1, \ldots, \mathcal{S}_n$ is computed – possibly in a weighted context – with an iterative message-passing algorithm that transfers information between components until a fixed point is reached. However, termination is only guaranteed when the communication graph is acyclic.

2 Preliminaries

Transition systems. A *labelled transition system* (LTS) is a tuple $\mathcal{A} = (\Sigma, S, T, \lambda, s^0)$ where Σ is a set of *actions*, S is a set of *states*, $T \subseteq S \times S$ is a set of *transitions*, $\lambda \colon T \to \Sigma$ is a *labelling function*, and $s^0 \in S$ is an *initial state*. An a-transition is a transition labelled by a. We use this definition – excluding the possibility to have two transitions with different labels between the same pair of states – for simplicity. However, the results presented in this paper would still hold if this possibility was not excluded. A (finite or infinite) action sequence $\sigma = a_1 a_2 a_3 \ldots \in \Sigma^* \cup \Sigma^\omega$ is a *trace* of $\mathcal{A}$ if there is a (finite or infinite) sequence $s_0 s_1 s_2 \ldots$ of states such that $s_0 = s^0$, $t_i = (s_{i-1}, s_i) \in T$ and $\lambda(t_i) = a_i$ for every i. The path $s_0 s_1 s_2 \ldots$ is a *realization* of σ. The set of traces of $\mathcal{A}$ is denoted by $Tr(\mathcal{A})$. Figure 1 shows (on its left) three transition systems.

Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be LTSs where $\mathcal{A}_i = (\Sigma_i, S_i, T_i, \lambda_i, s_i^0)$. The *parallel composition* $\mathbf{A} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ is the LTS defined as follows. The set of actions is $\mathbf{\Sigma} = \Sigma_1 \cup \ldots \cup \Sigma_n$. The states, called *global states*, are the tuples $\mathbf{s} = (s_1, \ldots, s_n)$ such that $s_i \in S_i$ for every $i \in \{1..n\}$. The *initial global state* is $\mathbf{s}^0 = (s_1^0, \ldots, s_n^0)$. The transitions, called *global transitions*, are the tuples $\mathbf{t} = (t_1, \ldots, t_n) \in (T_1 \cup \{\star\}) \times \cdots \times (T_n \cup \{\star\}) \setminus \{(\star, \ldots, \star)\}$ such that there is

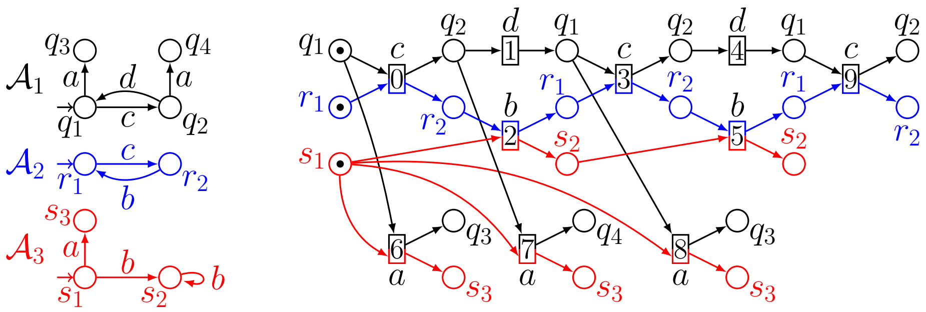

Figure 1 Three labeled transition systems (left) and a branching process (right)

an action $a \in \mathbf{\Sigma}$ satisfying for every $i \in \{1..n\}$: if $a \in \Sigma_i$, then t_i is an a-transition of T_i, otherwise $t_i = \star$; the label of $\mathbf{t}$ is the action a. If $t_i \neq \star$ we say that $\mathcal{A}_i$ *participates* in $\mathbf{t}$. It is easy to see that $\sigma \in \mathbf{\Sigma}^* \cup \mathbf{\Sigma}^\omega$ is a trace of $\mathbf{A}$ iff for every $i \in \{1..n\}$ the projection of σ on Σ_i, denoted by $\sigma_{|\Sigma_i}$, is a trace of $\mathcal{A}_i$.

Petri nets. A *labelled net* is a tuple $(\Sigma, P, T, F, \lambda)$ where Σ is a set of *actions*, P and T are disjoint sets of *places* and *transitions* (jointly called *nodes*), $F \subseteq (P \times T) \cup (T \times P)$ is a set of *arcs*, and $\lambda \colon P \cup T \to \Sigma$ is a *labelling function*. For $x \in P \cup T$ we denote by ${}^\bullet x = \{\, y \mid (y, x) \in F \,\}$ and $x^\bullet = \{\, y \mid (x, y) \in F \,\}$ the sets of *inputs* and *outputs* of x, respectively. A set M of places is called a *marking*. A *labelled Petri net* is a tuple $\mathcal{N} = (\Sigma, P, T, F, \lambda, M_0)$ where $(\Sigma, P, T, F, \lambda)$ is a labelled net and $M_0 \subseteq P$ is the *initial marking*. A marking M *enables* a transition $t \in T$ if ${}^\bullet t \subseteq M$. In this case t can *occur* or *fire*, leading to the new marking $M' = (M \setminus {}^\bullet t) \cup t^\bullet$. An *occurrence sequence* is a (finite or infinite) sequence of transitions that can occur from M_0 in the order specified by the sequence. A *trace* is the sequence of labels of an occurrence sequence. The set of traces of $\mathcal{N}$ is denoted by $Tr(\mathcal{N})$.

Branching processes. The finite *branching processes* of $\mathbf{A} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ are labelled Petri nets whose places are labelled with states of $\mathcal{A}_1, \ldots, \mathcal{A}_n$, and whose transitions are labelled with global transitions of $\mathbf{A}$. Following tradition, we call the places and transitions of these nets *conditions* and *events*, respectively. (Since global transitions are labelled with actions, each event is also implicitly labelled with an action.) We say that a marking M of these nets *enables* a global transition $\mathbf{t}$ of $\mathbf{A}$ if for every state $s \in {}^\bullet\mathbf{t}$ some condition of M is labelled by s. The set of *finite branching processes* of $\mathbf{A}$ is defined inductively as follows:

1. A labelled Petri net with conditions $b_1^0, ..., b_n^0$ labelled by $s_1^0, \ldots, s_n^0$, no events, and with initial marking $\{b_1^0, ..., b_n^0\}$, is a branching process of $\mathbf{A}$.
2. Let $\mathcal{N}$ be a branching process of $\mathbf{A}$ such that some reachable marking of $\mathcal{N}$ enables some global transition $\mathbf{t}$. Let M be the subset of conditions of the marking labelled by ${}^\bullet\mathbf{t}$. If $\mathcal{N}$ has no event labelled by $\mathbf{t}$ with M as input set, then the Petri net obtained by adding to $\mathcal{N}$: a new event e, labelled by $\mathbf{t}$; a new condition for every state s of $\mathbf{t}^\bullet$, labelled by s; new arcs leading from each condition of M to e, and from e to each of the new conditions, is also a branching process of $\mathbf{A}$.

Figure 1 shows on the right a branching process of the parallel composition of the LTSs on the left. Events are labelled with their corresponding actions.

The set of all branching processes of a net, finite and infinite, is defined by closing the finite branching processes under countable unions (after a suitable renaming of conditions and events) [4]. In particular, the union of all finite branching processes yields the *unfolding* of the net, which intuitively corresponds to the result of exhaustively adding all extensions in the definition above.

A *trace* of a branching process $\mathcal{N}$ is the sequence of action labels of an occurrence sequence of events of $\mathcal{N}$. In Figure 1, firing the events on the top half of the process yields any of the traces *cbdcbd*, *cdbcbd*, *cbdcdb*, or *cdbcdb*. The sets of traces of $\mathbf{A}$ and of its unfolding coincide.

Let x, y be nodes of a branching process. We say that x is a *causal predecessor* of y, denoted by $x < y$, if there is a non-empty path of arcs from x to y; further, $x \leq y$ denotes that either $x < y$ or $x = y$. If $x \leq y$ or $x \geq y$, then x and y are *causally related.* We say that x and y are *in conflict*, denoted by $x \# y$, if there is a condition z (different from x and y) from which one can reach both x and y, exiting z by different arcs. Finally, x and y are *concurrent* if they are neither causally related nor in conflict.

A set of events E is a *configuration* if it is *causally closed* (that is, if $e \in E$ and $e' < e$ then $e' \in E$) and *conflict-free* (that is, for every $e, e' \in E$, e and e' are not in conflict). The *past* of an event e, denoted by $[e]$, is the set of events e' such that $e' \leq e$ (so it is a configuration). For any event e, we denote by $M(e)$ the unique marking reached by any occurrence sequence that fires exactly the events of $[e]$. Notice that, for each component $\mathcal{A}_i$ of $\mathbf{A}$, $M(e)$ contains exactly one condition labelled by a state of $\mathcal{A}_i$. We denote this condition by $M(e)_i$. We write $\mathbf{St}(e) = \{\, \lambda(x) \mid x \in M(e) \,\}$ and call it the *global state reached by e.*

3 The Summary Problem

Let $\mathbf{A} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ be a parallel composition with a distinguished component $\mathcal{A}_i$, called the *interface.* An *environment* of $\mathbf{A}$ is any LTS $\mathbf{E}$ (possibly a parallel composition) that only communicates with $\mathbf{A}$ through the interface, i.e, $\Sigma_\mathbf{E} \cap (\Sigma_1 \cup \ldots \cup \Sigma_n) = \Sigma_\mathbf{E} \cap \Sigma_i$. We wish to compute a *summary* $\mathcal{S}_i$, i.e., an LTS with the same actions as $\mathcal{A}_i$ such that $\mathit{Tr}(\mathbf{E} \parallel \mathbf{A})|_{\Sigma_\mathbf{E}} = \mathit{Tr}(\mathbf{E} \parallel \mathcal{S}_i)|_{\Sigma_\mathbf{E}}$ for every environment $\mathbf{E}$, where $X|_\Sigma$ denotes the projection of the traces of X onto Σ. It is well known (and follows easily from the definitions) that this holds iff $\mathit{Tr}(\mathcal{S}_i) = \mathit{Tr}(\mathbf{A})|_{\Sigma_i}$ [13]. We therefore address the following problem:

▶ **Definition 1** (Summary problem)**.** Given LTSs $\mathcal{A}_1, \ldots, \mathcal{A}_n$ with interface $\mathcal{A}_i$, compute an LTS $\mathcal{S}_i$ satisfying $\mathit{Tr}(\mathcal{S}_i) = \mathit{Tr}(\mathbf{A})|_{\Sigma_i}$, where $\mathbf{A} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$.

The problem can be solved by computing the LTS $\mathbf{A}$, but the size of $\mathbf{A}$ can be exponential in $\mathcal{A}_1, \ldots, \mathcal{A}_n$. So we investigate an unfolding approach.

The *interface projection* $\mathcal{N}_i$ of a branching process $\mathcal{N}$ of $\mathbf{A}$ onto $\mathcal{A}_i$ is the following labelled subnet of $\mathcal{N}$: (1) the conditions of $\mathcal{N}_i$ are the conditions of $\mathcal{N}$ with labels in S_i; (2) the events of $\mathcal{N}_i$ are the events of $\mathcal{N}$ where $\mathcal{A}_i$ participates; (3) (x, y) is an arc of $\mathcal{N}_i$ iff it is an arc of $\mathcal{N}$ and (x, y) are nodes of $\mathcal{N}_i$. Obviously, every event of $\mathcal{N}_i$ has exactly one input and one output condition, and $\mathcal{N}_i$ can therefore be seen as an LTS; thus, we sometimes speak of the LTS $\mathcal{N}_i$. The interface projection $\mathcal{N}_1$ for the branching process of Figure 1 is the subnet given by the black conditions and their input and output events, and its LTS representation is shown in the left of Figure 2.

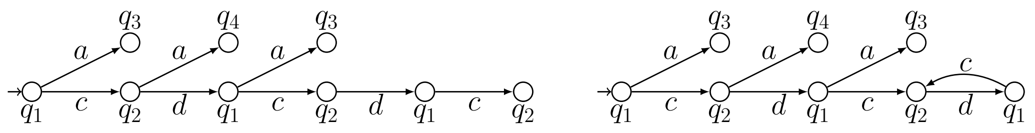

Figure 2 Projection of the branching process of Figure 1 on $\mathcal{A}_1$ (left) and a folding (right)

The projection $\mathcal{U}_i$ of the full unfolding of $\mathbf{A}$ onto $\mathcal{A}_i$ clearly satisfies $\mathit{Tr}(\mathcal{U}_i) = \mathit{Tr}(\mathbf{A})_{|\Sigma_i}$; however, $\mathcal{U}_i$ can be infinite. In the rest of the paper we show how to compute a *finite* branching process $\mathcal{N}$ and an equivalence relation $\equiv$ between the conditions of $\mathcal{N}_i$ such that the result of *folding* $\mathcal{N}_i$ into a finite LTS by merging the conditions of each equivalence class yields the desired $\mathcal{S}_i$. The *folding* of $\mathcal{N}_i$ is the LTS whose states are the equivalence classes of $\equiv$, and every transition (s, s') of $\mathcal{N}_i$ yields a transition $([s]_\equiv, [s']_\equiv)$ of the folding. Figure 2 shows on the right the result of folding the LTS on the left when the only equivalence class with more than one member is formed by the two rightmost states labelled by q_2.

We construct $\mathcal{N}$ by starting with the branching processes without events and iteratively add one event at a time. Some events are marked as *cut-offs* [4]. An event e added to $\mathcal{N}$ becomes a cut-off if $\mathcal{N}$ already contains an e', called the *companion* of e, satisfying a certain, yet to be specified *cut-off criterion*. Events with cut-offs in their past cannot be added. The algorithm terminates when no more events can be added. The equivalence relation $\equiv$ is determined by the *interface cut-offs*: the cut-offs labelled with interface actions. If an interface cut-off e has companion e', then we set $M(e)_i \equiv M(e')_i$. Algorithm 1 is pseudocode for the unfolding, where $Ext(\mathcal{N}, co)$ denotes the *possible extensions*: the events which can be added to $\mathcal{N}$ without events from the set co of cut-offs in their past.

Algorithm 1 Unfolding procedure for a product **A**.

```
let $\mathcal{N}$ be the unique branching process of $\mathbf{A}$ without events and let $co = \emptyset$
While $Ext(\mathcal{N}, co) \neq \emptyset$ do
    choose $e$ in $Ext(\mathcal{N}, co)$ and extend $\mathcal{N}$ with $e$
    If $e$ is a cut-off event then let $co = co \cup \{e\}$
For every $e \in co$ with companion $e'$ do merge $[M(e)_i]_\equiv$ and $[M(e')_i]_\equiv$
```

Notice that the algorithm is nondeterministic: the order in which events are added is not fixed (though it necessarily respects causal relations). We wish to find a definition of cut-offs such that the LTS $\mathcal{S}_i$ delivered by the algorithm is a correct solution to the summary problem. Several papers have addressed the problem of defining cut-offs such that the branching process delivered by the algorithm contains all global states of the system (see [4] and the references therein). In [5] we show that these approaches do not "unfold enough".

4 Two Attempts

The solution turns out to be remarkably subtle, and so we approach it in a series of steps.

4.1 First attempt

In the following we shall call events in which $\mathcal{A}_i$ participates *i-events* for short; analogously, we call *i-conditions* the conditions labelled by states of $\mathcal{A}_i$.

The simplest idea is to declare an i-event e a cut-off if the branching process already contains another i-event e' with $\mathbf{St}(e) = \mathbf{St}(e')$. Intuitively, the behaviours of the interface after the configurations $[e]$ and $[e']$ is identical, and so we only explore the future of $[e']$.

Cut-off definition 1. An event e is a cut-off event if it is an i-event and $\mathcal{N}$ contains an i-event e' such that $\mathbf{St}(e) = \mathbf{St}(e')$.

It is not difficult to show that this definition is correct for *non-divergent* systems.

▶ **Definition 2.** A parallel composition **A** with interface $\mathcal{A}_i$ is *divergent* if some infinite trace of **A** contains only finitely many occurrences of actions of Σ_i.

▶ **Theorem 3.** *Let* $\mathbf{A}$ *be non-divergent. The instance of Algorithm 1 with cut-off definition 1 terminates with a finite branching process* $\mathcal{N}$*, and the folding* $\mathcal{S}_i$ *of* $\mathcal{N}_i$ *is a summary of* $\mathbf{A}$*.*

The proof of this theorem is given in [5].

The system of Figure 1 is non-divergent. Algorithm 1 computes the branching process on the right of Figure 1. The only cut-off is event 9 with companion 3. The folding is shown in Figure 2 (right) and is a correct summary. However, cut-off definition 1 *never* works if $\mathbf{A}$ is divergent because the unfolding procedure does not terminate. Indeed, if the system has divergent traces then we can easily construct an infinite firing sequence of the unfolding such that none of the finitely many i-events in the sequence is a cut-off. Since no other events can be cut-offs, Algorithm 1 adds all events of the sequence. This occurs for instance for the system of Figure 3 with interface $\mathcal{A}_1$, where the occurrence sequence of the unfolding for the trace $i(\mathit{fcd})^\omega$ contains no cut-off.

4.2 Second attempt

To ensure termination for divergent systems, we extend the definition of cut-off. For this, we define for each event e its *i-predecessor*. Intuitively, the i-predecessor of an event e is the last condition that e "knows" has been reached by the interface.

▶ **Definition 4.** The *i-predecessor* of an event e, denoted by $\mathit{ip}(e)$, is the condition $M(e)_i$.

Assume now that two events $e_1 < e_2$, neither of them interface event, satisfy $\mathit{ip}(e_1) = \mathit{ip}(e_2)$ and $\mathbf{St}(e_1) = \mathbf{St}(e_2)$. Then any occurrence sequence σ that executes the events of the set $[e_2] \setminus [e_1]$ leads from a marking to itself and contains no interface events. So σ can be repeated infinitely often, leading to an infinite trace with only finitely many interface actions. It is therefore plausible to mark e_2 as cut-off event, in order to avoid this infinite repetition.

Cut-off definition 2. An event e is a cut-off if
(1) e is an i-event, and $\mathcal{N}$ contains an i-event e' with $\mathbf{St}(e) = \mathbf{St}(e')$, or
(2) e is not an i-event, and some event $e' < e$ satisfies $\mathbf{St}(e) = \mathbf{St}(e')$ and $\mathit{ip}(e) = \mathit{ip}(e')$.

We give an example showing that this natural definition does not work: the algorithm always terminates but can yield a wrong result. Consider the parallel composition at the left of Figure 3, with interface $\mathcal{A}_1$. Clearly $\mathit{Tr}(\mathcal{A})|_{\Sigma_1} = \mathit{Tr}(A_1) = \mathit{iab}^*e$. For any strategy the algorithm generates the branching process $\mathcal{N}$ at the top right of the figure (without the dashed part). $\mathcal{N}$ has two cut-off events: the interface event 6, which is of type (1), and event 8, a non-interface event, of type (2). Event 6 has 5 as companion, with $\mathbf{St}(5) = \mathbf{St}(6) = \{q_2, r_2, s_2\}$. Event 8 has 0 as companion, with $\mathbf{St}(0) = \{q_1, r_1, s_1\} = \mathbf{St}(8)$; moreover, $0 < 8$ and $\mathit{ip}(0) = \mathit{ip}(8)$. The folding of $\mathcal{N}_1$ is shown at the bottom right of the figure. It is clearly not trace-equivalent to $\mathcal{A}_1$ because it "misses" the trace *iabe*. The dashed event at the bottom right, which would correct this, is not added by the algorithm because it is a successor of 8.

5 The Solution

Intuitively, the reason for the failure of our second attempt on the example of Figure 3 is that $\mathcal{A}_1$ can only execute *iabe* if $\mathcal{A}_2$ and $\mathcal{A}_3$ execute *ifcd* first. However, when the algorithm observes that the markings before and after the execution of *ifcd* are identical, it declares 8 a cut-off event, and so it cannot "use" it to construct event e. So, on the one hand, 8 should not be a cut-off event. But, on the other hand, *some* event of the trace $i(\mathit{fcd})^\omega$ must be declared cut-off, otherwise the algorithm does not terminate.

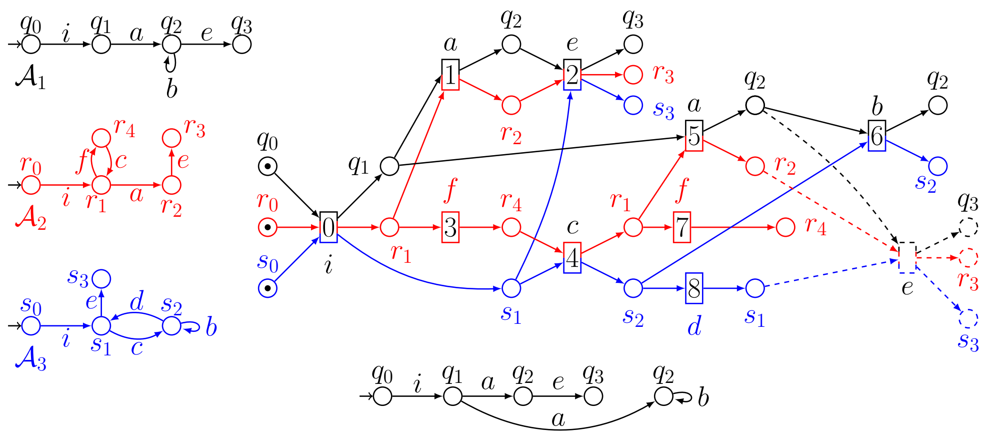

Figure 3 Cut-off definition 2 produces an incorrect result on $\mathbf{A} = \mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \mathcal{A}_3$

The way out of this dilemma is to introduce *cut-off candidates*. If an event is declared a cut-off candidate, the algorithm does not add any of its successors, just as with regular cut-offs. However, cut-off candidates may stop being candidates if the addition of a new event *frees* them. (So, an event is a cut-off candidate *with respect to the current branching process*.) A generic unfolding procedure using these ideas is given in Algorithm 2, where $Ext(\mathcal{N}, co, coc)$ denotes the possible extensions of $\mathcal{N}$ that do not have any event of co or coc in their past. Assuming suitable definitions of cut-off candidates and freeing, the algorithm would, in our example, declare event 8 a cut-off candidate, momentarily stop adding any of its successors, but later free event 8 when event 5 is discovered.

Algorithm 2 Unfolding procedure for a product **A**.

```
let $\mathcal{N}$ be the unique branching process of **A** without events; let $co = \emptyset$ and $coc = \emptyset$
**While** $Ext(\mathcal{N}, co, coc) \neq \emptyset$ **do**
    choose $e$ in $Ext(\mathcal{N}, co, coc)$ according to the search strategy
    **If** $e$ is a cut-off event **then** let $co = co \cup \{e\}$
    **Elseif** $e$ is a cut-off candidate of $\mathcal{N}$ **then** let $coc = coc \cup \{e\}$
    **Else for every** $e' \in coc$ **do**
        **If** $e$ frees $e'$ **then** $coc = coc \setminus \{e'\}$
    extend $\mathcal{N}$ with $e$
**For every** $e \in co$ with companion $e'$ **do** merge $[M(e)_i]_\equiv$ and $[M(e')_i]_\equiv$
```

The main contribution of our paper is the definition of a correct notion of cut-off candidate for the projection problem. We shall declare event e a cut-off candidate if e is not an interface event, and $\mathcal{N}$ contains a companion $e' < e$ such that $\mathbf{St}(e') = \mathbf{St}(e)$, $ip(e) = ip(e')$, and, additionally, no interface event e'' of $\mathcal{N}$ is concurrent with e without being concurrent with e'. As long as this condition holds, the successors of e are put “on hold”. In the example of Figure 3, if the algorithm first adds events 0, 3, 4, and 8, then event 8 becomes a cut-off candidate with 0 as companion. However, the addition of the interface event 5 frees event 8, because 5 is concurrent with 8 and not with 0.

However, we are not completely done yet. The parallel composition at the left of Figure 4 gives an example in which even with this notion of cut-off candidate the result is still wrong. $\mathcal{A}_1$ is the interface. One branching process is represented at the top right of the figure. Event 3 (concurrent with 1) is a cut-off candidate with 2 (concurrent with 1, 4, and 5) as companion. This prevents the lower dashed part of the net to be added. Event 6 is cut-off with 1 as companion. This prevents the upper dashed part of the net to be added. The refolding obtained then (bottom right) does not contain the word *abcb*.

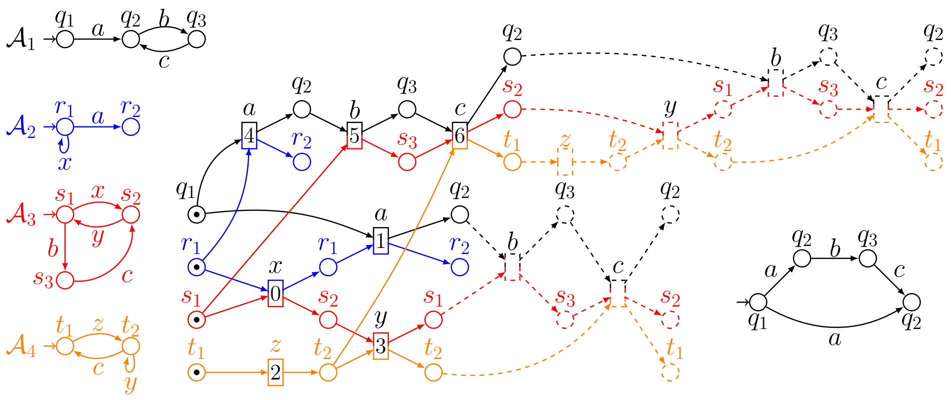

Figure 4 An example illustrating the use of strong causality

If we wish a correct algorithm for all strategies, we need a final touch: replace the condition $e' < e$ by $e' \ll e$, where $\ll$ is the *strong causal relation*:

▶ **Definition 5.** Event e' is a *strong cause* of event e, denoted by $e' \ll e$, if $e' < e$ and $b' < b$ for every $b \in M(e) \setminus M(e'), b' \in M(e') \setminus M(e)$.

Using this definition, event 3 is no longer a cut-off candidate in the branching process of Figure 4 as it is not in strong causal relation with its companion 2 (because the t_2-labelled condition just after 2 belongs to $M(2) \setminus M(3)$ and is not causally related with the r_1-labelled condition just after 0 which belongs to $M(3) \setminus M(2)$).

We are now in a position to provide adequate definitions for Algorithm 2.

▶ **Definition 6** (Cut-off and cut-off candidate)**.** Let $Ico_{\mathcal{N}}(e)$ denote the set of non cut-off interface events of $\mathcal{N}$ that are concurrent with e. An event e

- is a cut-off if it is an i-event, and $\mathcal{N}$ contains an i-event e' such that $\mathbf{St}(e) = \mathbf{St}(e')$.
- is a cut-off candidate of $\mathcal{N}$ if it is not an i-event, and $\mathcal{N}$ contains $e' \ll e$ such that $\mathbf{St}(e) = \mathbf{St}(e')$, $ip(e') = ip(e)$, and $Ico_{\mathcal{N}}(e) \subseteq Ico_{\mathcal{N}}(e')$.
- frees a cut-off candidate e_c of $\mathcal{N}$ if e_c is not a cut-off candidate of the branching process obtained by adding e to $\mathcal{N}$.

▶ **Theorem 7.** *Let* $\mathbf{A} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ *with interface* $\mathcal{A}_i$*. The instance of Algorithm 2 given by Definition 6 terminates and returns a branching process* $\mathcal{N}$ *such that the folding* $\mathcal{S}_i$ *of* $\mathcal{N}_i$ *is a summary of* $\mathbf{A}$*.*

The proof of this theorem is involved. It is given in [5]. We sketch the main ideas. Termination follows from a lemma showing that every infinite chain of causally related events contains an infinite subchain of strongly causally related events. The equality $Tr(\mathcal{S}_i) = Tr(\mathbf{A})|_{\Sigma_i}$ is proved in two parts. $Tr(\mathcal{S}_i) \subseteq Tr(\mathbf{A})|_{\Sigma_i}$ follows easily from the definitions. The proof of $Tr(\mathcal{S}_i) \supseteq Tr(\mathbf{A})|_{\Sigma_i}$ is more involved. For every trace of $\mathbf{A}$ we identify a *strongly succinct* occurrence sequence in the unfolding with this trace as projection. Intuitively, in such a sequence, interface events occur as early as possible, and the number of non-interface events occurring between them is minimal. The main point in the proof is to show that every cut-off in strongly succinct sequences is necessarily an interface event, which allows one to conclude the proof as in the non-divergent case. This is proved by contradiction: If e is a cut-off candidate with companion e', we show that (1) e and e' are located between the same two interface events (this uses $Ico_{\mathcal{N}}(e) \subseteq Ico_{\mathcal{N}}(e')$), (2) there is no i-event in $[e] \setminus [e']$, and (3) every event of $[e] \setminus [e']$ is also located between the same two interface events (this is ensured by $e' \ll e$). The events from $[e] \setminus [e']$ can then be removed from the sequence, contradicting the definition of strongly succinct sequence.

6 Implementation and Experiments

As an illustration of the previous results, we report in this section on an implementation of Algorithm 2 as a modification of the existing unfolding tool Mole. All programs and data used are publicly available.[1] Many components of Mole could be re-used. The main work consisted in determining cut-off candidates and the "freeing" condition of Definition 6. This required two main algorithmic additions discussed in detail in [5]: (i) an efficient traversal of $[e]$, for a given event e, that allows to determine the conditions for cut-off candidates; (ii) computing $Ico_N(e)$ for an event e. Both additions could be obtained by extending existing components of the tool. While the additions were not always trivial, they could be obtained with small additional overhead.

We tested our implementation on well-known benchmarks used widely in the unfolding literature, see e.g. [2, 6, 16]. The input is the set of components $\mathcal{A}_1, \ldots, \mathcal{A}_n$, which are converted into an equivalent Petri net. For each example, we report the number of events (including cut-offs) in the prefix. Notice that this prefix is computed in less than one second in most cases (more detailed experimental results are given in [5]). We also report the number of reachable markings (taken from [20] where available, and computed combinatorially for DpSyn).

The experiments are summarized in Table 1. We used the following families of examples [2]: the CyclicC and CyclicS families are a model of Milner's cyclic scheduler with n consumers and n schedulers; in one case we compute the folding for a consumer, in the other for a scheduler. The Dac family represents a divide-and-conquer computation. Ring is a mutual-exclusion protocol on a token-ring. The tasks are not entirely symmetric, we report the results for the first. Finally, Dp, Dpsyn, and Dpd are variants of Dining Philosophers. In Dp, philosophers take and release forks one by one, whereas in Dpsyn they take and release both at once. In Dpd, deadlocks are prevented by passing a dictionary.

Test case	Events	Markings
CyclicC(6)	426	639
CyclicC(9)	3347	7423
CyclicC(12)	26652	74264
CyclicS(6)	303	639
CyclicS(9)	2328	7423
CyclicS(12)	18464	74264
Dac(9)	86	1790
Dac(12)	134	14334
Dac(15)	191	114686
Dp(6)	935	729
Dp(8)	5121	6555
Dp(10)	31031	48897
Dpd(4)	2373	601
Dpd(5)	23789	3489
Dpd(6)	245013	19861
Dpsyn(10)	176	123
Dpsyn(20)	701	15127
Dpsyn(30)	1576	1860498
Ring(5)	511	1290
Ring(7)	3139	17000
Ring(9)	16799	211528

Table 1 Experimental results

In all cases except one (Dpd) our algorithm needs clearly fewer events than there are reachable markings; in some families (Dac, Dpsyn, Ring) there are far fewer events. A comparison of Dp and Dpsyn is instructive. In Dp, neighbours can concurrently pick and

[1] `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/summaries.tar.gz`

drop forks. Intuitively, this leads to fewer cases in which the condition $\mathit{Ico}_{\mathcal{N}}(e) \subseteq \mathit{Ico}_{\mathcal{N}}(e')$ for cut-off candidates is satisfied. On the other hand, in DPSYN both forks are picked and dropped synchronously, and so no event in $\mathcal{A}_i$ is concurrent to any event in the neighbouring components, making the unfolding procedure much more efficient.

7 Extensions: Divergences and Weights

We conclude the paper by showing that our algorithm can be extended to handle more complex semantics than traces. Indeed, the divergences of the system can be captured by the summaries, as well as the minimal weights of the finite traces from $\mathit{Tr}(\mathbf{A})|_{\Sigma_i}$ when $\mathcal{A}_1 \dots \mathcal{A}_n$ are weighted systems.

7.1 Divergences

We first extend our algorithm so that the summary also contains information about *divergences*. Intuitively, a divergence is a finite trace of the interface after which the system can "remain silent" forever.

▶ **Definition 8.** Let $\mathcal{A}_1, \dots, \mathcal{A}_n$ be LTSs with interface $\mathcal{A}_i$. A *divergence* of $\mathcal{A}_i$ is a finite trace $\sigma \in \mathit{Tr}(\mathcal{A}_i)$ such that $\sigma = \tau_{|\Sigma_i}$ for some infinite trace $\tau \in \mathit{Tr}(\mathbf{A})$. A *divergence-summary* is a pair $(\mathcal{S}_i, D)$, where $\mathcal{S}_i$ is a summary and D is a subset of the states of $\mathcal{S}_i$ such that $\sigma \in \mathit{Tr}(\mathcal{S}_i)$ is a divergence of $\mathcal{A}_i$ iff some realization of σ in $\mathcal{S}_i$ leads to a state of D.

We define the set of divergent conditions of the output of Algorithm 2, and show that it is a correct choice for the set D.

▶ **Definition 9.** Let $\mathcal{N}$ be the output of Algorithm 2. A condition s of $\mathcal{N}_i$ is *divergent* if after termination of the algorithm there is $e \in \mathit{coc}$ with companion e' such that s is concurrent to both e and e'. We denote the set of divergent conditions by DC.

▶ **Theorem 10.** *A finite trace $\sigma \in \mathit{Tr}(\mathcal{S}_i)$ is a divergence of $\mathcal{A}_i$ iff there is a divergent condition s of $\mathcal{N}_i$ such that some realization of σ leads to $[s]_\equiv$. Therefore, $(\mathcal{S}_i, [\mathit{DC}]_\equiv)$ is a divergence-summary.*

The proof of this theorem is given in [5].

7.2 Weights

We now consider weighted systems, e.g parallel compositions of weighted LTS. More formally, a weighted LTS $\mathcal{A}^w = (\mathcal{A}, c)$ consists of an LTS $\mathcal{A} = (\Sigma, S, T, \lambda, s^0)$ and a weight function $c : T \to \mathbb{R}_+$ associating a weight to each transition. A *weighted trace* of $\mathcal{A}^w$ is a pair (σ, w) where $\sigma = a_1 \dots a_k$ is a finite trace of $\mathcal{A}$ and w is the minimal weight among the paths realizing σ, i.e:

$$w = \min_{\substack{s_0 \dots s_k \in S^{k+1}, s_0 = s^0, \\ t_i = (s_{i-1}, s_i) \in T, \lambda(t_i) = a_i}} \sum_{j=1}^{k} c(t_j).$$

We denote by $\mathit{Tr}(\mathcal{A}^w)$ the set of all the weighted traces of $\mathcal{A}^w$. The parallel composition $\mathbf{A}^w = (\mathbf{A}, \mathbf{c}) = \mathcal{A}_1^w \,||_w \cdots ||_w\, \mathcal{A}_n^w$ of the LTS $\mathcal{A}_1^w, \dots, \mathcal{A}_n^w$ is such that $\mathbf{A} = \mathcal{A}_1 || \dots || \mathcal{A}_n$ and the weight of a global transition $\mathbf{t} = (t_1, \dots, t_n)$ is:

$$\mathbf{c}(\mathbf{t}) = \sum_{t_i \neq \star} c_i(t_i).$$

Similarly a *weighted labelled Petri net* is a tuple $\mathcal{N}^w = (\mathcal{N}, c)$ where $\mathcal{N} = (\Sigma, P, T, F, \lambda, M_0)$ is a labelled Petri net and $c : T \to \mathbb{R}_+$ associates weights to transitions. A weighted trace in $\mathcal{N}^w$ is a pair (σ, w) with σ a finite trace of $\mathcal{N}$ and w the minimal weight of an occurrence sequence corresponding to σ, where the weight of an occurrence sequence is the sum of the weights of its transitions. By $Tr(\mathcal{N}^w)$ we denote the set of all the weighted traces of $\mathcal{N}^w$.

The branching processes of $\mathcal{A}_1^w ||_w \dots ||_w \mathcal{A}_n^w$ are defined as weighted labelled Petri nets like in the non-weighted case, where each event is implicitly labelled by an action (as before) and a cost. Given a finite set of weighted traces W we define its restriction to alphabet Σ as

$$W|_\Sigma = \{\, (\sigma, w) : \exists (\sigma', w') \in W, \sigma = \sigma'|_\Sigma \wedge w = \min_{\substack{(\sigma', w') \in W \\ \sigma'|_\Sigma = \sigma}} w' \,\}.$$

As in the non-weighted case we are interested in solving the following summary problem:

▶ **Definition 11** (Weighted summary problem). Given weighted LTSs $\mathcal{A}_1^w, \dots, \mathcal{A}_n^w$ with interface $\mathcal{A}_i^w$, compute a weighted LTS $\mathcal{S}_i^w$ satisfying $Tr(\mathcal{S}_i^w) = Tr(\mathbf{A}^w)|_{\Sigma_i}$, where $\mathbf{A}^w = \mathcal{A}_1^w ||_w \dots ||_w \mathcal{A}_n^w$.

This section aims at showing that the approach to the summary problem proposed in the non-weighted case still works in the weighted one. In other words, $\mathcal{S}_i^w$ can be obtained by computing a finite branching process $\mathcal{N}^w$ of $\mathbf{A}^w$ (using Definition 6 of cut-off and cut-off candidates and Algorithm 2) and then taking the interface projection $\mathcal{N}_i^w$ of $\mathcal{N}^w$ on $\mathcal{A}_i^w$ and folding it. The notion of interface projection needs to be slightly modified to take weights into account. The conditions, events, and arcs of $\mathcal{N}_i^w$ are defined exactly as above, and the weight of an event e of $\mathcal{N}_i^w$ is $c_i(e) = c([e]) - c([e'])$ if the predecessor e' of e in $\mathcal{N}_i^w$ exists and $c_i(e) = c([e])$ else, where c is the weight function of $\mathcal{N}^w$ and $c([e]) = \sum_{e_k \in [e]} c(e_k)$, where $[e]$ is the past of e in the weighted branching process $\mathcal{N}^w$.

▶ **Theorem 12.** *Let $\mathbf{A}^w = \mathcal{A}_1^w ||_w \dots ||_w \mathcal{A}_n^w$ with interface $\mathcal{A}_i^w$. The instance of Algorithm 2 given by Definition 6 terminates and returns a weighted branching process $\mathcal{N}^w$ such that the folding S_i^w of $\mathcal{N}_i^w$ is a weighted summary of $\mathbf{A}^w$.*

The proof of this theorem is given in [5].

8 Conclusions

We have presented the first unfolding-based solution to the summarization problem for trace semantics. The final algorithm is simple, but its correctness proof is surprisingly subtle. We have shown that it can be extended (with minor modifications) to handle divergences and weighted systems.

The algorithm can also be extended to other semantics, including information about failures or completed traces; this material is not contained in the paper because, while laborious, it does not require any new conceptual ideas.

We conjecture that the condition $e' \ll e$ in the definition of cut-off candidate can be replaced by $e' < e$, if at the same time the algorithm is required to add events in a suitable order. Similar ideas have proved successful in the past (see e.g. [6, 16]).

Acknowledgement

Thanks to the anonymous reviewers for their valuable comments.

References

1 P. Baldan, A. Bruni, A. Corradini, B. König, C. Rodríguez, and S. Schwoon. Efficient unfolding of contextual Petri nets. *TCS*, 449(1):2–22, 2012.

2 J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22:161–180, 1996.

3 J-M. Couvreur, S. Grivet, and D. Poitrenaud. Designing a LTL model-checker based on unfolding graphs. In *Proceedings of ICATPN*, pages 123–145, 2000.

4 J. Esparza and K. Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking.* Springer, 2008.

5 J. Esparza, L. Jezequel, and S. Schwoon. Computation of summaries using net unfoldings. Technical report, arXiv:1310.2143, 2013.

6 J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proceedings of TACAS*, pages 87–106, 1996.

7 E. Fabre, A. Benveniste, S. Haar, and C. Jard. Distributed monitoring of concurrent and asynchronous systems. *DEDS*, 15(1):33–84, 2005.

8 E. Fabre and L. Jezequel. Distributed optimal planning: an approach by weighted automata calculus. In *Proceedings of CDC*, pages 211–216, 2009.

9 B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Optimal Zielonka-type construction of deterministic asynchronous automata. In *Proceedings of ICALP*, pages 52–63, 2010.

10 S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proceedings of CAV*, pages 186–196, 1990.

11 D. Harel, O. Kupferman, and M. Y. Vardi. On the complexity of verifying concurrent transition systems. In *Proceedings of CONCUR*, pages 258–272, 1997.

12 S. Hickmott, J. Rintanen, S. Thiébaux, and L. White. Planning via Petri net unfolding. In *Proceedings of IJCAI*, pages 1904–1911, 2007.

13 C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

14 V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings.* PhD thesis, Newcastle University, 2003.

15 V. Khomenko and M. Koutny. LP deadlock checking using partial-order dependencies. In *Proceedings of CONCUR*, pages 410–425, 2000.

16 V. Khomenko, M. Koutny, and W. Vogler. Canonical prefixes of Petri net unfoldings. *Acta Informatica*, 40(2):95–118, 2003.

17 V. Khomenko, A. Madalinski, and A. Yakovlev. Resolution of encoding conflicts by signal insertion and concurrency reduction based on STG unfoldings. In *Proceedings of ACSD*, pages 57–68, 2006.

18 K. McMillan. A technique of state space search based on unfolding. *FMSD*, 6(1):45–65, 1995.

19 M. Mukund and M. Sohoni. Gossiping, asynchronous automata and Zielonka's theorem. Technical Report TCS-94-2, SPIC Science Foundation, 1994.

20 S. Römer. *Theorie und Praxis der Netzentfaltungen als Grundlage für die Verifikation nebenläufiger Systeme.* Phd thesis, TU München, 2000.

21 A. Valmari. Compositionality in state space verification methods. In *Proceedings of ICATPN*, pages 29–56, 1996.

22 G. Winskel. Events, causality and symmetry. *Computer Journal*, 54:42–57, 2011.

23 F. Zaraket, J. Baumgartner, and A. Aziz. Scalable compositional minimization via static analysis. In *Proceedings of ICCAD*, pages 1060–1067, 2005.

24 W. Zielonka. Notes on finite asynchronous automata. *RAIRO - Theoretical Informatics and Applications*, 21(2):99–135, 1987.