Independence Abstractions and Models of Concurrency

Vijay D'Silva[1], Daniel Kroening[2], and Marcelo Sousa[2⋆]

[1] Google Inc., San Francisco
[2] University of Oxford

Abstract. Mathematical representations of concurrent systems rely on two fundamental notions: an atomic unit of behaviour called an event, and a constraint called independence which asserts that the order in which certain events occur does not affect the final configuration of the system. We apply abstract interpretation to study models of concurrency by treating events and independence as abstractions. Events arise as Boolean abstractions of traces. Independence is a parameter to an abstraction that adds certain permutations to a set of sequences of events. Our main result is that several models of concurrent system are a composition of an event abstraction and an independence specification. These models include Mazurkiewicz traces, pomsets, prime event structures, and transition systems with independence. These results establish the first connections between abstraction interpretation and event-based models of concurrency and show that there is a precise sense in which independence is a form of abstraction.

1 Models of Concurrency as Abstractions

Concurrency theory is rich with structures that have been developed for representing and modelling the behaviour of concurrent systems. These include Petri nets [17], process algebra [8], Mazurkiewicz traces [13], partially-ordered multisets (pomsets) [19], various event-based models such as event structures [23], flow event structures [3], and event automata [18], and transition systems augmented with independence information [22]. Research into *comparative concurrency semantics* is concerned with identifying criteria for classifying and comparing these models and constructions that translate between models when possible.

There are several approaches to comparative concurrency semantics. The linear time-branching time spectrum [9] is a classification based on notions of semantic equivalence between processes. The interleaving/non-interleaving classification, is based on whether a model distinguishes between concurrent executions and non-deterministic ones. In an interleaving semantics, the finest representation of a systems execution is a linearly ordered sequence of events and concurrency is understood in terms of the linearizations it induces. Examples of interleaving models include traces, transition systems and synchronization

⋆ Supported by a Google PhD Fellowship.

trees [22]. In the non-interleaving view, the behaviour of a system can be viewed as a partial order on transitions or events. Examples of such models are Petri nets, event based models and pomsets.

A third classification is based on the duality of state and observation, also called an automaton-schedule duality [20]. State-based models such as automata represent a system by states and state changes. An event or schedule-oriented model focuses instead on the points of causal interaction between concurrent events. The concurrency cube of [22] combines these three perspectives in classifying and comparing models of concurrency.

Each representation has distinct mathematical properties and leads to different algorithms for analysis of concurrent systems. In particular, partial order reduction [11, 7] is based on the theory of Mazurkiewicz traces, which is a linear-time, event-based view, while net unfoldings [6] are based on event structures [23], a branching-time, event based model.

The Abstract Interpretation Perspective. Abstract interpretation is a theory for approximation of semantics. A common, practical application of abstract interpretation is static analysis of programs, but the framework has been applied to compare and contrast semantic models [4]. The abstract interpretation approach to comparative semantics is to start with an expressive semantics that describes all the properties of interest in a system and then derive other semantic representations as abstractions. Certain relationships between semantic models then manifest as relationships between abstractions.

The motivation for this work stems from the development of a program analyzer based on non-interleaving semantics [21]. An analyzer that uses an event-based, non-interleaving representation can succinctly represent concurrent schedules that would require an exponential number of interleavings in an interleaving model [6, 14]. On the other hand, event-based analyzers do not include data abstraction: at present, distinct states give rise to distinct events, triggering an explosion of events [21]. The research in this paper is a first step towards composing event-based models with abstract domains.

Contribution and Overview This paper examines different representations of concurrent behaviour as abstractions of a fine-grained semantics of a system. We advance the thesis that the fundamental components of several models of concurrency are parameters to an abstraction functor and the models themselves are abstract domains generated by this functor.

The first component of a concurrency model is an indivisible unit of computation called an event. We define events as Boolean abstractions that satisfy a history condition. The second component of concurrency models is a notion of independence, which dictates when events may fire concurrently. Independence defines an abstraction of a domain of sequences of events. A concurrency model arises as a composition of an event abstraction and an independence specification. We demonstrate that several models of concurrency can be instantiated by our domain functor. This research gives credence to the idea that concurrency itself is a form of abstraction.

2 Order Theory and Abstract Interpretation Primer

Sets and Posets. We denote the subset ordering as $\subseteq$ and strict subset as $\subset$. The *image* of a set $X \subseteq A$ with respect to a relation $R \subseteq A \times C$ is the set $R(X) \hat{=} \{y \in C \mid x \in X \text{ and } (x, y) \in R\}$. The *preimage* of $X \subseteq C$ is $R^{-1}(X)$. The composition of $R \subseteq A \times B$ and $S \subseteq B \times C$ is the relation $R \circ S \hat{=} \{(a, c) \mid (a, b) \in R \text{ and } (b, c) \in S \text{ for some } b \in B\}$.

We assume the notions of a poset and a lattice. A *bounded lattice* has a least element $\bot$ and a greatest element $\top$, called bottom and top, respectively. A lattice L is *complete* if every subset $S \subseteq L$ has a meet $\bigsqcap S$ and a join $\bigsqcup S$. Superscripts and subscripts introduced for disambiguation will always be dropped when they are clear from the context.

Functions. Consider lattices $(L, \sqsubseteq, \sqcap, \sqcup)$ and $(M, \preccurlyeq, \curlywedge, \curlyvee)$. A function $g : L \to M$ is *monotone* if, for all x and y in L, $x \sqsubseteq y$ implies $g(x) \preccurlyeq g(y)$. The function g is a *lattice homomorphism* if it is monotone and satisfies $g(x \sqcap y) = g(x) \curlywedge g(y)$ and $g(x \sqcup y) = g(x) \curlyvee g(y)$ for all x and y. A homomorphism of complete lattices must further commute with arbitrary meets and joins, of Boolean lattices must commute with complements, etc. A homomorphism with respect to some set of lattice operations is one that commutes with those operations. The *De Morgan dual* of a function $f : L \to M$ between Boolean lattices maps x to $\neg f(\neg x)$.

Galois connections and Closures Let $(L, \sqsubseteq)$ and $(M, \preccurlyeq)$ be posets. A *Galois connection* $(L, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \preccurlyeq)$, is a pair of functions $\alpha : L \to M$ and $\gamma : M \to L$ satisfying that for all $x \in L$ and $y \in M$, $\alpha(x) \preccurlyeq y$ exactly if $x \sqsubseteq \gamma(y)$. When the orders involved are clear, we write $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M$. A Galois connection is a *Galois insertion* if α is surjective. A function in $L \to L$ is called an *operator*. The operator f is *extensive* if $x \sqsubseteq f(x)$, *reductive* if $f(x) \sqsubseteq x$ and *idempotent* if $f(f(x)) = f(x)$. An operator is an *upper closure* if it is monotone, idempotent and extensive and is a *lower closure* if it is monotone, idempotent and reductive. A closure is an upper or a lower closure.

Abstract Domains A *domain* in the sense of abstract interpretation, is a complete lattice equipped with monotone functions, called *transformers*, and non-monotone operations called widening and narrowing for enforcing the convergence of an analysis. We do not consider widening and narrowing. We use signatures to compare transformers from different domains.

Fix a signature containing a set of symbols *Sig* with an arity function $ar : Sig \to \mathbb{N}$. A domain $\mathcal{A} = (A, O_A)$ is a complete lattice A and a collection of transformers $f^{\mathcal{A}} : A^{ar(f)} \to A$, for each symbol in *Sig*. For notational simplicitly, the next definition uses unary transformers. A domain $\mathcal{A} = (A, O_A)$ is an *abstraction* of $\mathcal{C} = (C, O_C)$ if there exists a Galois connection $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ such that for all f in *Sig*, $\alpha \circ f^C \sqsubseteq f^A \circ \alpha$. The *best abstract transformer* corresponding to f^C is the function $\alpha \circ f^C \circ \gamma$, which represents the most precise approximation of a single application of f^C. A *Sig*-domain *homomorphism* is

a complete lattice isomorphism that commutes with transformers of the two domains. A *Sig*-domain *isomorphism* is similarly defined.

Definition 1. *Two abstractions* $\mathcal{A} = (A, O_A)$ *and* $\mathcal{B} = (B, O_B)$*, of a concrete domain* $\mathcal{C} = (C, O_C)$*, specified by Galois connections* $(C, \leqslant) \underset{\alpha_{\mathcal{A}}}{\overset{\gamma_{\mathcal{A}}}{\leftrightarrows}} (A, \sqsubseteq_A)$ *and* $(C, \leqslant) \underset{\alpha_{\mathcal{B}}}{\overset{\gamma_{\mathcal{B}}}{\leftrightarrows}} (B, \sqsubseteq_B)$ *are* equivalent *if there exists a domain isomorphism* $h : A \to B$*, satisfying that* $h(\alpha_A(c)) = \alpha_B(c)$ *and* $\gamma_A(a) = \gamma_B(h(a))$*.*

Process Algebra We use process algebra only for notational convenience in this paper, and review it informally here. Let *Act* be a set of actions. The terms in a standard process algebra [2] are generated by the grammar below.

$$P ::= a \;\mid\; P;Q \;\mid\; P + Q \;\mid\; P^* \;\mid\; P \| Q$$

A process P may be an atomic action a, the sequential composition $P;Q$ of two processes, a choice $P + Q$ between two processes, the iteration P^* of a process, or the parallel composition $P \| Q$ of two processes.

3 Events as Abstractions

The contribution of this section is to show that various notions of an event arise as abstract interpretations. We introduce a new domain of transition sequences, the notion of an event abstraction, and a domain functor for constructing event sequences from an event abstraction.

3.1 The Domain of Transition Sequences

Let *Act* be a set of actions and *State* be a set of states. A labelled relation on *State* is a subset of $Rel \mathrel{\hat{=}} State \times Act \times State$. A *transition* (s, a, t) is an element of a labelled relation in which s is the *source*, t is the *target* and a is the *label*. Moreover, s is the *predecessor* of t and t is the *successor* of s.

The most detailed behaviour we consider for a system is sequences of transitions. Transition sequences are not paths because the endpoints of transitions are duplicated. We refer to $(s_0, a_0, t_0), (s_1, a_1, t_1), \ldots, (s_{n-1}, a_{n-1}, t_{n-1})$ as a *transition sequence* that has length n. The empty sequence ε has length 0. The first and last states in the sequence are s_0 and t_{n-1}, while (s_0, a_0, t_0) and (s_{n-1}, a, t_{n-1}) are the first and last transitions. A transition sequence of length n is *consistent* if adjoining target and source states coincide: $t_i = s_{i+1}$ for all $i < n - 1$. An *inconsistent* transition sequence is not consistent. The concatenation of two transition sequences τ and σ is defined using the standard string concatenation, denoted $\tau \cdot \sigma$, and abbreviated to $\tau\sigma$ when no confusion arises. The sequence τ is the prefix of $\tau\sigma$, denoted $\tau \preccurlyeq \sigma$, where $\preccurlyeq$ is the prefix order.

A *labelled transition system* (LTS) $M = (State, Trans)$ consists of a set of states and a *transition relation* $Trans \subseteq Rel$. A transition (s, a, t) is *enabled* in a state s if (s, a, t) is in *Trans*. A transition sequence is *feasible* if it only

contains transitions from *Trans* and is *infeasible* otherwise. A *path* or *history* of length n is a sequence $s_0, a_1, s_1, \ldots, a_{n-1}, s_{n-1}$, which corresponds to a feasible, consistent transition sequence $(s_0, a_1, s_1) \ldots (s_{n-2}, a_{n-1}, s_{n-1})$. Feasibility and consistency are unrelated as there exist transition sequences that infeasible and consistent, and sequences that are feasible and inconsistent.

We recall the main properties of interest for verification, which are reachability of states and the analogous property, firability of transitions. Let $Init_S$ (Fin_S) $\subseteq$ *State* be a set of initial (final) states. An *initial* transition is one that is enabled in a state in $Init_S$. Some final state is *reachable* if there is a path whose first state is in $Init_S$ and last state is in Fin_S. A history is *firable* if it starts in a state in $Init_S$. A transition is *firable* if it is the last transition of a firable history. An action is firable if it is the label of a firable transition.

We introduce a domain of transition sequences, which consists of all possible sets of transition sequences and transformers for extending such sequences. We write Rel^* for the set of finite sequences of transitions. The *lattice of transition sequences* is $(\mathcal{P}(Rel^*), \subseteq)$. The forward and backwards *enabled* transformers $en_\rightarrow, en_\leftarrow : \mathcal{P}(Rel^*) \rightarrow \mathcal{P}(Rel)$ map from a sequence to the transitions enabled either at the end or before the beginning of the sequence.

$$en_\rightarrow(X) \mathrel{\hat{=}} \{(s,b,t) \mid \tau(r,a,s) \in X \text{ and } (s,b,t) \in Trans\}$$
$$en_\leftarrow(X) \mathrel{\hat{=}} \{(r,a,s) \mid (s,b,t)\tau \in X \text{ and } (r,a,s) \in Trans\}$$

The transformers below are defined on $\mathcal{P}(Rel^*) \rightarrow \mathcal{P}(Rel^*)$.

$$st(X) \mathrel{\hat{=}} \left\{\sigma(s,a,t) \mid \sigma \in Rel^*, \tau(r,b,t) \in X\right\}$$
$$tr(X) \mathrel{\hat{=}} \left\{\sigma(s,a,t) \mid \sigma \in Rel^*, \tau(s,a,t) \in X\right\}$$
$$ext_\rightarrow(X) \mathrel{\hat{=}} \{\tau(s,b,t) \mid \tau \in X, (s,b,t) \in en_\rightarrow(\{\tau\})\}$$
$$ext_\leftarrow(X) \mathrel{\hat{=}} \{(r,a,s)\tau \mid \tau \in X, (r,a,s) \in en_\leftarrow(\{\tau\})\}$$

The *state closure* transformer st extends X with all sequences that have the same terminal state as some sequence in X. The *transition closure* transformer tr extends X with all sequences that have the same terminal transition as some sequence in X. These two closures serve two purposes: they allow for states and transitions to be viewed as abstractions of sequences, and they allow for reconstructing a transition system representation of a system from a domain encoding its behaviour. It is easy to reconstruct a transition system from transition sequences, but the task is not as easy in an abstraction.

The *forward extension* transformer $ext_\rightarrow$ extends the end of a sequence with transitions that respect the transition relation and the *backward extension* transformer $ext_\leftarrow$ extends a sequence backwards in a similar manner. The transformers above are existential in that they rely on the existence of a sequence in their argument or in *Rel*. These transformers have universal variants defined in the standard way by complementation.

$$\widetilde{st} \mathrel{\hat{=}} \neg \circ st \circ \neg \quad \widetilde{tr} \mathrel{\hat{=}} \neg \circ tr \circ \neg \quad \widetilde{ext}_\rightarrow \mathrel{\hat{=}} \neg \circ ext_\rightarrow \circ \neg \quad \widetilde{ext}_\leftarrow \mathrel{\hat{=}} \neg \circ ext_\leftarrow \circ \neg$$

Proposition 1. *The transformers of the transition sequence domain are monotone and satisfy the following properties.*

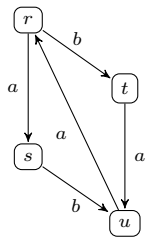
1. *There is a Galois connection between* $ext_{\rightarrow}$ *and* $\widetilde{ext}_{\leftarrow}$.
2. *There is a Galois connection between* $ext_{\leftarrow}$ *and* $\widetilde{ext}_{\rightarrow}$.
3. *The transformers st and tr are upper closure operators.*

Let $Init_T$ and Fin_T be sets of initial and final transitions. The standard characterizations of reachability of final states from initial states lifts below to histories as the following fixed point: $\mathsf{lfp}\ x.\ Init_T \cup ext_{\rightarrow}(x)$.

3.2 Events as Abstractions

We illustrate different notions of an event that may arise from a single system.

Example 1. The LTS below represents the operational semantics of the term $((\mathrm{a}\|\mathrm{b}) \cdot \mathrm{a})^*$. The initial state of this system is r.



We consider five different notions of an event. An event as a *firable history* corresponds to treating nodes in a computation tree as events. This notion is history- and interleaving dependent, so $(r, a, s)(s, b, u)$ and (r, b, t) represent different events. A *prime event*, which we name after the notion used in prime event structures, is history-dependent but *interleaving-independent*. An event respects history but disregards concurrent scheduling differences.

Thus, the sequences $(r, a, s)(s, b, u)$ and (r, b, t) represent the same event because (r, a, s) does not *causally precede* (s, b, u). However, the sequences (r, b, t) and $(r, a, s)(s, b, u)(u, a, r)(r, b, t)$ represent different events because the longer sequence has a history that is not due to concurrent interleaving alone.

Viewing transitions as events leads to a history-independent, interleaving-dependent notion. An *independent transition*, is an equivalence class of transitions that does not distinguish between transitions that arise due to scheduling differences. For example (r, b, t) and (s, b, u) are equivalent, and (r, a, s) and (t, a, u) are equivalent, leading to three events in this example. A *Mazurkiewicz event* is an action, so this system only has two Mazurkiewicz events. ◁

We formalize events as Boolean abstractions of transition sequences. The prefix condition in Definition 2 is used later to construct event sequences.

Definition 2. *Let* $M = (State, Trans)$ *be a* LTS *with transition sequences* Rel^*. *An event abstraction* Ev*, parameterized by a set Event, satisfies these conditions.*

1. *There is a Galois connection* $(\mathcal{P}(Rel^*), \subseteq) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} (\mathcal{P}(Event), \subseteq)$.
2. *The concretization is a homomorphism with respect to* $\bigcup, \bigcap$ *and complement.*

3. If $\alpha_{\mathsf{Ev}}(\{\sigma\}) \neq \emptyset$ *and* $\tau \preccurlyeq \sigma$*, for non-empty* τ*, then* $\alpha_{\mathsf{Ev}}(\{\tau\}) \neq \emptyset$*.*

The event abstraction is total if $\gamma_{\mathsf{Ev}}(\{e\}) \neq \gamma_{\mathsf{Ev}}(\emptyset)$ *for every event* e*.*

The first condition above asserts that events are abstractions of sequences of transitions. The second condition ensures that a set of transition sequences of interest can be partitioned into the events they generate and that a concrete transition sequence maps to at most one event. The concretization conditions are weaker than the requirement that γ_{Ev} is a homomorphism of Boolean algebras because the condition $\gamma_{\mathsf{Ev}}(\emptyset) = \emptyset$ is missing. This is because a transition sequence τ may not map to an event. In an event abstraction in which $\alpha_{\mathsf{Ev}}(\{\tau\}) \neq \emptyset$ exactly if τ is a firable history, $\gamma_{\mathsf{Ev}}(\emptyset)$ will contain exactly the infeasible sequences.

Lemma 1. *For every transition sequence* τ*,* $\alpha_{\mathsf{Ev}}(\{\tau\}) \subseteq \{e\}$ *for some event* e*.*

We give an alternative characterization of events by equivalence relations, as this formulation is sometimes convenient to use. Recall that a *partial equivalence relation* (PER) on a set S is a symmetric, transitive, binary relation on S. Unlike an equivalence relation, a PER is not reflexive. The quotient $S/\equiv$ contains of equivalence classes of S with respect to $\equiv$ and $[s]_{\equiv}$ is the equivalence class of s. As $\equiv$ is not reflexive, $[s]_{\equiv}$ may not be defined.

A PER on transition sequences is *prefix-closed* if whenever $\tau \equiv \sigma$ and there exists τ' such that $\tau' \preccurlyeq \tau$, there exists σ' such that $\tau' \equiv \sigma'$. Note that there is no requirement that σ' be a prefix of σ. A PER $\equiv$ generates a lattice $(\mathcal{P}(Rel^*/\equiv), \subseteq)$ of equivalence classes of sequences. This lattice is an abstraction of transition sequences as given by the functions below.

$$\alpha_{\equiv}(X) \mathrel{\hat{=}} \{[s] \mid s \in X, s \equiv s\} \qquad \gamma_{\equiv}(X) \mathrel{\hat{=}} \left(\bigcup X\right) \cup \{s \mid s \not\equiv s\}$$

The abstraction replaces sequences by equivalence classes while the concretization is the union the contents of equivalence classes and those sequences on which equivalence is not defined. Lemma 2 shows that prefix-closed PERs and total event abstractions are equivalent ways of defining the same concept.

Lemma 2. *Every total event abstraction is isomorphic to the abstraction generated by a prefix-closed* PER*.*

Example 2. A PER for the prime event abstraction in Example 1 is the least PER over firable histories satisfying these constraints: $\varepsilon \equiv \varepsilon$, $\tau \equiv \tau$ for all firable histories τ and for all $\tau \equiv \sigma$, $\tau(r,a,s) \equiv \sigma(r,b,t)(t,a,u)$, $\tau(r,b,t) \equiv \sigma(r,a,s)(s,b,u)$. Note that the concatenations above must produce firable histories. ◁

Since the event abstractions operates over a powerset lattice (of events or equivalence classes of transition sequences), we can relate several event abstractions. In Figure 1, we illustrate the relationship between the several event abstractions described in Example 1. The *No Events* abstraction simply abstracts any transition sequence to the empty set.

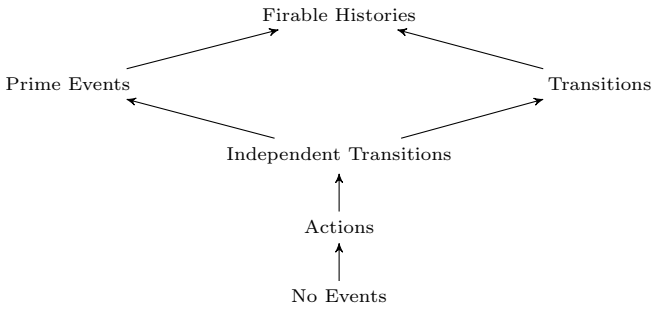


Fig. 1. Relationship between event abstractions.

3.3 The Event Sequence Domain Functor

Events represent an abstract unit of behaviour. The evolution of a system over time has different representations, including as partial orders on events [23] or relations between sets of events [10]. We model behaviour as sequences of events. We introduce a domain functor for generating a domain of event sequences from a domain of events. This domain exploits the prefix-closure condition of Definition 2 to derive event sequences from transition sequences.

Example 3. Consider the sequence $\sigma \mathrel{\hat{=}} (r,a,s)(s,b,u)(u,a,r)(r,b,t)$ from the system in Example 1. When firable histories define events, each prefix of σ of length n represents a unique event e_n. In particular $\alpha_{\mathsf{Ev}}(\{\sigma\}) = \{e_4\}$. The sequence σ can be viewed in terms of the events fired along it: $e_1e_2e_3e_4$. $\lhd$

The functor $\mathsf{ESeq}(\cdot)$ maps an event abstraction $\mathcal{P}(Rel^*) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} \mathcal{P}(Event)$ to the event sequence abstraction $\mathsf{ESeq}(\mathsf{Ev})$ defined below. When clear from context, we write ESeq for $\mathsf{ESeq}(\mathsf{Ev})$. The lattice of *event sequences* is $(\mathcal{P}(Event^*), \subseteq)$. The abstraction and concretization maps are given below.

$$\alpha_{\mathsf{ESeq}} : \mathcal{P}(Rel^*) \to \mathcal{P}(Event^*)$$

$$\alpha_{\mathsf{ESeq}}(\{\sigma\}) \mathrel{\hat{=}} \begin{cases} \{\varepsilon\} \text{ if } \sigma = \varepsilon \\ \emptyset \text{ if } \alpha_{\mathsf{Ev}}(\{\sigma\}) = \emptyset \\ \{\tau' \cdot e \mid e \in \alpha_{\mathsf{Ev}}(\{\sigma\}), \tau' \in \alpha_{\mathsf{ESeq}}(\{\sigma'\}), \sigma = \sigma'(s,b,t)\}\,, \text{ o.w.} \end{cases}$$

$$\gamma_{\mathsf{ESeq}} : \mathcal{P}(Event^*) \to \mathcal{P}(Rel^*)$$

$$\gamma_{\mathsf{ESeq}}(\{\sigma\}) \mathrel{\hat{=}} \begin{cases} \{\varepsilon\} \text{ if } \sigma = \varepsilon \\ \{\tau(s,b,t) \mid \tau(s,b,t) \in \gamma_{\mathsf{Ev}}(e), \tau \in \gamma_{\mathsf{ESeq}}(\{\sigma'\}), \sigma = \sigma' e\} \text{ o.w.} \end{cases}$$

$$\alpha_{\mathsf{ESeq}}(X) \mathrel{\hat{=}} \bigcup_{\tau \in X} \alpha_{\mathsf{ESeq}}(\{\tau\}) \qquad \gamma_{\mathsf{ESeq}}(X) \mathrel{\hat{=}} \bigcup_{\sigma \in X} \gamma_{\mathsf{ESeq}}(\{\sigma\})$$

The concretization map goes from a sequence of events to those transition sequences whose prefix closure generates exactly the same event sequence. The enabled event transformers $en^{\mathsf{ESeq}}_{\rightarrow}, en^{\mathsf{ESeq}}_{\leftarrow} : \mathcal{P}(\mathit{Event}^*) \rightarrow \mathcal{P}(\mathit{Event})$ map an event sequence to events enabled at the beginning or end.

$$en^{\mathsf{ESeq}}_{\rightarrow}(X) \mathrel{\hat{=}} \bigcup \{\alpha_{\mathsf{Ev}}(\{\sigma(s,b,t)\}) \mid (s,b,t) \in en_{\rightarrow}(\{\sigma\}), \sigma \in \gamma_{\mathsf{ESeq}}(X)\}$$
$$en^{\mathsf{ESeq}}_{\leftarrow}(X) \mathrel{\hat{=}} \bigcup \{\alpha_{\mathsf{Ev}}(\{(r,a,s)\}) \mid (r,a,s) \in en_{\leftarrow}(\gamma_{\mathsf{ESeq}}(X))\}$$

Forward enabledness constructs the enabled event by concatenating it with its history, but backwards enabledness is simply the event abstraction of a transition. The extension transformers concatenate a sequence with an enabled event.

$$ext^{\mathsf{ESeq}}_{\rightarrow}(X) \mathrel{\hat{=}} \left\{\sigma e \mid \sigma \in X, e \in en^{\mathsf{ESeq}}_{\rightarrow}(\{\sigma\})\right\}$$
$$ext^{\mathsf{ESeq}}_{\leftarrow}(X) \mathrel{\hat{=}} \left\{e\sigma \mid \sigma \in X, e \in en^{\mathsf{ESeq}}_{\leftarrow}(\{\sigma\})\right\}$$

We define a few event abstractions below.

Histories The set of histories *Hist* consists of feasible, consistent transition sequences. The event abstraction Hist maps every history to itself and ignores other sequences. The least fixed point providing firable histories when evaluated on event sequences will concretize to the set of firable histories, representing the *unfoldings* of a transition system.

Transitions The domain Tr uses transitions in *Rel* as events with maps $\alpha_{\mathsf{Tr}}(X) = \{(r,a,s) \mid \sigma(r,a,s) \in X\})$ and $\gamma_{\mathsf{Tr}}(Y) = \{\sigma(r,a,s) \mid (r,a,s) \in Y\}$. The domain $\mathsf{ESeq}(\mathsf{Tr})$ of event sequences is equivalent to the domain of transition sequences. The use of prefixes in α_{ESeq} and γ_{ESeq} is necessary for this equivalence to arise. A simplistic lifting of events to event sequences that ignored prefix information would not lead to this equivalence.

Actions Consider *Act* to be the set of events and Act as the domain with $\alpha_{\mathsf{Act}}(X) \mathrel{\hat{=}} \{a \mid \sigma(r,a,s) \in X\}$, and $\gamma_{\mathsf{Act}}(Y) \mathrel{\hat{=}} \left\{\sigma(r,a,s) \in \mathit{Rel}^* \mid a \in Y\right\}$. The event sequences generated by the domain functor correspond to the language abstraction of the system, where only the sequence of labels is retained and not the underlying states. The least fixed point of firable sequences generates the *Hoare language* for a system [22].

4 Independence as an Abstraction

Independence information is used in a model of concurrency to distinguish the situation in which an event precedes another due to a scheduling choice from that in which the precedence is due to causal dependence. The contribution of this section is a function modelling independence and a functor for generating an abstraction from it.

Example 4. This example shows how concurrent behaviours beyond the scope of Mazurkiewicz traces and prime event structures fit into our framework. We represent which events may fire concurrently by a function *Ind* from event sequences to sets of sets of events.

Consider a system with three events a, b, c in which every two events may occur simultaneously but all three may not. This situation, called a *ternary conflict*, can be modelled by a function $Ind_1(\varepsilon) = \{\{a,b\},\{a,c\},\{b,c\},\{a\},\{b\},\{c\}\}$ expressing which events may fire concurrently.

Suppose instead that all three events may fire but only a and c may fire concurrently, we have a *binary conflict* between a and b and between b and c. Further, if a and b can fire concurrently once c has occurred, the conflict is *transient* or is said to be *resolved* by c. To model this situation, consider $Ind_2(\varepsilon) = \{\{a,c\},\{a\},\{b\},\{c\}\}$, encoding the initial conflict, and $Ind(c) = \{\{b,c\},\{a\},\{b\},\{c\}\}$, encoding the situation after c fires. $\lhd$

Definition 3. *An independence function* $Ind : Event^* \to \mathcal{P}(\mathcal{P}(Event))$ *from event sequences to the set of sets of events that may fire independently after that sequence is one that satisfies the following conditions.*

1. *For all* σ *and* $e \in Event$,$\{e\} \in Ind(\sigma)$.
2. *For all* σ *and* $X \in Ind(\sigma)$ *if* $Y \subseteq X$, *then* $Y \in Ind(\sigma)$ *must hold.*
3. *For all* σ, $X \in Ind(\sigma)$, $e \in X$, *there is* $Y \in Ind(\sigma e)$ *such that* $X \setminus \{e\} \subseteq Y$.

The first condition expresses that a singleton set of events is an independent set irrespective of the current event sequence. The second condition states that if a set of events can occur concurrently, every subset of those events can also occur concurrently. Thus, we exclude for this paper models based on distributed protocols that require a specific number of participants. The third condition expresses that if a set of events can potentially fire independently, firing some of them will not disable the others. Note that these events may not be enabled at the particular event sequence as the independence function only represents a may fire concurrently relation. Futher, these events may still be disabled by an event external to the set. The notion of independence above is a functional representation of the local independence of [12] and generalizes many notions of independence in the literature.

Though independence has syntactic similarities to enabledness, it is fundamentally different. Enabledness is a property of a given system while independence may be viewed either as a property of a system or just of actions. In the context of programs, if atomic statements are treated as events, enabledness dictates which statements of the program may fire but independence dictates which statements of the programming language may execute concurrently. The next definition defines an equivalence on sequences based on independence. Let $perm(S)$ represent all permutations of a finite set S.

Definition 4. *The independence equivalence* $\equiv_{Ind}$ *generated by* $Ind : Event^* \to \mathcal{P}(\mathcal{P}(Event))$ *is the least (total) equivalence relation satisfying these conditions.*

1. *For all* σ, $X \in Ind(\sigma)$ *and* $\tau_1, \tau_2 \in perm(X)$, $\sigma\tau_1 \equiv_{Ind} \sigma\tau_2$.
2. *If* $\sigma_1 \equiv_{Ind} \sigma_2$, *then for all* τ, $\sigma_1\tau \equiv_{Ind} \sigma_2\tau$.

We introduce an independence domain functor Ind that generates a domain $\mathsf{Ind}(\mathsf{Ev}, Ind)$ given an event abstraction and an independence function. The lattice of independent sequences $(\mathcal{P}(Event^*/\equiv_{Ind}), \subseteq)$ is the powerset of equivalence classes of independent traces, which abstracts $\mathcal{P}(Event^*)$.

$$\alpha_{\mathsf{Ind}}(X) \hat{=} \{[\sigma]_{\equiv_{Ind}} \mid \sigma \in X\} \qquad \gamma_{\mathsf{Ind}}(X) \hat{=} \bigcup X$$

The enabledness transformer on the domain of independent sequences is not a simple lifting because it considers contiguous sequences of independent events that may occur in the future of a sequence and allows for them to fire at the beginning.

Example 5. Suppose a system has three events a, b, c, which are permitted to fire concurrently. Suppose the only trace in the system is abc. As adjoining events are independent, the equivalence class of abc under independence contain all permutations of the events. However, the only event enabled by the concrete system is a, so the only abstract event that would be enabled is $[a]$. To make the concurrently enabled events explicitly visible, we need that $[a]$, $[b]$ and $[c]$ are all enabled in the abstract before any have fired. ◁

We use a helper function $iext_{\rightarrow} : Event^* \rightarrow \mathcal{P}(Event^*)$ which maps a sequence to possible extension consisting only of independent events. The abstract enabledness function $en^{\mathsf{Ind}}_{\rightarrow} : \mathcal{P}(Event^*/\equiv_{Ind}) \rightarrow \mathcal{P}(Event)$ is also given below.

$$\begin{aligned}
iext_{\rightarrow}(\sigma) &\hat{=} \mathsf{lfp}X.\, en^{\mathsf{ESeq}}_{\rightarrow}(\{\sigma\}) \cup \big\{\tau e \mid \tau \in X, e \in en^{\mathsf{ESeq}}_{\rightarrow}(\{\sigma\tau\}),\\
&\qquad\qquad \text{for some } Y \in Ind(\sigma) \text{ and } \theta \in perm(Y), \tau e \preccurlyeq \theta\big\}\\
en^{\mathsf{Ind}}_{\rightarrow}(X) &\hat{=} \{e \mid \text{for some } [\sigma]_{\equiv_{Ind}} \in X, \tau \in iext_{\rightarrow}(\sigma), e\theta \in [\tau]_{\equiv_{Ind}}\}\\
ext^{\mathsf{Ind}}_{\rightarrow}(X) &\hat{=} \big\{[\sigma e]_{\equiv_{Ind}} \mid e \in en^{\mathsf{Ind}}_{\rightarrow}(\{[\sigma]_{\equiv_{Ind}}\}), [\sigma]_{\equiv_{Ind}} \in X\big\}
\end{aligned}$$

Example 6. Revisit the system in Ex. 5. $Ind(\varepsilon) = \mathcal{P}(\{a, b, c\})$, with the function on event sequences satisfying the required constraints. The steps in the fixed point computation of $iext_{\rightarrow}(\varepsilon)$ are $\{a\}$, due to the initial enabled event, then $\{a, ab\}$, then $\{a, ab, abc\}$. The set $en^{\mathsf{Ind}}_{\rightarrow}(\{\varepsilon\}) = \{a, b, c\}$. ◁

Binary Independence Relations Independence in the theory of Mazurkiewicz traces [13], is a irreflexive, symmetric, binary relation on events. The relation need not be transitive. This instance is of particular importance as it arises in several models of concurrency, namely trace theory, prime event structures with binary conflict and transition systems with independence.

A binary independence relation I generates an independence function that maps every sequence to singleton sets and the sets that contain pairwise independent events.

$$Ind_I(\sigma) \hat{=} \{Y \subseteq Event \mid |Y| = 1 \text{ or for every } e, e' \in Y.\ e\ I\ e'\}$$

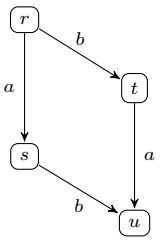
Lemma 3. *Ind_I is an independence function.*

This independence function is *static* as it yields the same independent sets of events irrespectively of the event sequence. Thus, it is clear that a binary independence relation is not sufficient to model ternary or transient conflicts.

Independence Abstraction and Reduction There is an important distinction between the independence abstraction in this work and the notion of independence already inherent in a semantic model of a concurrent system.

Our independence abstraction extends a transition system so that its behaviour is modelled by a particular model of concurrency with respect to a specified event abstraction and independence function. However, there is a set of valid independence functions associated per model of concurrency. A given LTS has several representations within a model of concurrency, dependending on the event abstraction and the associated independence function. We now illustrate how the event abstraction defines a basic independence function. Using such independence functions we obtain complete abstractions in that our abstraction would not add any behaviour.

Example 7. Consider the transition system in the Figure below. The initial state of this system is r.



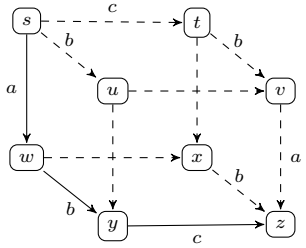
This LTS can be a representation of the term $(a \cdot b) + (b \cdot a)$ or $a \parallel b$. In the former, the event abstraction is represented by a set composed of four events: $\{a_1, a_2, b_1, b_2\}$, where t_i represents the ocurrence of the transition t in a path at the i-th position of the path. In this case, the only independence function that yields a complete abstraction is the one that maps any event sequence to singleton sets as two events are never concurrently enabled.

Declaring the events a_1 and b_1 independent will abstract the LTS to the term $(a \parallel b) + (a \cdot b) + (b \cdot a)$. In the case where the LTS is meant to represent the term $a \parallel b$, the event abstraction is represented by a set composed of two events:$\{a, b\}$. In this case, the independence function is constrained to declare the event a independent with the event b. ◁

This observation on completeness is the foundation for algorithmic techniques such as partial order reduction or model checking with representatives [16]. In that view, the equivalence classes generated by an independence function can be used to construct a reduced LTS from the representatives of these equivalent classes. The resulting reduced LTS is complete for certain properties of interest such as deadlock detection. In that case, the independence function is used a reduction. An important question is whether it exists a *largest* independence function, i.e. the valid independence function that generates the smallest number of equivalence classes and if we can devise pratical methods to compute it. Our independence abstraction is complete in the following sense:

Let M be a concrete LTS, M_R the reduced LTS generated by some independence function Ind, and N the abstract LTS generated by the same independence function. It is the case that both M and N have the same set of traces.

Example 8. Consider the LTS in Example 5 that only has one trace abc.



This LTS is depicted in the figure as the solid path from state s to state z. The history preserving event abstraction generates a set with three events $\{a, b, c\}$. The independence abstraction induced by the independence function in Example 5 generates the equivalence classes of traces $\{\{abc, bac, acb\}\}$.

The abstract LTS is the dashed completion of the trace into the cube in the figure. Observe that the initial LTS is a reduction of the cube LTS which is generated by the same independence function and a choice of abc as the representative of the equivalence class. ◁

5 Instances of Independence Abstractions

The contribution of this section is to demonstrate that models of concurrency arise as compositions of an event abstraction and an independence function. For the remainder of this section, we define independence as a relation between a set of events *Event*.

We now describe the abstraction of this independence function in three models of concurrency: Mazurkiewicz traces, prime event structures and transition systems with independence. The case of asymmetric conflict is analogous as the same independence function is valid for an asymmetric independence relation.

Mazurkiewicz Abstraction and Pomsets We are immediately in the setting of trace theory if we consider the set of events *Event* as the alphabet and the independence relation as an irreflexive and symmetric relation over events. In this case, the abstraction is simply the Parikh equivalence where equivalent sequences are permutations of words in the language theoretic sense.

Example 9. Consider the event sequence abc and the independence relation as the symmetric closure of the relation $a\ I\ b$ and $b\ I\ c$. The trace abstraction will generate the equivalence classes $\{\{abc, bac, acb\}\}$. ◁

Unlike the Mazurkiewicz trace abstraction, there is no requirement to abstract actions as events, so multiple events may have the same label. A sequence of events $e_1, \ldots e_n$ represents a linear order $e_1 \leq \cdots \leq e_n$. An equivalence class $[\sigma]_{\equiv_{Ind}}$, represents the partial order generated by the intersection of all linear orders in that class. Since multiple elements of the partial order may have the

same label, this representation of the equivalence class is called a *partially ordered multiset* or pomset [19].

Prime Event Structures A prime event structure (PES) is a tuple $\mathcal{E} \hat{=} \langle \mathit{Event}, <, \# \rangle$ where $< \subseteq \mathit{Event} \times \mathit{Event}$ is a strict partial order on Event, called *causality relation*, and $\# \subseteq \mathit{Event} \times \mathit{Event}$ is a symmetric, irreflexive *conflict relation*, satisfying

- for all $e \in \mathit{Event}$, the *causes* of e, $\lceil e \rceil := \{e' \in \mathit{Event} : e' < e\}$, is a finite set
- for all $e, e', e'' \in \mathit{Event}$, if $e \# e'$ and $e' < e''$, then $e \# e''$

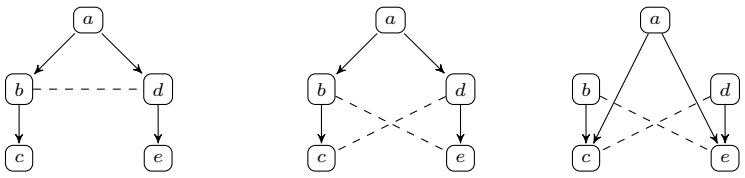
The central concept in a PES is that of a configuration. A *configuration* of $\mathcal{E}$ is any finite set $C \subseteq \mathit{Event}$ satisfying:

- (causally closed) for all $e \in C$ we have $\lceil e \rceil \subseteq C$;
- (conflict free) for all $e, e' \in C$, it holds that $\neg(e \# e')$.

We denote by $\mathit{Conf}(\mathcal{E})$ the set of configurations of $\mathcal{E}$.

We consider the prime event structures where the configurations represent the equivalence classes of the independence abstraction. It is straightforward to see that given an event sequence σ, the independence abstraction will generate an equivalence class composed of permutations of σ and that the intersection of each event sequence in the equivalence class (when seen as a total order) generates a partial order. This partial order is in fact a representation of a Mazurkiewicz trace. Thus, when the conflict relation is generated by the complement of the independence relation, there is a bijection between Mazurkiewicz traces and configurations of a prime event structure.

Example 10. The prime event structure below (on the left) represents the partial semantics of the term $a \cdot ((b \cdot c) + (d \cdot e))$. In the representation of event structures, two events are in conflict if there is a dashed line between them. Also, for the sake of clarity, we only represent immediate conflicts. Two events e, e' are in *immediate conflict*, $e \#^i e'$, iff $e \# e'$ and both $\lceil e \rceil \cup [e']$ and $[e] \cup \lceil e' \rceil$ are configurations.



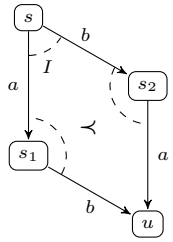
Note that the event b is in conflict with every sucessor of d, and vice versa. Also, note that if $b \; I \; e$, then the same prime event structure is a representation of the system. For the remainder of this example, we consider that b is not independent with sucessors of d and vice versa. The independence abstraction that arises by considering $b \; I \; d$ over this prime event structure amounts to removing the conflict between b and d (prime event structure in the center).

However, note that the independence abstraction does not simply amount to removing conflicts. In particular, the independence abstraction over the prime event structure in the center that considers $a\ I\ b$ and $a\ I\ d$ removes the causality between those events (prime event structure in the right). $\lhd$

Transition Systems with Independence We now study the independence abstraction in transition systems with independence [22]. This model of concurrency has the characteristic of defining events as a derived concept, in particular as equivalence classes of transitions generated by an independence relation on transitions. Thus, this independence relation is used in the event abstraction directly and used in the independence abstraction by lifting the relation over the generated events.

Definition 5. *A labelled transition system with independence (TSI) is a structure $T = (M_T, I_T)$ where M_T is a labelled transition system and $I_T \subseteq Rel \times Rel$ is an* independence relation*, an* irreflexive, symmetric *binary relation such that, using $\prec$ to denote the following binary relation on transitions*

$$
\begin{aligned}
(s,a,s_1) \prec (s_2,a,u) \iff\ & \\
& \exists b \in Act.\ (s,a,s_1)\ I_T\ (s,b,s_2)\ \textit{and} \\
& \qquad (s,a,s_1)\ I_T\ (s_1,b,u)\ \textit{and} \\
& \qquad (s,b,s_2)\ I_T\ (s_2,a,u)
\end{aligned}
$$

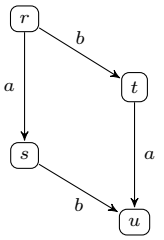


and $\sim$ the least equivalence *relation on transitions which includes $\prec$, we have*

T_1. $(s,a,s_1) \prec (s,a,s_2)$ *implies* $s_1 = s_2$*;*
T_2. $(s,a,s_1)\ I_T\ (s,b,s_2)$ *implies* $\exists u.\ (s,a,s_1)\ I_T\ (s_1,b,u)$ *and* $(s,b,s_2)\ I_T\ (s_2,a,u)$*;*
T_3. $(s,a,s_1)\ I_T\ (s_1,b,u)$ *implies* $\exists s_2.\ (s,a,s_1)\ I_T\ (s,b,s_2)$ *and* $(s,b,s_2)\ I_T\ (s_2,a,u)$*;*
T_4. $(s,a,s_1) \sim (s_2,a,u)\ I_T\ (w,b,w')$ *implies* $(s,a,s_1)\ I_T\ (w,b,w')$.

The event abstraction is generated by the PER $\sim$. Intuitively, an *event* corresponds to a set of transitions associated with the same action. These transitions are equivalent with respect to some property captured by the independence relation. Observe that $\prec$ is not a partial order.

Example 11. Recall the transition system of Example 7 bellow that represents the term $(a \cdot b) + (b \cdot a)$. The initial state of this system is r.



In this initial case, the independence relation is empty as the TSI represents the mutual exclusion between actions a and b. The event abstraction generates one event per transition and the independence relation on events is also the empty relation. Thus, the set of event sequences of this system is $\{\varepsilon, a, b, ab', ba'\}$ where the x' denotes the second transition of label x. If we consider the case where transitions labelled by action a are independent, we obtain the same event and independence abstraction.

Finally, the case where the independence matches to one in Definition 5, the system represents the operational semantics of the term $a \parallel b$. In this case, the event abstraction generates a set with two events a and b, and the independence relation on events considers these events independent. ◁

Example 12. The independence abstraction for TSI is a completion of the underlying transition system with respect to the concurrency diamonds specified by the independence relation. Consider the following TSIs up to the dashed arrows.

The transition system on the left represents the term $a + b$ while the one on the right represents the term $a \cdot b$. If we consider the transition associated with action a independent with the transition associated with b, these systems are not TSIs as they do not satisfy, respectively, $T2$ and $T3$. The structures $T = (M_T, I_T)$ where M_T is a labelled transition system and $I_T \subseteq \mathit{Rel} \times \mathit{Rel}$ is a *symmetric* binary relation are called pre-transition systems with independence. The independence abstraction extends a pre-transition system with independence into the minimal valid TSI which correspond to extending the pre-TSIs with the dashed arrows. It is clear that these completions are analogous to the ones described in the prime event structure case: the left completion corresponds to removing the conflict between events a and b, and the right completion corresponds to removing the causality between events a and b. ◁

Beyond binary independence. In order to express ternary conflict as in the example 4, we consider ternary independence relations $\subseteq \mathit{Event} \times \sigma \times \mathit{Event}$. Lifting the independence function defined previously is straightforward.

For more expressive models able to model more refined behaviour such as n-ary conflicts we assume that the independence function Ind is provided as input.

It is known that models of computation that can represent such behaviours lead to redundancies in the representation when one uses models that are more suited for binary, symmetric independence relations such as prime event structures [10].

6 Related Work

There is a significant body of work devoted to comparing and translating between models for concurrency. For example, see [15] for a uniform approach to compare net classes, [1] for a survey on process algebras, [5] for a comparison and translations between Mazurkiewicz traces and other models, [22] for a categorical study between transition systems with independence and event structures and [10] for *configuration structures*, a general characterization of event models where the notion of state is represented with a set of events, called *configuration*, that respects certain axioms.

The observation that independence is an abstraction is implicit in the original paper [22] that introduces transition systems with independence. In particular, this observation stems from the realization that the state quotient abstraction over a TSI does not necessarily produces a (deterministic) TSI and a completion procedure over a more specialized domain is provided. We have evidence from the lack of methods that combine data abstractions with partial order methods that the phenomenon of counter-intuitive interaction between standard abstractions and independence is important and understudied.

Furthermore, although the idea that events as a derived notion and independence are related, we are not aware of an in-depth study or application of abstraction interpretation theory that precisely formalizes the idea of *independence as abstraction* and provides a systematic separation between these concepts.

Conclusion In this paper, we examined the notions of event and independence, two distinct but related aspects in the representation of concurrent behaviour, as abstractions of a fine-grained semantics of a system. We showed that various notions of systems behaviour arise as abstract interpretations where events are Boolean abstractions of transition sequences. Futhermore, we showed how this event abstraction defines an independent domain functor over the domain of event sequences. Thus, we precisely formalize *independence as an abstraction* and described known models of concurrency such as Mazurkiewicz traces, Winskel prime event structures and transition systems with independence using our framework.

References

1. Baeten, J.C.M.: A brief history of process algebra. Theor. Comput. Sci. 335(2-3), 131–146 (May 2005)
2. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)

3. Boudol, G.: Flow event structures and flow nets. In: Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings. pp. 62–95 (1990)
4. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science 277(1-2), 47–103 (Apr 2002)
5. Diekert, V.: The Book of Traces. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1995)
6. Esparza, J., Heljanko, K.: Unfoldings – A Partial-Order Approach to Model Checking. EATCS Monographs in Theoretical Computer Science, Springer (2008)
7. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of Programming Languages (POPL). pp. 110–121. ACM (2005)
8. Fokkink, W.: Introduction to Process Algebra. Springer (2000)
9. van Glabbeek, R.: The linear time – branching time spectrum I. the semantics of concrete, sequential processes. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 3 – 99. Elsevier (2001)
10. van Glabbeek, R.J., Plotkin, G.D.: Configuration structures, event structures and petri nets. Theor. Comput. Sci. 410(41), 4111–4159 (2009)
11. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer (1996)
12. Kuske, D., Morin, R.: Pomsets for local trace languages. J. Automata Languages and Combinatorics 7(2), 187–224 (Nov 2001)
13. Mazurkiewicz, A.: Trace theory. In: Petri Nets: Applications and Relationships to Other Models of Concurrency, LNCS, vol. 255, pp. 278–324. Springer (1987)
14. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of async. circuits. In: Proc. CAV'92. LNCS, vol. 663, pp. 164–177. Springer (1993)
15. Padberg, J., Ehrig, H.: Parameterized net classes: A uniform approach to petri net classes. In: Unifying Petri Nets, Advances in Petri Nets. pp. 173–229 (2001)
16. Peled, D.A.: Partial order reduction: Model-checking using representatives. In: Mathematical Foundations of Computer Science. pp. 93–112 (1996)
17. Petri, C.A.: Fundamentals of a theory of asynchronous information flow. In: IFIP Congress. pp. 386–390 (1962)
18. Pinna, G.M., Poigné, A.: On the nature of events: Another perspective in concurrency. Theor. Comput. Sci. 138(2), 425–454 (1995)
19. Pratt, V.: Modeling concurrency with partial orders. International Journal of Parallel Programming 15(1), 33–71 (1986)
20. Pratt, V.R.: Event-state duality: The enriched case. In: CONCUR 2002 - Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings. pp. 41–56 (2002)
21. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: Concurrency Theory (CONCUR). Leibniz International Proceedings in Informatics, vol. 42, pp. 456–469. Dagstuhl Publishing (2015)
22. Sassone, V., Nielsen, M., Winskel, G.: Models for concurrency: Towards a classification. Theoretical Computer Science 170(1-2), 297–348 (1996)
23. Winskel, G.: An introduction to event structures. In: School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. pp. 364–397 (1988)