Asynchronous cascade products for trace languages and propositional dynamic logic

Pascal Weil

ReLaX, CNRS; and LaBRI, CNRS, Université de Bordeaux

Joint work with Bharat Adsul (IIT Bombay), Paul Gastin (LMF), Saptarshi Sarkar (IIT Bombay)

CIRM, DeLTA Meeting, June 2022

Overview

- Recognizable *trace* languages and *the Krohn-Rhodes theorem*

Overview

- Recognizable *trace* languages and *the Krohn-Rhodes theorem*
- The trace monoid = free partially commutative monoid = a model for distributed computations: see next slide

Overview

- Recognizable *trace* languages and *the Krohn-Rhodes theorem*
- The trace monoid = free partially commutative monoid = a model for distributed computations: see next slide
- The Krohn-Rhodes theorem for languages of finite words: here seen as a statement on automata. *Every regular language is accepted by a cascade product of very simple automata: permutation automata and 2-state reset automata*

Overview

- Recognizable *trace* languages and *the Krohn-Rhodes theorem*
- The trace monoid = free partially commutative monoid = a model for distributed computations: see next slide
- The Krohn-Rhodes theorem for languages of finite words: here seen as a statement on automata. *Every regular language is accepted by a cascade product of very simple automata: permutation automata and 2-state reset automata*
- Objective: a Krohn-Rhodes theorem for regular trace languages

Overview

- Recognizable *trace* languages and *the Krohn-Rhodes theorem*
- The trace monoid = free partially commutative monoid = a model for distributed computations: see next slide
- The Krohn-Rhodes theorem for languages of finite words: here seen as a statement on automata. *Every regular language is accepted by a cascade product of very simple automata: permutation automata and 2-state reset automata*
- Objective: a Krohn-Rhodes theorem for regular trace languages
- Main tool: a "distributed" logic which is expressively complete w.r.t. regular trace languages

Overview

- Recognizable *trace* languages and *the Krohn-Rhodes theorem*
- The trace monoid = free partially commutative monoid = a model for distributed computations: see next slide
- The Krohn-Rhodes theorem for languages of finite words: here seen as a statement on automata. *Every regular language is accepted by a cascade product of very simple automata: permutation automata and 2-state reset automata*
- Objective: a Krohn-Rhodes theorem for regular trace languages
- Main tool: a "distributed" logic which is expressively complete w.r.t. regular trace languages
- DeLTA 2021: a partial result in this direction, for aperiodic / star-free / FO-definable trace languages, by means of a distributed linear temporal logic

Distributed alphabet, trace monoid

- Distributed behaviors. $\mathscr{P}$, set of processes

Distributed alphabet, trace monoid

- Distributed behaviors. $\mathscr{P}$, set of processes
- (Σ, loc) distributed alphabet: $\mathrm{loc}\colon \Sigma \longrightarrow 2^{\mathscr{P}}$; letters a, b are *independent* if $\mathrm{loc}(a) \cap \mathrm{loc}(b) = \emptyset$

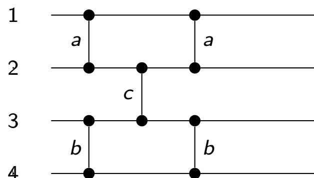
Distributed alphabet, trace monoid

- Distributed behaviors. $\mathscr{P}$, set of processes
- (Σ, loc) distributed alphabet: $\mathsf{loc}\colon \Sigma \longrightarrow 2^{\mathscr{P}}$; letters a, b are *independent* if $\mathsf{loc}(a) \cap \mathsf{loc}(b) = \emptyset$
- $\mathrm{Tr}(\Sigma)$: quotient of Σ^* by the relation $ab = ba$ if a, b are independent. Projection morphism: $\pi\colon \Sigma^* \longrightarrow \mathrm{Tr}(\Sigma)$

Distributed alphabet, trace monoid

- Distributed behaviors. $\mathscr{P}$, set of processes
- (Σ, loc) distributed alphabet: $\mathsf{loc}\colon \Sigma \longrightarrow 2^{\mathscr{P}}$; letters a, b are *independent* if $\mathsf{loc}(a) \cap \mathsf{loc}(b) = \emptyset$
- $\mathrm{Tr}(\Sigma)$: quotient of Σ^* by the relation $ab = ba$ if a, b are independent. Projection morphism: $\pi\colon \Sigma^* \longrightarrow \mathrm{Tr}(\Sigma)$
- poset representation of a trace: $t = (E, \leq, \lambda)$; $E =$ *events*

Let $\mathsf{loc}(a) = \{1, 2\}$, $\mathsf{loc}(b) = \{3, 4\}$, $\mathsf{loc}(c) = \{2, 3\}$, $t = abcba$



Recognizable trace languages

- Defn: $L \subseteq \mathrm{Tr}(\Sigma)$ is recognizable if $\pi^{-1}(L) \subseteq \Sigma^*$ (the language of all linearizations of the traces in L) is recognizable

Recognizable trace languages

- Defn: $L \subseteq \mathrm{Tr}(\Sigma)$ is recognizable if $\pi^{-1}(L) \subseteq \Sigma^*$ (the language of all linearizations of the traces in L) is recognizable
- Equivalently: there exists a morphism φ from $\mathrm{Tr}(\Sigma)$ to a finite transformation monoid (X, M) such that $L = \varphi^{-1}(M_{s,F})$ ($s \in X$ initial state, $F \subseteq X$ accepting states)

Recognizable trace languages

- Defn: $L \subseteq \mathrm{Tr}(\Sigma)$ is recognizable if $\pi^{-1}(L) \subseteq \Sigma^*$ (the language of all linearizations of the traces in L) is recognizable
- Equivalently: there exists a morphism φ from $\mathrm{Tr}(\Sigma)$ to a finite transformation monoid (X, M) such that $L = \varphi^{-1}(M_{s,F})$ ($s \in X$ initial state, $F \subseteq X$ accepting states)
- Characterization: Zielonka's asynchronous automata — see later

Recognizable trace languages

- Defn: $L \subseteq \mathrm{Tr}(\Sigma)$ is recognizable if $\pi^{-1}(L) \subseteq \Sigma^*$ (the language of all linearizations of the traces in L) is recognizable
- Equivalently: there exists a morphism φ from $\mathrm{Tr}(\Sigma)$ to a finite transformation monoid (X, M) such that $L = \varphi^{-1}(M_{s,F})$ ($s \in X$ initial state, $F \subseteq X$ accepting states)
- Characterization: Zielonka's asynchronous automata — see later
- Characterization: MSO-definable languages (letter predicates & binary predicate $\leq$)

Recognizable trace languages

- Defn: $L \subseteq \mathrm{Tr}(\Sigma)$ is recognizable if $\pi^{-1}(L) \subseteq \Sigma^*$ (the language of all linearizations of the traces in L) is recognizable
- Equivalently: there exists a morphism φ from $\mathrm{Tr}(\Sigma)$ to a finite transformation monoid (X, M) such that $L = \varphi^{-1}(M_{s,F})$ ($s \in X$ initial state, $F \subseteq X$ accepting states)
- Characterization: Zielonka's asynchronous automata — see later
- Characterization: MSO-definable languages (letter predicates & binary predicate $\leq$)
- PDL, *propositional dynamic logic*. First introduced to reason about programs (Fischer, Ladner 1979). A version for finite words (LDL, Giacomo, Vardi, 2013). Now for traces

Past Propositional Dynamic Logic PastPDL 1/2

- There are *event* formulas, *path* formulas and *trace* formulas

Past Propositional Dynamic Logic PastPDL 1/2

- There are *event* formulas, *path* formulas and *trace* formulas
- *Event formulas*:
  $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle$

Past Propositional Dynamic Logic PastPDL 1/2

- There are *event* formulas, *path* formulas and *trace* formulas
- *Event formulas*:
  $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle$
- *Path formulas*: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past Propositional Dynamic Logic PastPDL 1/2

- There are *event* formulas, *path* formulas and *trace* formulas
- *Event formulas*:
  $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle$
- *Path formulas*: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- Event formulas interpreted on events, path formulas interpreted on pairs of events: $t, e \models \varphi$ and $t, e, f \models \pi$

Past Propositional Dynamic Logic PastPDL 1/2

- There are *event* formulas, *path* formulas and *trace* formulas
- *Event formulas*:
  $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle$
- *Path formulas*: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- Event formulas interpreted on events, path formulas interpreted on pairs of events: $t, e \models \varphi$ and $t, e, f \models \pi$
- $t, e \models \langle\pi\rangle$ if there exists an event f such that $t, e, f \models \pi$

Past Propositional Dynamic Logic PastPDL 1/2

- There are *event* formulas, *path* formulas and *trace* formulas
- *Event formulas*:
  $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle$
- *Path formulas*: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- Event formulas interpreted on events, path formulas interpreted on pairs of events: $t, e \models \varphi$ and $t, e, f \models \pi$
- $t, e \models \langle\pi\rangle$ if there exists an event f such that $t, e, f \models \pi$
- $t, e, f \models \leftarrow_i$ if f is the immediate predecessor of e on process i

Past Propositional Dynamic Logic PastPDL 1/2

- There are *event* formulas, *path* formulas and *trace* formulas
- *Event formulas*:
  $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle$
- *Path formulas*: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- Event formulas interpreted on events, path formulas interpreted on pairs of events: $t, e \models \varphi$ and $t, e, f \models \pi$
- $t, e \models \langle\pi\rangle$ if there exists an event f such that $t, e, f \models \pi$
- $t, e, f \models \leftarrow_i$ if f is the immediate predecessor of e on process i
- $t, e, f \models \varphi?$ if $e = f$ and $t, e \models \varphi$

Past Propositional Dynamic Logic PastPDL 2/2

- Constant event formulas: a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$

Past Propositional Dynamic Logic PastPDL 2/2

- Constant event formulas: a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$
- If $t = (E, \leq, \lambda)$ is a trace, E_i is the set of i-events of t; if $e \in E$, $\downarrow e$ ($\Downarrow e$) is the (strict) past of e, $e_i = \Downarrow e \cap E_i$

Past Propositional Dynamic Logic PastPDL 2/2

- Constant event formulas: a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$
- If $t = (E, \leq, \lambda)$ is a trace, E_i is the set of i-events of t; if $e \in E$, $\downarrow e$ ($\Downarrow e$) is the (strict) past of e, $e_i = \Downarrow e \cap E_i$
- $t, e \models \mathsf{Y}_i \leq \mathsf{Y}_j$ if e_i, e_j exist and $e_i \leq e_j$

Past Propositional Dynamic Logic PastPDL 2/2

- Constant event formulas: a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$
- If $t = (E, \leq, \lambda)$ is a trace, E_i is the set of i-events of t; if $e \in E$, $\downarrow e$ ($\Downarrow e$) is the (strict) past of e, $e_i = \Downarrow e \cap E_i$
- $t, e \models \mathsf{Y}_i \leq \mathsf{Y}_j$ if e_i, e_j exist and $e_i \leq e_j$
- $t, e \models \mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ if $(e_i)_j, e_k$ exist and $(e_i)_j \leq e_k$

Past Propositional Dynamic Logic PastPDL 2/2

- Constant event formulas: a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$
- If $t = (E, \leq, \lambda)$ is a trace, E_i is the set of i-events of t; if $e \in E$, $\downarrow e$ ($\Downarrow e$) is the (strict) past of e, $e_i = \Downarrow e \cap E_i$
- $t, e \models \mathsf{Y}_i \leq \mathsf{Y}_j$ if e_i, e_j exist and $e_i \leq e_j$
- $t, e \models \mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ if $(e_i)_j, e_k$ exist and $(e_i)_j \leq e_k$
- *Trace formulas*: $\Phi ::= \mathsf{EM}\,\varphi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,j} \leq \mathsf{L}_k \mid \Phi \vee \Phi \mid \neg\Phi$

Past Propositional Dynamic Logic PastPDL 2/2

- Constant event formulas: a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$
- If $t = (E, \leq, \lambda)$ is a trace, E_i is the set of i-events of t; if $e \in E$, $\downarrow e$ ($\Downarrow e$) is the (strict) past of e, $e_i = \Downarrow e \cap E_i$
- $t, e \models \mathsf{Y}_i \leq \mathsf{Y}_j$ if e_i, e_j exist and $e_i \leq e_j$
- $t, e \models \mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ if $(e_i)_j, e_k$ exist and $(e_i)_j \leq e_k$
- *Trace formulas*: $\Phi ::= \mathsf{EM}\,\varphi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,j} \leq \mathsf{L}_k \mid \Phi \vee \Phi \mid \neg\Phi$
- $t \models \mathsf{EM}\,\varphi$ if φ holds at some maximal event of t

Past Propositional Dynamic Logic PastPDL 2/2

- Constant event formulas: a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$
- If $t = (E, \leq, \lambda)$ is a trace, E_i is the set of i-events of t; if $e \in E$, $\downarrow e$ ($\Downarrow e$) is the (strict) past of e, $e_i = \Downarrow e \cap E_i$
- $t, e \models \mathsf{Y}_i \leq \mathsf{Y}_j$ if e_i, e_j exist and $e_i \leq e_j$
- $t, e \models \mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ if $(e_i)_j, e_k$ exist and $(e_i)_j \leq e_k$
- *Trace formulas*: $\Phi ::= \mathsf{EM}\,\varphi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,j} \leq \mathsf{L}_k \mid \Phi \vee \Phi \mid \neg\Phi$
- $t \models \mathsf{EM}\,\varphi$ if φ holds at some maximal event of t
- $t \models \mathsf{L}_i \leq \mathsf{L}_j$ if $E_i, E_j \neq \emptyset$ and $\max E_i \leq \max E_j$

Past Propositional Dynamic Logic PastPDL 2/2

- Constant event formulas: a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$
- If $t = (E, \leq, \lambda)$ is a trace, E_i is the set of i-events of t; if $e \in E$, $\downarrow e$ ($\Downarrow e$) is the (strict) past of e, $e_i = \Downarrow e \cap E_i$
- $t, e \models \mathsf{Y}_i \leq \mathsf{Y}_j$ if e_i, e_j exist and $e_i \leq e_j$
- $t, e \models \mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ if $(e_i)_j, e_k$ exist and $(e_i)_j \leq e_k$
- *Trace formulas*: $\Phi ::= \mathsf{EM}\,\varphi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,j} \leq \mathsf{L}_k \mid \Phi \vee \Phi \mid \neg\Phi$
- $t \models \mathsf{EM}\,\varphi$ if φ holds at some maximal event of t
- $t \models \mathsf{L}_i \leq \mathsf{L}_j$ if $E_i, E_j \neq \emptyset$ and $\max E_i \leq \max E_j$
- $t \models \mathsf{L}_{i,j} \leq \mathsf{L}_k$ if $E_i, E_j \cap \downarrow E_i, E_k \neq \emptyset$ and $\max E_j \cap \downarrow E_i \leq \max E_k$

PastPDL and path automata, LocPastPDL

- Path formulas: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

PastPDL and path automata, LocPastPDL

- Path formulas: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- They define regular expressions on the alphabet of local left moves $\leftarrow_i$ ($i \in \mathscr{P}$) and of tests $\varphi?$

PastPDL and path automata, LocPastPDL

- Path formulas: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- They define regular expressions on the alphabet of local left moves $\leftarrow_i$ ($i \in \mathscr{P}$) and of tests $\varphi?$
- Variant: path automata = DFA $\mathcal{A}$ on a finite alphabet of local left moves and tests

PastPDL and path automata, LocPastPDL

- Path formulas: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- They define regular expressions on the alphabet of local left moves $\leftarrow_i$ ($i \in \mathscr{P}$) and of tests $\varphi?$
- Variant: path automata = DFA $\mathcal{A}$ on a finite alphabet of local left moves and tests
- Equivalent definition of PastPDL: no more path formulas, and event formulas are
  $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle \mathcal{A} \rangle$

PastPDL and path automata, LocPastPDL

- Path formulas: $\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- They define regular expressions on the alphabet of local left moves $\leftarrow_i$ ($i \in \mathscr{P}$) and of tests $\varphi?$
- Variant: path automata = DFA $\mathcal{A}$ on a finite alphabet of local left moves and tests
- Equivalent definition of PastPDL: no more path formulas, and event formulas are
  $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle \mathcal{A} \rangle$
- LocPastPDL = the *local* fragment of PastPDL where the path automata $\mathcal{A}$ use left moves all on the same process i

Expressive completeness

Theorem

PastPDL *and* LocPastPDL *are expressively complete with respect to regular trace languages.*

Expressive completeness

Theorem

PastPDL *and* LocPastPDL *are expressively complete with respect to regular trace languages.*

- PastPDL can easily be translated into MSO: it specifies only regular trace languages

Expressive completeness

Theorem

PastPDL *and* LocPastPDL *are expressively complete with respect to regular trace languages.*

- PastPDL can easily be translated into MSO: it specifies only regular trace languages
- Conversely, let L be a trace language recognized by a morphism $\eta\colon \mathrm{Tr}(\Sigma) \longrightarrow M$, into a finite monoid; wlog $L = \eta^{-1}(m)$. Find a LocPastPDL formula for L.

Expressive completeness

Theorem

PastPDL *and* LocPastPDL *are expressively complete with respect to regular trace languages.*

- PastPDL can easily be translated into MSO: it specifies only regular trace languages
- Conversely, let L be a trace language recognized by a morphism $\eta\colon \mathrm{Tr}(\Sigma) \longrightarrow M$, into a finite monoid; wlog $L = \eta^{-1}(m)$. Find a LocPastPDL formula for L.
- The proof is complex and subtle

Expressive completeness

Theorem

PastPDL *and* LocPastPDL *are expressively complete with respect to regular trace languages.*

- PastPDL can easily be translated into MSO: it specifies only regular trace languages
- Conversely, let L be a trace language recognized by a morphism $\eta\colon \mathrm{Tr}(\Sigma) \longrightarrow M$, into a finite monoid; wlog $L = \eta^{-1}(m)$. Find a LocPastPDL formula for L.
- The proof is complex and subtle
- First: find an event formula $\varphi^{(m)}$ such that, if t is a prime trace (one with a single maximal event), $\eta(t) = m$ if and only if $t, \max(t) \models \varphi^{(m)}$

Expressive completeness

Theorem

PastPDL *and* LocPastPDL *are expressively complete with respect to regular trace languages.*

- PastPDL can easily be translated into MSO: it specifies only regular trace languages
- Conversely, let L be a trace language recognized by a morphism $\eta\colon \mathrm{Tr}(\Sigma) \longrightarrow M$, into a finite monoid; wlog $L = \eta^{-1}(m)$. Find a LocPastPDL formula for L.
- The proof is complex and subtle
- First: find an event formula $\varphi^{(m)}$ such that, if t is a prime trace (one with a single maximal event), $\eta(t) = m$ if and only if $t, \max(t) \models \varphi^{(m)}$
- For the general case, decompose an arbitrary (non prime) trace into a product of prime traces in a controlled fashion

Digression: Cascade product of DFAs 1/2

- Consider a DFA $\mathcal{A} = (Q, \delta, s_{\text{in}})$, with transition morphism φ

Digression: Cascade product of DFAs 1/2

- Consider a DFA $\mathcal{A} = (Q, \delta, s_{\rm in})$, with transition morphism φ
- The associated *sequential transducer* $\sigma_{\mathcal{A}} \colon \Sigma^* \longrightarrow (\Sigma \times Q)^*$ maps 1 to 1, and ua to $\sigma_{\mathcal{A}}(ua) = \sigma_{\mathcal{A}}(u)(a, q)$, where $q = s_{\rm in} \cdot u$. In other words, q is what $\mathcal{A}$ knows *before* this a; and (a, q) is the name of the last used transition of $\mathcal{A}$

Digression: Cascade product of DFAs 1/2

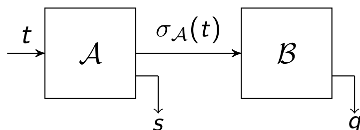
- Consider a DFA $\mathcal{A} = (Q, \delta, s_{\rm in})$, with transition morphism φ
- The associated *sequential transducer* $\sigma_{\mathcal{A}}\colon \Sigma^* \longrightarrow (\Sigma \times Q)^*$ maps 1 to 1, and ua to $\sigma_{\mathcal{A}}(ua) = \sigma_{\mathcal{A}}(u)(a, q)$, where $q = s_{\rm in} \cdot u$. In other words, q is what $\mathcal{A}$ knows *before* this a; and (a, q) is the name of the last used transition of $\mathcal{A}$
- $\sigma_{\mathcal{A}}$ turns $\mathcal{A}$ from an *accepting* to a *computing* device. It is "the most general function" computed by $\mathcal{A}$

Digression: Cascade product of DFAs 1/2

- Consider a DFA $\mathcal{A} = (Q, \delta, s_{\text{in}})$, with transition morphism φ
- The associated *sequential transducer* $\sigma_{\mathcal{A}} \colon \Sigma^* \longrightarrow (\Sigma \times Q)^*$ maps 1 to 1, and ua to $\sigma_{\mathcal{A}}(ua) = \sigma_{\mathcal{A}}(u)(a, q)$, where $q = s_{\text{in}} \cdot u$. In other words, q is what $\mathcal{A}$ knows *before* this a; and (a, q) is the name of the last used transition of $\mathcal{A}$
- $\sigma_{\mathcal{A}}$ turns $\mathcal{A}$ from an *accepting* to a *computing* device. It is "the most general function" computed by $\mathcal{A}$
- If $\mathcal{B}$ is a DFA on alphabet $\Sigma \times Q$, the *cascade product* $\mathcal{A} \circ \mathcal{B}$ processes an input word w through $\mathcal{A}$, with output $\sigma_{\mathcal{A}}(w)$, and then processes this output through $\mathcal{B}$

Digression: Cascade product of DFAs 1/2

- Consider a DFA $\mathcal{A} = (Q, \delta, s_{\rm in})$, with transition morphism φ
- The associated *sequential transducer* $\sigma_{\mathcal{A}} \colon \Sigma^* \longrightarrow (\Sigma \times Q)^*$ maps 1 to 1, and ua to $\sigma_{\mathcal{A}}(ua) = \sigma_{\mathcal{A}}(u)(a, q)$, where $q = s_{\rm in} \cdot u$. In other words, q is what $\mathcal{A}$ knows *before* this a; and (a, q) is the name of the last used transition of $\mathcal{A}$
- $\sigma_{\mathcal{A}}$ turns $\mathcal{A}$ from an *accepting* to a *computing* device. It is "the most general function" computed by $\mathcal{A}$
- If $\mathcal{B}$ is a DFA on alphabet $\Sigma \times Q$, the *cascade product* $\mathcal{A} \circ \mathcal{B}$ processes an input word w through $\mathcal{A}$, with output $\sigma_{\mathcal{A}}(w)$, and then processes this output through $\mathcal{B}$



Digression: Cascade product of DFAs 2/2

- *Wreath product* = translation of the cascade product on transformation monoids, which really is just the composition of the corresponding sequential transducers

Digression: Cascade product of DFAs 2/2

- *Wreath product* = translation of the cascade product on transformation monoids, which really is just the composition of the corresponding sequential transducers

Theorem (Krohn, Rhodes 196x)

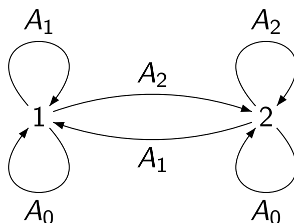
Every regular language is accepted by a cascade product of permutation automata and 2-state reset automata.

Digression: Cascade product of DFAs 2/2

- *Wreath product* = translation of the cascade product on transformation monoids, which really is just the composition of the corresponding sequential transducers

Theorem (Krohn, Rhodes 196x)

Every regular language is accepted by a cascade product of permutation automata and 2-state reset automata.

- 

Back to trace languages: Asynchronous automata

- A model of automata that implements the distributed structure of (Σ, loc)

Back to trace languages: Asynchronous automata

- A model of automata that implements the distributed structure of (Σ, loc)
- For each $i \in \mathscr{P}$, Q_i = *i-states*. *Global states*: $Q_{\mathscr{P}} = \prod_i Q_i$.
  For $a \in \Sigma$, *a-states*: $Q_a = \prod_{i \in \mathrm{loc}(a)} Q_i$

Back to trace languages: Asynchronous automata

- A model of automata that implements the distributed structure of (Σ, loc)
- For each $i \in \mathscr{P}$, Q_i = *i-states*. *Global states*: $Q_{\mathscr{P}} = \prod_i Q_i$. For $a \in \Sigma$, *a-states*: $Q_a = \prod_{i \in \mathrm{loc}(a)} Q_i$
- For each letter $a \in \Sigma$, the action of a depends only on, and modifies only the a-states: $\delta_a \colon Q_a \longrightarrow Q_a$, extended to a transformation Δ_a of $Q_{\mathscr{P}}$

Back to trace languages: Asynchronous automata

- A model of automata that implements the distributed structure of (Σ, loc)
- For each $i \in \mathscr{P}$, Q_i = *i-states*. *Global states*: $Q_{\mathscr{P}} = \prod_i Q_i$.
  For $a \in \Sigma$, *a-states*: $Q_a = \prod_{i \in \mathrm{loc}(a)} Q_i$
- For each letter $a \in \Sigma$, the action of a depends only on, and modifies only the a-states: $\delta_a \colon Q_a \longrightarrow Q_a$, extended to a transformation Δ_a of $Q_{\mathscr{P}}$
- Global initial and accepting states

Back to trace languages: Asynchronous automata

- A model of automata that implements the distributed structure of (Σ, loc)
- For each $i \in \mathscr{P}$, Q_i = *i-states*. *Global states*: $Q_{\mathscr{P}} = \prod_i Q_i$. For $a \in \Sigma$, *a-states*: $Q_a = \prod_{i \in \mathrm{loc}(a)} Q_i$
- For each letter $a \in \Sigma$, the action of a depends only on, and modifies only the a-states: $\delta_a \colon Q_a \longrightarrow Q_a$, extended to a transformation Δ_a of $Q_{\mathscr{P}}$
- Global initial and accepting states

Theorem (Zielonka, 1987)

Every recognizable language L in $\mathrm{Tr}(\Sigma)$ is accepted by an asynchronous automaton

Asynchronous automata as computing devices

- Same idea: turn an asynchronous automaton to a computing device. Consider an asynch. automaton $\mathcal{A} = (\{Q_i\}, \{\delta_a\}, s_{\text{in}})$, with transition morphism φ

Asynchronous automata as computing devices

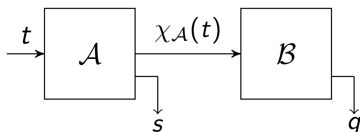
- Same idea: turn an asynchronous automaton to a computing device. Consider an asynch. automaton $\mathcal{A} = (\{Q_i\}, \{\delta_a\}, s_{\rm in})$, with transition morphism φ
- The *asynchronous transducer* $\chi_{\mathcal{A}}$ maps each trace $t = (E, \leq, \lambda)$ to a trace $(E, \leq, (\lambda, \mu))$ as follows: if $e \in E$ and $\lambda(e) = a$, if $t_e =\downarrow e$ is the causal past of e (all events $e' < e$), then $\mu(e) = (s_{\rm in} \cdot \varphi(t_e))_a$. That is: $\mu(e)$ *is the local view (a-view) of the causal past of e* — exactly as in the seq'l case

Asynchronous automata as computing devices

- Same idea: turn an asynchronous automaton to a computing device. Consider an asynch. automaton $\mathcal{A} = (\{Q_i\}, \{\delta_a\}, s_{\mathrm{in}})$, with transition morphism φ
- The *asynchronous transducer* $\chi_{\mathcal{A}}$ maps each trace $t = (E, \leq, \lambda)$ to a trace $(E, \leq, (\lambda, \mu))$ as follows: if $e \in E$ and $\lambda(e) = a$, if $t_e = \downarrow e$ is the causal past of e (all events $e' < e$), then $\mu(e) = (s_{\mathrm{in}} \cdot \varphi(t_e))_a$. That is: $\mu(e)$ *is the local view (a-view) of the causal past of* e — exactly as in the seq'l case
- Again, the function $\chi_{\mathcal{A}}$ is “the most general function” computed by $\mathcal{A}$ using only local state information

Local cascade product

- *Local cascade product* $\mathcal{A} \circ_\ell \mathcal{B}$: process input trace t through $\mathcal{A}$, output $\chi_\mathcal{A}(t)$, and then process it through $\mathcal{B}$



Interesting asynchronous automata

- Localized reset: 2 states on process i, 1 state on each other process (so 2 global states) $=$ a reset automaton on process i, which does nothing on other processes

Interesting asynchronous automata

- Localized reset: 2 states on process i, 1 state on each other process (so 2 global states) = a reset automaton on process i, which does nothing on other processes
- Localized permutation automaton: a permutation automaton on process i, which does nothing on other processes

Interesting asynchronous automata

- Localized reset: 2 states on process i, 1 state on each other process (so 2 global states) = a reset automaton on process i, which does nothing on other processes
- Localized permutation automaton: a permutation automaton on process i, which does nothing on other processes
- The *gossip automaton* $\mathcal{G}$ (Mukund, Sohoni, 1997), which depends only on the distributed architecture (Σ, loc), locally computes the truth values of the constants $\mathsf{Y}_i \leq \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$

Interesting asynchronous automata

- Localized reset: 2 states on process i, 1 state on each other process (so 2 global states) = a reset automaton on process i, which does nothing on other processes
- Localized permutation automaton: a permutation automaton on process i, which does nothing on other processes
- The *gossip automaton* $\mathcal{G}$ (Mukund, Sohoni, 1997), which depends only on the distributed architecture (Σ, loc), locally computes the truth values of the constants $\mathsf{Y}_i \leq \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$
- And the global states of $\mathcal{G}$ compute the truth values of the constants $\mathsf{L}_i \leq \mathsf{L}_j$ and $\mathsf{L}_{i,j} \leq \mathsf{L}_k$

Decomposition theorem

- Let θ^Y be the trace function which decorates each event of a trace with the tuple of the truth values of the $\mathsf{Y}_i \leq \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ — it is computed by $\mathcal{G}$

Decomposition theorem

- Let θ^Y be the trace function which decorates each event of a trace with the tuple of the truth values of the $\mathsf{Y}_i \leq \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ — it is computed by $\mathcal{G}$

Theorem

Let φ be a LocPastPDL *event formula and let θ^φ be the asynchronous function which decorates each event of a trace with the truth value of φ. Then θ^φ is computed by a restricted local cascade product of $\mathcal{G}$ with a local cascade product of localized reset and permutation automata — where restricted means that $\mathcal{G}$ passes only the θ^Y-image of its input*

Decomposition theorem

- Let θ^Y be the trace function which decorates each event of a trace with the tuple of the truth values of the $\mathsf{Y}_i \leq \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ — it is computed by $\mathcal{G}$

Theorem

Let φ be a LocPastPDL *event formula and let θ^φ be the asynchronous function which decorates each event of a trace with the truth value of φ. Then θ^φ is computed by a restricted local cascade product of $\mathcal{G}$ with a local cascade product of localized reset and permutation automata — where restricted means that $\mathcal{G}$ passes only the θ^Y-image of its input*

Corollary

Every regular trace language is accepted by a cascade product as above

About the proof 1/2

- By structural induction on φ

About the proof 1/2

- By structural induction on φ
- Recall $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\mathcal{A}\rangle$

About the proof 1/2

- By structural induction on φ
- Recall $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle \mathcal{A} \rangle$
- Easy for the constants a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$, and to deal with Boolean connectors

About the proof 1/2

- By structural induction on φ
- Recall $\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle \mathcal{A} \rangle$
- Easy for the constants a, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$, and to deal with Boolean connectors
- Last case: $\varphi = \langle \mathcal{A} \rangle$, with $\mathcal{A}$ i-local.

About the proof 2/2

- Let $\varphi = \langle \mathcal{A} \rangle$, with $\mathcal{A}$ i-local. Let F be the set of event formulas that are used in test transitions of $\mathcal{A}$ and θ^F the function which decorates a trace with the truth values of all the $\psi \in F$.

About the proof 2/2

- Let $\varphi = \langle \mathcal{A} \rangle$, with $\mathcal{A}$ i-local. Let F be the set of event formulas that are used in test transitions of $\mathcal{A}$ and θ^F the function which decorates a trace with the truth values of all the $\psi \in F$.
- For each $\psi \in F$, the function θ^ψ is computed by a local cascade product in the appropriate form. Combine them all (in a sort of direct product) to get $\mathcal{A}_F$, a local cascade product in the appropriate form computing θ^F

About the proof 2/2

- Let $\varphi = \langle \mathcal{A} \rangle$, with $\mathcal{A}$ i-local. Let F be the set of event formulas that are used in test transitions of $\mathcal{A}$ and θ^F the function which decorates a trace with the truth values of all the $\psi \in F$.
- For each $\psi \in F$, the function θ^ψ is computed by a local cascade product in the appropriate form. Combine them all (in a sort of direct product) to get $\mathcal{A}_F$, a local cascade product in the appropriate form computing θ^F
- Let now $\mathcal{L}_F(\mathcal{A})$ be the set of all $\theta^F(t) \cap E_i \cap {\downarrow} e \cap \uparrow f$, for all events e, f of t such that $t, e, f \models \langle \mathcal{A} \rangle$. Note $\mathcal{L}_F(\mathcal{A})$ is a language of words

About the proof 2/2

- Let $\varphi = \langle \mathcal{A} \rangle$, with $\mathcal{A}$ i-local. Let F be the set of event formulas that are used in test transitions of $\mathcal{A}$ and θ^F the function which decorates a trace with the truth values of all the $\psi \in F$.
- For each $\psi \in F$, the function θ^ψ is computed by a local cascade product in the appropriate form. Combine them all (in a sort of direct product) to get $\mathcal{A}_F$, a local cascade product in the appropriate form computing θ^F
- Let now $\mathcal{L}_F(\mathcal{A})$ be the set of all $\theta^F(t) \cap E_i \cap \downarrow e \cap \uparrow f$, for all events e, f of t such that $t, e, f \models \langle \mathcal{A} \rangle$. Note $\mathcal{L}_F(\mathcal{A})$ is a language of words
- A technical lemma: $\mathcal{L}_F(\mathcal{A})$ is regular. So use the classical Krohn-Rhodes: it is accepted by a local cascade product $\mathcal{B}$ of i-localized reset and permutation automata, and combine as $\mathcal{A}_F \circ_\ell \mathcal{B}$.

Thank you for your attention!

Some questions (Adsul, Gastin, Saptarshi, W. CONCUR 2020, LMCS 2022)

- Is the gossip automaton needed?

Some questions (Adsul, Gastin, Saptarshi, W. CONCUR 2020, LMCS 2022)

- Is the gossip automaton needed?
- We know how to get rid of it only if (Σ, loc) is an *acyclic architecture*, that is, the *communication graph* (vertices $= \mathscr{P}$; there is an edge from process i to process j if $\Sigma_i \cap \Sigma_j \neq \emptyset$) is acyclic (see Krishna & Muscholl 2013)

Some questions (Adsul, Gastin, Saptarshi, W. CONCUR 2020, LMCS 2022)

- Is the gossip automaton needed?
- We know how to get rid of it only if (Σ, loc) is an *acyclic architecture*, that is, the *communication graph* (vertices $= \mathscr{P}$; there is an edge from process i to process j if $\Sigma_i \cap \Sigma_j \neq \emptyset$) is acyclic (see Krishna & Muscholl 2013)
- What about aperiodic, or first-order definable trace languages?

Some questions (Adsul, Gastin, Saptarshi, W. CONCUR 2020, LMCS 2022)

- Is the gossip automaton needed?
- We know how to get rid of it only if (Σ, loc) is an *acyclic architecture*, that is, the *communication graph* (vertices $= \mathscr{P}$; there is an edge from process i to process j if $\Sigma_i \cap \Sigma_j \neq \emptyset$) is acyclic (see Krishna & Muscholl 2013)
- What about aperiodic, or first-order definable trace languages?
- We don't need the (localized) permutation automata — but we still need the gossip automaton in general (exception: acyclic architecture). And $\mathcal{G}$ is not aperiodic. But we use it only to pass on first-order definable information, namely the truth values of the $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$