Parametrized automata simulation and application to service composition

Walid Belkhir¹
Y. Chevalier², M. Rusinowitch¹

¹Loria, CASSIS
²Université Paul Sabatier & IRIT Toulouse

ICE Workshop, 2015
Motivations:
Why parametrized automata?

Automata over infinite alphabets:
- Web services: exchanging data over infinite domain
- XML documents: nodes carrying data over infinite domain
- Program verification: variables, time
Challenges:

- Extending the formalism of finite automata to infinite domain
- Simplicity and readability (for non-experts)
- Closure properties (intersection, union, Kleene operator, concatenation, . . . )
- Decidability of main problems (membership, universality, emptiness, inclusion, (bi)simulation, . . . )
- Applications: e.g. Web Services, protocols
Motivating example
Service composition: simulation of CLIENT by CART⊗SEARCH

\[p_0 \xrightarrow{\text{!Create\_Cart}} p_1 \xrightarrow{\text{?End\_Cart}} p_2 \xrightarrow{\text{!Search}} p_3 \xrightarrow{\text{?Num}} \]

\[q_0 \xrightarrow{\text{?Create\_Cart}} q_1 \xrightarrow{\text{!Add\_Cart}} \]

\[r_0 \xrightarrow{\text{!Num}} r_1 \xrightarrow{\text{?Search}} \]

\[x \text{ is refreshed at } p_0, p_1 \]
\[y \text{ is refreshed at } p_0 \]
\[z \text{ is refreshed at } q_0 \]
\[u \text{ is refreshed at } q_0, q_1 \]
\[w \text{ is refreshed at } r_0 \]
Parametrized automata (PAs)

Key ideas:

- Extension of finite automata to infinite alphabet (finite words, $|\Sigma| = \infty$)
- Transitions are labeled with variables or letters
  - The variables range over $\Sigma$
- Transitions are guarded with equalities/disequalities
- Variables are refreshed in some states
Parametrized automata (PAs)

Key ideas:

- Extension of finite automata to infinite alphabet (finite words, $|\Sigma| = \infty$)
- Transitions are labeled with variables or letters
  - The variables range over $\Sigma$
- Transitions are guarded with equalities/disequalities
- Variables are refreshed in some states

$\mathcal{A}_1$

$\mathcal{A}_2$

$\text{Fresh}(x) = \{p_0\}$ and $\text{Fresh}(z) = \{q_0, q_1, q_2\}$
Parametrized automata (PAs)

Key ideas:

- Extension of finite automata to infinite alphabet (finite words, $|\Sigma| = \infty$)
- Transitions are labeled with variables or letters
  - The variables range over $\Sigma$
- Transitions are guarded with equalities/disequalities
- Variables are refreshed in some states

$L(A_1) = \{a_0a_0a_1a_1 \ldots a_na_n \mid a_i \in \Sigma\}$

$Fresh(x) = \{p_0\}$ and $Fresh(z) = \{q_0, q_1, q_2\}$
Parametrized automata (PAs)

Key ideas:

- Extension of finite automata to infinite alphabet (finite words, $|\Sigma| = \infty$)
- Transitions are labeled with variables or letters
  - The variables range over $\Sigma$
- Transitions are guarded with equalities/disequalities
- Variables are refreshed in some states

\[ \mathcal{A}_1 \]

\[ \mathcal{A}_2 \]

- \( \text{Fresh}(x) = \{p_0\} \) and \( \text{Fresh}(z) = \{q_0, q_1, q_2\} \)
- \( L(\mathcal{A}_1) = \{a_0a_0a_1a_1 \ldots a_na_n \mid a_i \in \Sigma\} \)
- \( L(\mathcal{A}_2) = \) all words in which a letter appears at least twice
Parametrized automata

Example

\( p_0 \xrightarrow{y} p_1 \) \( y, y \neq x \) \( B_1 \)

\( \text{Fresh}(y) = \{ p_0 \} \)

\( q_0 \xrightarrow{x} q_1 \)

\( \text{Fresh}(x) = \text{Fresh}(y) = \{ q_0 \} \)

\( B_2 \)

\( y, y \neq x \)
$y, y \neq x$

$\mathcal{B}_1$

$p_0 \rightarrow x \rightarrow p_1$

$Fresh(y) = \{p_0\}$

$\mathcal{B}_2$

$q_0 \xrightarrow{x} q_1 \xleftarrow{y, y \neq x} q_0$

$Fresh(x) = Fresh(y) = \{q_0\}$

$L(\mathcal{B}_1) =$ all the words in which the last letter *differs* from all the other letters
Parametrized automata

Example

\[ y, \ y \neq x \]

\[ B_1 \]

\[ p_0 \rightarrow x \rightarrow p_1 \]

\[ \text{Fresh}(y) = \{p_0\} \]

\[ L(B_1) = \text{all the words in which the last letter differs from all the other letters} \]

\[ \text{Fresh}(x) = \text{Fresh}(y) = \{q_0\} \]

\[ B_2 \]

\[ q_0 \rightarrow x \rightarrow q_1 \]

\[ y, \ y \neq x \]

\[ \text{Fresh}(x) = \text{Fresh}(y) = \{q_0\} \]

\[ L(B_2) = \{a_1 a'_1 \ldots a_n a'_n \mid a_i \neq a'_i, a_i \in \Sigma\} \]
**Definition of PAs**

A PA is a tuple $\mathcal{A} = \langle \Sigma, \mathcal{X}, Q, Q_0, \delta, F, \kappa \rangle$ where

- $\Sigma$ is a infinite set of letters,
- $\mathcal{X}$ is a finite set of variables,
- $Q$ is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states,
- $\delta : Q \times (\Sigma_\mathcal{A} \cup \mathcal{X}) \times \mathcal{G} \rightarrow 2^Q$ is a transition function where $\Sigma_\mathcal{A}$ is a finite subset of $\Sigma$,
- $F \subseteq Q$ is a set of accepting states,
- $\kappa : \mathcal{X} \rightarrow 2^Q$ is the refreshing function that associates to every variable the (possibly empty) set of states where it is refreshed.

$$\mathcal{G} ::= (\bigwedge_i \alpha_i = \beta_i) \land (\bigwedge_j \alpha_j \neq \beta_j), \quad \alpha_i, \beta_i, \alpha_j, \beta_j \in \Sigma_\mathcal{A} \cup \mathcal{X}$$
Comparison with other models

- **FVA**: Fresh-variable automata (Belkhir et. al)
- **FMA**: Finite memory automata (Kaminsky et. al)
- **NVA**: Nondeterminist variable automata (Kupferman et. al)
- **PA**: Parametrized automata (Belkhir et. al)
- **FRA**: Fresh-register automata (N. Tzevelekos)
Checking whether a service satisfies a policy can be reduced to checking that the intersection of the service PA with a PA specifying the forbidden executions is empty.

- Closure of PAs under intersection
- Checking Nonemptyness of PAs
Closure properties and decision procedures in service composition

Checking whether a service satisfies a policy can be reduced to checking that the intersection of the service PA with a PA specifying the forbidden executions is empty.

- Closure of PAs under intersection
- Checking Nonemptyness of PAs

If the policy is expressed as a language of authorized traces, then checking whether a service $\mathcal{M}$ respects the policy $\mathcal{Q}$ amounts to check the containment $L(\mathcal{M}) \subseteq L(\mathcal{Q})$.

- Decidability of containment
- If containment is undecidable, it can be underapproximated by simulation preorder.
A usage policy for opening and reading files

- A file named $f$ must be open before being used, where $f$ is a variable,
- the number of files which are open at the same time is at most one.
Closure properties of PAs

Theorem

PAs are closed under intersection, union, concatenation and Kleene operator but not under complementation.

- Union, intersection: Straightforward
- Concatenation & Kleene operator: Introduction of unguarded $\varepsilon$-transitions does not increase the expressivity of PAs
- Complementation: the language of all words in which a letter appears at least twice cannot be complemented.
Theorem

For PAs

- Membership (i.e. $w \in L(\mathcal{A})$?) is NP-complete,
- Universality (i.e. $L(\mathcal{A}) = \Sigma^*$?) is undecidable,
- Inclusion (i.e. $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$?) is undecidable, and
- Nonemptiness (i.e. $L(\mathcal{A}) \neq \emptyset$?) is PSPACE-Complete.
Theorem

For PAs

- Membership (i.e. $w \in L(A)$?) is NP-complete,
- Universality (i.e. $L(A) = \Sigma^*$?) is undecidable,
- Inclusion (i.e. $L(A_1) \subseteq L(A_2)$?) is undecidable, and
- Nonemptiness (i.e. $L(A) \neq \emptyset$?) is PSPACE-Complete.

- Membership: guessing an accepting path. Reduction of Hamiltonian cycle
- Universality & Inclusion: undecidable for the strict subclass of Finite Memory Automata (FMA)
Lemma

Nonemptiness (i.e. $L(A) \neq \emptyset$) is PSPACE.

- Reduction to the Nonemptiness of PAs in which the variables range over a finite set of letters.
Lemma

Nonemptiness (i.e. \( L(A) \neq \emptyset \)) is PSPACE.

- Reduction to the Nonemptiness of PAs in which the variables range over a finite set of letters.
- A Turing machine stores two pieces of data:
Lemma

Nonemptiness (i.e. \(L(A) \neq \emptyset\)) is \(\text{PSPACE}\).

- Reduction to the Nonemptiness of PAs in which the variables range over a finite set of letters.
- A Turing machine stores two pieces of data:
  - the current configuration \((q, \sigma)\), where \(q\) is a state, \(\sigma\) is a substitution,
Lemma

Nonemptiness (i.e. \( L(A) \neq \emptyset \)) is PSPACE.

- Reduction to the Nonemptiness of PAs in which the variables range over a finite set of letters.
- A Turing machine stores two pieces of data:
  - the current configuration \((q, \sigma)\), where \(q\) is a state, \(\sigma\) is a substitution,
  - a counter for the number of configurations visited so far.
Lemma

Nonemptiness (i.e. $L(A) \neq \emptyset$?) is PSPACE.

- Reduction to the Nonemptiness of PAs in which the variables range over a finite set of letters.
- A Turing machine stores two pieces of data:
  - the current configuration $(q, \sigma)$, where $q$ is a state, $\sigma$ is a substitution,
  - a counter for the number of configurations visited so far.
- Since the size of this data is polynomial, a Turing machine running in polynomial space can solve the Nonemptiness problem:
Lemma

Nonemptiness (i.e. $L(A) \neq \emptyset$) is PSPACE.

- Reduction to the Nonemptiness of PAs in which the variables range over a finite set of letters.
- A Turing machine stores two pieces of data:
  - the current configuration $(q, \sigma)$, where $q$ is a state, $\sigma$ is a substitution,
  - a counter for the number of configurations visited so far.
- Since the size of this data is polynomial, a Turing machine running in polynomial space can solve the Nonemptiness problem:
  - exponential number of configurations.
Lemma

Nonemptiness (i.e. $L(A) \neq \emptyset$?) is PSPACE-hard.

- PAs can encode bounded one-counter automata ($\text{Boca}$):
  - PAs over the finite alphabet $\Sigma = \{0, 1\}$
  - variables play the role of registers (finite number of variables)
  - PAs can encode addition and subtraction

- Reachability of $\text{Boca}$ is PSPACE-complete:
  - reduce the reachability of $\text{Boca}$ to the Nonemptiness of PAs
$!x = !a_0, !a_1, !a_2, \ldots$ where $a_i \in \Sigma$

Diagram:

- **A**:
  - $p_0$ to $p_1$: $!x$
  - $p_1$ to $p_2$: $?z$
  - $p_2$ to $p_3$: $!w$

- **B**:
  - $q_0$ to $q_1$: $?y$
  - $q_1$ to $q_2$: $!a$
  - $q_2$ to $q_3$: $?y$
Theorem

The (communicating) simulation for PAs is in EXPTIME.

Key points of the proof:

1. Game theoretic characterization of the simulation ($\preceq$)
   (Eloise (service) vs Abelard (client))
   - $\mathcal{A}_1 \preceq \mathcal{A}_2$ iff Eloise has a winning strategy in $G_\Sigma(\mathcal{A}_1, \mathcal{A}_2)$
   - Positions of the game are pairs of configurations
     $((q_1, \sigma_1), (q_2, \sigma_2))$ where $q_i$ is a state of $\mathcal{A}_i$ and $\sigma_i : \mathcal{X} \to \Sigma$
     is a substitution
   - Infinite plays are winning for Eloise
Theorem

The (communicating) simulation for PAs is in EXPTIME.

Key points of the proof:

2. Constructing an equivalent game $G_{C_0}(A_1, A_2)$ in which the players instantiate the variables from the finite set of letters $C_0$:

$$C_0 = \Sigma_{A_1} \cup \Sigma_{A_2} \cup \{c_1, \ldots, c_n\}$$

where

$$\begin{align*}
\left\{ \Sigma_{A_i} \right\} &= \text{finite set of letters appearing in the PA } A_i, \\
\left| X_1 \cup X_2 \right| &= n
\end{align*}$$
How to prove that the games $G_\Sigma$ and $G_{C_0}$ are equivalent?

$G_\Sigma \implies G_{C_0}$:

$$G_\Sigma \quad G_{C_0}$$

$$\emptyset \overset{f}{\longrightarrow} \emptyset_0 = f(\emptyset)$$

$$\emptyset' \overset{f}{\longrightarrow} \emptyset'_0 = f(\emptyset')$$

$$\emptyset'' \overset{f}{\longrightarrow} \emptyset''_0 = f(\emptyset'')$$
Proof of direction $G_\Sigma \Rightarrow G_{C_0}$

Instantiation strategy:

1. $c \in \Sigma_{A_1} \cup \Sigma_{A_2}$
Proof of direction $G_\Sigma \Rightarrow G_{C_0}$

Instantiation strategy:

- $c \in \Sigma_{A_1} \cup \Sigma_{A_2}$
  - $c_0 = c$

\[
\Sigma \cap C_0 = \Sigma_{A_1} \cup \Sigma_{A_2}
\]
Proof of direction $G_{\Sigma} \Rightarrow G_{C_0}$

Instantiation strategy:

1. $c \in \Sigma_{A_1} \cup \Sigma_{A_2}$
   - $c_0 = c$

2. $c \in \Sigma \setminus C_0$ and $c$ is new in the current configuration

\[ \Sigma \cap C_0 = \Sigma_{A_1} \cup \Sigma_{A_2} \]
Proof of direction $G_\Sigma \Rightarrow G_{C_0}$

Instantiation strategy:

1. $c \in \Sigma_{A_1} \cup \Sigma_{A_2}$
   - $c_0 = c$

2. $c \in \Sigma \setminus C_0$ and $c$ is new in the current configuration
   - $c_0 \in C_0 \setminus (\Sigma \cap C_0)$ must be new

$\Sigma \cap C_0 = \Sigma_{A_1} \cup \Sigma_{A_2}$
Proof of direction $G_\Sigma \Rightarrow G_{C_0}$

Instantiation strategy:

1. $c \in \Sigma_{A_1} \cup \Sigma_{A_2}$
   - $c_0 = c$

2. $c \in \Sigma \setminus C_0$ and $c$ is new in the current configuration
   - $c_0 \in C_0 \setminus (\Sigma \cap C_0)$ must be new

3. $c \in \Sigma \setminus C_0$ and $c$ appears in the current configuration

\[ \Sigma \cap C_0 = \Sigma_{A_1} \cup \Sigma_{A_2} \]
Decidability of the (communicating) simulation (4/5)

Proof of direction $G_{\Sigma} \Rightarrow G_{C_0}$

Instantiation strategy:

1. $c \in \Sigma_{A_1} \cup \Sigma_{A_2}$
   - $c_0 = c$

2. $c \in \Sigma \setminus C_0$ and $c$ is **new** in the current configuration
   - $c_0 \in C_0 \setminus (\Sigma \cap C_0)$ must be new

3. $c \in \Sigma \setminus C_0$ and $c$ **appears** in the current configuration
   - $c_0 = $ back to the previous choice
Simulation for PAs over finite alphabet is in EXPTIME.

We use an alternating Turing machine running in polynomial space:

- We need to store two pieces of data:
  - the current game position \(((q_1, \sigma_1), (q_2, \sigma_2))\), where \(q_i\) is a state, \(\sigma_i\) is a substitution,
  - a counter for the number of positions visited so far.
Simulation for PAs over finite alphabet is in EXPTIME.

We use an alternating Turing machine running in polynomial space:

- We need to store two pieces of data:
  - the current game position \(((q_1, \sigma_1), (q_2, \sigma_2))\), where \(q_i\) is a state, \(\sigma_i\) is a substitution,
  - a counter for the number of positions visited so far.
- Since the size of this data is polynomial, an ATM can solve the simulation game:
  - existential states \(\sim\) player Eloise
  - universal states \(\sim\) player Abelard
Simulation for PAs over finite alphabet is in EXPTIME.

We use an alternating Turing machine running in polynomial space:

- We need to store two pieces of data:
  - the current game position \( ((q_1, \sigma_1), (q_2, \sigma_2)) \), where \( q_i \) is a state, \( \sigma_i \) is a substitution,
  - a counter for the number of positions visited so far.

- Since the size of this data is polynomial, an ATM can solve the simulation game:
  - existential states \( \sim \) player Eloise
  - universal states \( \sim \) player Abelard

- \( \text{APSPACE}=\text{EXPTIME} \)
### Decision procedures: summary

<table>
<thead>
<tr>
<th></th>
<th>FVAs</th>
<th>PAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Membership</td>
<td>NP-Complete</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>Nonemptiness</td>
<td>NL-Complete</td>
<td>PSPACE-Complete</td>
</tr>
<tr>
<td>Universality</td>
<td>Decidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>Inclusion</td>
<td>?</td>
<td>Undecidable</td>
</tr>
<tr>
<td>(bi)Simulation</td>
<td>EXPTIME</td>
<td>EXPTIME-complete</td>
</tr>
</tbody>
</table>

FVAs = PAs without guards (all guards are true)


Model checking of PAs with parametrized $\mu$-calculus

Service synthesis under policy enforcement as parametrized $\mu$-calculus model-checking

Service synthesis under timed constraints with timed PAs