Synthesizing Computable Functions from Synchronous Specifications

Sarah Winter

Université libre de Bruxelles, Belgium

January 6, 2021
YR-OWLS, online
Reactive Synthesis of Non-terminating Systems

synthesize

**Specification** ----> **Implementation**

one input is in relation with several outputs

algorithm that selects a unique output for each input
Church Synthesis

\[
\neg \text{req/} \neg \text{grt} \quad \ast/\neg \text{grt} \\
\text{req/} \neg \text{grt} \\
\ast/\text{grt} \\
\text{req/} \text{grt}
\]

**Synchronous specifications**
(synchronous relations)

e.g., given by synchronous transducers with parity acceptance
Church Synthesis

Synchronous specifications (synchronous relations)
  e.g., given by synchronous transducers with parity acceptance

Synchronous implementations given by Mealy machines
Church Synthesis

\[ \neg \text{req} / \neg \text{grt} \quad * / \neg \text{grt} \]

\[ \text{req} / \neg \text{grt} \]

\[ * / \text{grt} \]

\[ \text{req} / \text{grt} \]

\[ \neg \text{req} / \neg \text{grt} \]

\[ \neg \text{req} / \neg \text{grt} \]

\[ \text{req} / \text{grt} \]

**Synchronous specifications**
(synchronous relations)

e.g., given by synchronous transducers with parity acceptance

**Synthesizing Computable Functions from Synchronous Specifications – Sarah Winter 3 of 24**

**Theorem** (Büchi/Landweber’69). It is decidable whether a synchronous specification is implementable by a Mealy machine.

**Synchronous implementations**
given by Mealy machines
More Relaxed Implementations

**Goal** Decide whether a synchronous specification is implementable (by an algorithm/a program/a deterministic Turing machine).
More Relaxed Implementations

**Goal**  Decide whether a synchronous specification is implementable (by an algorithm/a program/a deterministic Turing machine).

**Example.**
- Specification: contains pairs of the form

\[(a_1a_2a_3\cdots,a_3\cdots) \in \{a,b\}^\omega \times \{a,b\}^\omega\]
More Relaxed Implementations

**Goal** Decide whether a synchronous specification is implementable (by an algorithm/a program/a deterministic Turing machine).

**Example.**

- Specification: contains pairs of the form

\[(a_1a_2a_3\cdots, a_3\cdots) \in \{a, b\}^{\omega} \times \{a, b\}^{\omega}\]

- no implementation by a Mealy machine exists,
More Relaxed Implementations

**Goal** Decide whether a synchronous specification is implementable (by an algorithm/a program/a deterministic Turing machine).

**Example.**

- Specification: contains pairs of the form
  \[(a_1a_2a_3\cdots, a_3\cdots) \in \{a, b\}^\omega \times \{a, b\}^\omega\]
- no implementation by a Mealy machine exists,
- can be implemented, every deterministic machine has to wait until it sees the third input letter
Example.

- Specification: contains pairs of the form

\[(uA\alpha, A^{|u|}\beta) \quad (uB\alpha, B^{|u|}\beta),\]

where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega, A, B\) are special letters.
More Relaxed Implementations

Example.

- Specification: contains pairs of the form

  \[(uA\alpha, A^{|u|}\beta) \ (uB\alpha, B^{|u|}\beta),\]

  where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega, A, B\) are special letters

- can be implemented, but, every deterministic machine has to wait arbitrary long to output something valid
More Relaxed Implementations

Example.

- Specification: contains pairs of the form

\[(uA\alpha, A^{\mid u \mid} \beta) \ (uB\alpha, B^{\mid u \mid} \beta),\]

where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega\), \(A, B\) are special letters

- can be implemented, but, every deterministic machine has to wait arbitrary long to output something valid

- e.g., implemented by a deterministic machine that computes the function

\[uA\alpha \mapsto A^{\mid u \mid} \alpha \quad uB\alpha \mapsto B^{\mid u \mid} \alpha\]
Computability

What does it mean to be implementable for a relation?
What does it mean to be **implementable** for a relation?

- There is a computable function $f$ with the same domain as the relation $R$ such that $(\alpha, f(\alpha)) \in R$ for all $\alpha \in \text{dom}(R)$.
What does it mean to be implementable for a relation?

> There is a computable function \( f \) with the same domain as the relation \( R \) such that \( (\alpha, f(\alpha)) \in R \) for all \( \alpha \in \text{dom}(R) \).

A function \( f : \Sigma^\omega \rightarrow \Gamma^\omega \) is computable if there exists a deterministic Turing machine that

- outputs longer and longer prefixes of an acceptable output
- while it reads longer and longer prefixes of the input.
Consider a deterministic Turing machine $M$ with
Consider a deterministic Turing machine $M$ with

- three tapes
  - a one-way read-only input tape
  - a two-way working tape
  - a one-way write-only output tape

$M(\alpha,k)$ denotes the output written after reading the first $k$ letters of the input sequence $\alpha$.

$M$ computes $f$ if for all $\alpha \in \text{dom}(f)$:

- $\forall k: M(\alpha,k)$ is a prefix of $f(\alpha)$, and
- $\forall i \exists j: |M(\alpha,j)| \geq i$.
Consider a deterministic Turing machine $M$ with

- three tapes
  - a one-way read-only input tape
  - a two-way working tape
  - a one-way write-only output tape

$M(\alpha, k)$ denotes the output written after reading the first $k$ letters of the input sequence $\alpha$
Consider a deterministic Turing machine $M$ with

- three tapes
  - a one-way read-only input tape
  - a two-way working tape
  - a one-way write-only output tape

$M(\alpha, k)$ denotes the output written after reading the first $k$ letters of the input sequence $\alpha$

$M$ computes $f$ if for all $\alpha \in \text{dom}(f)$:

- $\forall k: M(\alpha, k)$ is a prefix of $f(\alpha)$, and
- $\forall i \exists j: |M(\alpha, j)| \geq i$
A function $f : \Sigma^\omega \rightarrow \Gamma^\omega$ is continuous at $\alpha \in \text{dom}(f)$ if
\[ \forall i \exists j \forall \beta \in \text{dom}(f) : |\alpha \land \beta| \geq j \implies |f(\alpha) \land f(\beta)| \geq i. \]

$f$ is continuous if it is continuous at every $\alpha \in \text{dom}(f)$.

**Examples.**

- $f_1 : uA\alpha \mapsto \rightarrow A |u| \alpha uB\alpha \mapsto \rightarrow B |u| \alpha$, for all $u \in \{a, b\}^*$, $\alpha \in \{a, b\}^\omega$ is continuous.

- $f_2 : \alpha \mapsto \rightarrow \{a^\omega\text{ if }\alpha\text{ contains }\infty\text{ many }a\} b^\omega\text{ otherwise}$ for all $\alpha \in \{a, b\}^\omega$ is not continuous.

If $f : \Sigma^\omega \rightarrow \Gamma^\omega$ is computable, then it is continuous, the converse does not hold.
A function $f : \Sigma^\omega \rightarrow \Gamma^\omega$ is **continuous** at $\alpha \in \text{dom}(f)$ if

$\forall i \exists j \forall \beta \in \text{dom}(f): |\alpha \land \beta| \geq j$ implies $|f(\alpha) \land f(\beta)| \geq i$. 
Computability and Continuity

A function $f : \Sigma^\omega \rightarrow \Gamma^\omega$ is **continuous** at $\alpha \in \text{dom}(f)$ if

$\forall i \exists j \forall \beta \in \text{dom}(f) : |\alpha \land \beta| \geq j$ implies $|f(\alpha) \land f(\beta)| \geq i$.

$f$ is **continuous** if it is continuous at every $\alpha \in \text{dom}(f)$. 

Examples.

- $f_1 : uA\alpha \mapsto A|u| \quad uB\alpha \mapsto B|u|$ for all $u \in \{a,b\}^*$, $\alpha \in \{a,b\}^\omega$ is continuous

- $f_2 : \alpha \mapsto \begin{cases} a^\omega & \text{if } \alpha \text{ contains } \infty \text{ many } a \\ b^\omega & \text{otherwise} \end{cases}$ for all $\alpha \in \{a,b\}^\omega$ is not continuous

- If $f : \Sigma^\omega \rightarrow \Gamma^\omega$ is computable, then it is continuous, the converse does not hold.
A function \( f : \Sigma^\omega \rightarrow \Gamma^\omega \) is continuous at \( \alpha \in \text{dom}(f) \) if
\[ \forall i \exists j \ \forall \beta \in \text{dom}(f) : |\alpha \land \beta| \geq j \text{ implies } |f(\alpha) \land f(\beta)| \geq i. \]
f is continuous if it is continuous at every \( \alpha \in \text{dom}(f) \).

Examples.
\[ f_1 : uA\alpha \mapsto A^{|u|}\alpha \quad uB\alpha \mapsto B^{|u|}\alpha, \]
for all \( u \in \{a, b\}^* \), \( \alpha \in \{a, b\}^\omega \) is continuous.
Computability and Continuity

A function $f : \Sigma^\omega \rightarrow \Gamma^\omega$ is **continuous** at $\alpha \in \text{dom}(f)$ if

$\forall i \exists j \forall \beta \in \text{dom}(f) : |\alpha \wedge \beta| \geq j$ implies $|f(\alpha) \wedge f(\beta)| \geq i$.

$f$ is **continuous** if it is continuous at every $\alpha \in \text{dom}(f)$.

**Examples.**

- $f_1 : uA\alpha \mapsto A|u|\alpha$  
  $uB\alpha \mapsto B|u|\alpha$,  
  for all $u \in \{a, b\}^*$, $\alpha \in \{a, b\}^\omega$ is continuous

- $f_2 : \alpha \mapsto \begin{cases} 
  a^\omega & \text{if } \alpha \text{ contains } \infty \text{ many } a \\
  b^\omega & \text{otherwise}
\end{cases}$  
  for all $\alpha \in \{a, b\}^\omega$ is not continuous
A function $f : \Sigma^\omega \rightarrow \Gamma^\omega$ is **continuous** at $\alpha \in \text{dom}(f)$ if

\begin{itemize}
  \item \( \forall i \ \exists j \ \forall \beta \in \text{dom}(f) : |\alpha \land \beta| \geq j \) implies \( |f(\alpha) \land f(\beta)| \geq i \).
\end{itemize}

\( f \) is **continuous** if it is continuous at every $\alpha \in \text{dom}(f)$.

**Examples.**

- \( f_1 : uA\alpha \mapsto A^{\lfloor u \rfloor} \alpha \quad uB\alpha \mapsto B^{\lfloor u \rfloor} \alpha \), for all $u \in \{a, b\}^*$, $\alpha \in \{a, b\}^\omega$ is continuous

- \( f_2 : \alpha \mapsto \begin{cases} a^\omega & \text{if } \alpha \text{ contains } \infty \text{ many } a \\ b^\omega & \text{otherwise} \end{cases} \)
  for all $\alpha \in \{a, b\}^\omega$ is not continuous

- If $f : \Sigma^\omega \rightarrow \Gamma^\omega$ is computable, then it is continuous,
- the converse does not hold.
Computability and Continuity

![Diagram showing the relationship between continuous functions, computable functions, regular functions, rational functions, synchronous functions, and Mealy machines.](image-url)
Total vs. Partial Domain

In synthesis, often a total specification domain is assumed, else the synthesis task fails by design.

Here: We allow partial domain

Example. Specification: contains pairs of the form $(uA\alpha,A | u|\beta)$ $(uB\alpha,B | u|\beta)$, where $u \in \{a,b\}^*$, $\alpha,\beta \in \{a,b\}^\omega$, $A,B$ are special letters

▶ has partial domain $\{a,b\}^* \{A,B\} \{a,b\}^\omega$

▶ e.g., implemented by a deterministic machine that computes the function $uA\alpha \mapsto A | u|\alpha uB\alpha \mapsto B | u|\alpha$

▶ There is no way to complete the domain and remain implementable!
Total vs. Partial Domain

- In synthesis, often a total specification domain is assumed, else the synthesis task fails by design.
Total vs. Partial Domain

- In synthesis, often a total specification domain is assumed, else the synthesis task fails by design
- Here: We allow partial domain

Example:
- Specification: contains pairs of the form \((uA\alpha, A | u | \beta) (uB\alpha, B | u | \beta)\), where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega\), \(A, B\) are special letters
- Has partial domain \(\{a, b\}^* \{A, B\} \{a, b\}^\omega\)
- E.g., implemented by a deterministic machine that computes the function \(uA\alpha \mapsto A | u | \alpha \ uB\alpha \mapsto B | u | \alpha\)
- There is no way to complete the domain and remain implementable!
In synthesis, often a total specification domain is assumed, else the synthesis task fails by design.

Here: We allow partial domain

**Example.**

- Specification: contains pairs of the form

  \[(uA\alpha, A|u|\beta) \quad (uB\alpha, B|u|\beta),\]

  where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega, A, B\) are special letters

- has partial domain \(\{a, b\}^*\{A, B\}\{a, b\}^\omega\)

- e.g., implemented by a deterministic machine that computes the function \(uA\alpha \mapsto A|u|\alpha \quad uB\alpha \mapsto B|u|\alpha\)
In synthesis, often a total specification domain is assumed, else the synthesis task fails by design.

Here: We allow partial domain.

Example.

Specification: contains pairs of the form

\[(uA\alpha, A|u|\beta) \quad (uB\alpha, B|u|\beta),\]

where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega, A, B\) are special letters.

Has partial domain \(\{a, b\}^*\{A, B\}\{a, b\}^\omega\)

E.g., implemented by a deterministic machine that computes the function \(uA\alpha \mapsto A|u|\alpha\) \(uB\alpha \mapsto B|u|\alpha\)

There is no way to complete the domain and remain implementable!
Results for Total Domain

Theorem (Holtmann/Kaiser/Thomas'10). It is decidable in $2EXPTIME$ whether a continuous function can be synthesized from a given synchronous relation with total domain.

Theorem (Klein/Zimmermann'14). It is $EXPTime$-complete to decide whether a continuous function can be synthesized from a given synchronous relation with total domain.
Theorem (Holtmann/Kaiser/Thomas’10). It is decidable in $2\text{EXPTIME}$ whether a continuous function can be synthesized from a given synchronous relation with total domain.
Results for Total Domain

**Theorem (Holtmann/Kaiser/Thomas’10).** It is decidable in 2EXP-TIME whether a continuous function can be synthesized from a given synchronous relation with total domain.

**Theorem (Klein/Zimmermann’14).** It is EXPTIME-complete to decide whether a continuous function can be synthesized from a given synchronous relation with total domain.
Results for Total Domain

**Theorem (Holtmann/Kaiser/Thomas’10).** It is decidable in 2EXPTIME whether a continuous function can be synthesized from a given synchronous relation with *total domain*.

**Theorem (Klein/Zimmermann’14).** It is EXPTIME-complete to decide whether a continuous function can be synthesized from a given synchronous relation with *total domain*.

Is the function computable?
Implementations for Total Domain

Such a synthesized function is computable by a sequential transducer. A transducer is sequential if its underlying input automaton is a DFA.

Example.

\[
\begin{array}{cccc}
& a & \varepsilon \\
0 & a & b \\
 & b & b \\
1 & \varepsilon & c \\
2 & a & a \\
 & a & aab
\end{array}
\]

(asynchronous) transducer

\[
\begin{array}{cccc}
& a & \varepsilon \\
 & c & a \\
 & b & \varepsilon \\
 & a & c \\
 & a & aab
\end{array}
\]

sequential transducer
Theorem (Holtmann/Kaiser/Thomas’10). Such a synthesized function is computable by a sequential transducer.
Implementations for Total Domain

**Theorem (Holtmann/Kaiser/Thomas'10).** Such a synthesized function is computable by a sequential transducer.

A transducer is **sequential** if its underlying input automaton is a DFA.
Theorem (Holtmann/Kaiser/Thomas’10). Such a synthesized function is computable by a sequential transducer.

A transducer is **sequential** if its underlying input automaton is a DFA.

**Example.**

```
0 ---- ba/ε ----> 0
  |                |                |
  v                v                v
  0 ----> b/aab    0 ----> b/ε
  |                        |
  v                        v
  0                        0
```

(asynchronous) transducer
**Theorem** (Holtmann/Kaiser/Thomas'10). Such a synthesized function is computable by a sequential transducer.

A transducer is **sequential** if its underlying input automaton is a DFA.

**Example.**

- (asynchronous) transducer

- sequential transducer
Results for Partial Domain

It is EXPTime-complete to decide whether a continuous function can be synthesized from a given synchronous relation with partial domain. Such a synthesized function is computable.
Results for Partial Domain

**Theorem** (Filiot/W.). It is EXPTIME-complete to decide whether a continuous function can be synthesized from a given synchronous relation with *partial domain*. Such a synthesized function is computable.
Proof Idea

Adam plays input letters

Eve plays output letters

If the input sequence is in the specification domain, input + output sequence must be in relation wrt the specification

Problem

Eve might need an unbounded lookahead on Adams moves

We want a finite game arena, cannot store the lookahead explicitly

Solution

Instead of an explicit lookahead, store a finite abstraction
Proof Idea

**Game view**

- Adam plays input letters

- Eve plays output letters

If the input sequence is in the specification domain, input + output sequence must be in relation wrt the specification.

Problem

- Eve might need an unbounded lookahead on Adams moves

- We want a finite game arena, cannot store the lookahead explicitly

Solution

- Instead of an explicit lookahead, store a finite abstraction
Proof Idea

**Game view**

- Adam plays input letters
- Eve plays output letters

If the input sequence is in the specification domain, the input + output sequence must be in relation with respect to the specification.

Problem

- Eve might need an unbounded lookahead on Adams moves
- We want a finite game arena, cannot store the lookahead explicitly

Solution

- Instead of an explicit lookahead, store a finite abstraction
Proof Idea

Game view

- Adam plays input letters
- Eve plays output letters
- If the input sequence is in the specification domain, input + output sequence must be in relation wrt the specification
Proof Idea

**Game view**

- Adam plays input letters
- Eve plays output letters
- If the input sequence is in the specification domain, input + output sequence must be in relation wrt the specification

**Problem**
Proof Idea

**Game view**
- Adam plays input letters
- Eve plays output letters
- If the input sequence is in the specification domain, input + output sequence must be in relation wrt the specification

**Problem**
- Eve might need an unbounded lookahead on Adams moves
Proof Idea

**Game view**
- Adam plays input letters
- Eve plays output letters
- If the input sequence is in the specification domain, input + output sequence must be in relation wrt the specification

**Problem**
- Eve might need an unbounded lookahead on Adams moves
- We want a finite game arena, cannot store the lookahead explicitly
Proof Idea

**Game view**

- Adam plays input letters
- Eve plays output letters
- If the input sequence is in the specification domain, input + output sequence must be in relation wrt the specification

**Problem**

- Eve might need an unbounded lookahead on Adams moves
- We want a finite game arena, cannot store the lookahead explicitly

**Solution**
Proof Idea

**Game view**

- Adam plays input letters
- Eve plays output letters
- If the input sequence is in the specification domain, input + output sequence must be in relation wrt the specification

**Problem**

- Eve might need an unbounded lookahead on Adams moves
- We want a finite game arena, cannot store the lookahead explicitly

**Solution**

- Instead of an explicit lookahead, store a finite abstraction
Proof Idea

Given a finite input word \( u \in \Sigma^* \), its profile \( P_u \) stores all inducible state transformations with respect to the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles.
- Eve can delay her move or choose a state transformation from a lookahead profile (instead of playing output letters).
Proof Idea

Given a finite input word $u \in \Sigma^*$, its profile $P_u$ stores all inducible state transformations wrt the specification automaton.
Proof Idea

Given a finite input word $u \in \Sigma^*$, its **profile** $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea
Proof Idea

Given a finite input word $u \in \Sigma^*$, its profile $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles
Proof Idea

Given a finite input word $u \in \Sigma^*$, its profile $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles
- Eve can delay her her move, or chose a state transformation from a lookahead profile (instead of playing output letters)
Proof Idea

Given a finite input word $u \in \Sigma^*$, its profile $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or choose a state transformation from a lookahead profile (instead of playing output letters)

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_1 \in P_{u_1}$</td>
<td>$\lambda_2 \in P_{u_2}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Proof Idea

Given a finite input word \( u \in \Sigma^* \), its **profile** \( P_u \) stores all inducible state transformations wrt the specification automaton.

**Game Idea**

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or chose a state transformation from a lookahead profile (instead of playing output letters)

\[
\begin{array}{c|c|c|c}
  u_1 & u_2 & u_3 & u_4 \cdot a \\
\end{array}
\]

\[
\begin{array}{c|c}
  \lambda_1 \in P_{u_1} & \lambda_2 \in P_{u_2} \\
\end{array}
\]
Proof Idea

Given a finite input word $u \in \Sigma^*$, its **profile** $P_u$ stores all inducible state transformations wrt the specification automaton.

**Game Idea**

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or choose a state transformation from a lookahead profile (instead of playing output letters)

\[
\begin{array}{cccc}
  u_1 & u_2 & u_3 & u_4 \cdot aa \\
\end{array}
\]

\[
\begin{array}{cc}
  \lambda_1 \in P_{u_1} & \lambda_2 \in P_{u_2} \\
\end{array}
\]
Proof Idea

Given a finite input word \( u \in \Sigma^* \), its **profile** \( P_u \) stores all inducible state transformations wrt the specification automaton.

**Game Idea**

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or chose a state transformation from a lookahead profile (instead of playing output letters)

\[
\begin{array}{cccc}
u_1 & u_2 & u_3 & u_4 \cdot aaa \\
\hline
\lambda_1 \in P_{u_1} & \lambda_2 \in P_{u_2} & \\
\end{array}
\]
Proof Idea

Given a finite input word $u \in \Sigma^*$, its profile $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or chose a state transformation from a lookahead profile (instead of playing output letters)

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4 \cdot aaa$</th>
</tr>
</thead>
</table>

| $\lambda_1 \in P_{u_1}$ | $\lambda_2 \in P_{u_2}$ | $\lambda_3 \in P_{u_3}$ |
Proof Idea

Given a finite input word $u \in \Sigma^*$, its profile $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or chose a state transformation from a lookahead profile (instead of playing output letters)

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4 \cdot aaaa \cdot a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_1 \in P_{u_1}$</td>
<td>$\lambda_2 \in P_{u_2}$</td>
<td>$\lambda_3 \in P_{u_3}$</td>
<td></td>
</tr>
</tbody>
</table>
Proof Idea

Given a finite input word $u \in \Sigma^*$, its **profile** $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or choose a state transformation from a lookahead profile (instead of playing output letters)

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4 \cdot aaa$</th>
<th>$a \cdots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_1 \in P_{u_1}$</td>
<td>$\lambda_2 \in P_{u_2}$</td>
<td>$\lambda_3 \in P_{u_3}$</td>
<td>$\lambda_4 \in \cdots$</td>
<td></td>
</tr>
</tbody>
</table>
Proof Idea

Given a finite input word $u \in \Sigma^*$, its profile $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or choose a state transformation from a lookahead profile (instead of playing output letters)

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4 \cdot aaa$</th>
<th>$a \cdots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_1 \in P_{u_1}$</td>
<td>$\lambda_2 \in P_{u_2}$</td>
<td>$\lambda_3 \in P_{u_3}$</td>
<td>$\lambda_4 \in \cdots$</td>
<td></td>
</tr>
</tbody>
</table>

Winning condition If Adam plays a valid input sequence,
Proof Idea

Given a finite input word $u \in \Sigma^*$, its profile $P_u$ stores all inducible state transformations wrt the specification automaton.

Game Idea

- Adam plays input letters, building lookahead profiles
- Eve can delay her move, or choose a state transformation from a lookahead profile (instead of playing output letters)

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4 \cdot aaa$</th>
<th>$a \cdots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_1 \in P_{u_1}$</td>
<td>$\lambda_2 \in P_{u_2}$</td>
<td>$\lambda_3 \in P_{u_3}$</td>
<td>$\lambda_4 \in \cdots$</td>
<td></td>
</tr>
</tbody>
</table>

Winning condition

If Adam plays a valid input sequence,
- Eve makes a move infinitely often,
- her moves describe an accepting run wrt the specification.
Implementations for Partial Domain

If a synchronous relation with partial domain is implementable, then it can be implemented by a deterministic two-way transducer.

Example. Specification: contains pairs of the form $(uA\alpha,A\mid u\mid \beta)\ (uB\alpha,B\mid u\mid \alpha)$, where $u \in \{a,b\}^*$, $\alpha,\beta \in \{a,b\}^\omega$, $A,B$ are special letters.

- e.g., implemented by a deterministic two-way transducer that computes $uA\alpha \mapsto A\mid u\mid \alpha\ uB\alpha \mapsto B\mid u\mid \alpha$.

- The transducer goes right until $A$ resp. $B$ is read, no output.

- Then goes back left to the beginning, no output.

- Then goes right, outputs $A$ resp. $B$ for every letter until $A$ resp. $B$ is read.

- Then goes right and copies the input.
**Theorem (Filiot/W.).** If a synchronous relation with partial domain is implementable, then it can be implemented by a deterministic two-way transducer.
Theorem (Filiot/W.). If a synchronous relation with partial domain is implementable, then it can be implemented by a deterministic two-way transducer.

Example.

▶ Specification: contains pairs of the form

\[(uA\alpha, A^{|u|}\beta) \quad (uB\alpha, B^{|u|}\beta),\]

where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega, A, B\) are special letters

▶ e.g., implemented by a deterministic two-way transducer that computes \(uA\alpha \mapsto A^{|u|}\alpha \quad uB\alpha \mapsto B^{|u|}\alpha\)
Implementations for Partial Domain

**Theorem (Filiot/W.).** If a synchronous relation with partial domain is implementable, then it can be implemented by a deterministic two-way transducer.

**Example.**

- Specification: contains pairs of the form
  
  \[(u A \alpha, A^{\mid u \mid} \beta) \ (u B \alpha, B^{\mid u \mid} \beta),\]

  where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega, A, B\) are special letters

  - e.g., implemented by a deterministic two-way transducer that computes \(u A \alpha \mapsto A^{\mid u \mid} \alpha \ u B \alpha \mapsto B^{\mid u \mid} \alpha\)

  - transducer goes right until \(A\) resp. \(B\) is read, no output
Implementations for Partial Domain

**Theorem (Filiot/W.).** If a synchronous relation with partial domain is implementable, then it can be implemented by a deterministic two-way transducer.

**Example.**

- Specification: contains pairs of the form
  \[(uA\alpha, A|u|\beta) \quad (uB\alpha, B|u|\beta),\]

  where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega, A, B\) are special letters

- e.g., implemented by a deterministic two-way transducer that computes
  \[uA\alpha \mapsto A|u|\alpha \quad uB\alpha \mapsto B|u|\alpha\]

  - transducer goes right until \(A\) resp. \(B\) is read, no output
  - goes back left to the beginning, no output
Implementations for Partial Domain

**Theorem (Filiot/W.).** If a synchronous relation with partial domain is implementable, then it can be implemented by a deterministic two-way transducer.

**Example.**

- Specification: contains pairs of the form
  \[(uA\alpha, A|u|\beta) \quad (uB\alpha, B|u|\beta),\]
  where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega, A, B\) are special letters
- e.g., implemented by a deterministic two-way transducer that computes
  \(uA\alpha \mapsto A|u|\alpha \quad uB\alpha \mapsto B|u|\alpha\)
  - transducer goes right until \(A\) resp. \(B\) is read, no output
  - goes back left to the beginning, no output
  - goes right, outputs \(A\) resp. \(B\) for every letter until \(A\) resp. \(B\) is read,
Implementations for Partial Domain

**Theorem (Filiot/W.).** If a synchronous relation with partial domain is implementable, then it can be implemented by a deterministic two-way transducer.

**Example.**
- Specification: contains pairs of the form
  
  \[(uA\alpha, A^{|u|}\beta) \quad (uB\alpha, B^{|u|}\beta),\]

  where \(u \in \{a, b\}^*, \alpha, \beta \in \{a, b\}^\omega\), \(A, B\) are special letters

- e.g., implemented by a deterministic two-way transducer that computes \(uA\alpha \mapsto A^{|u|}\alpha \quad uB\alpha \mapsto B^{|u|}\alpha\)
  - transducer goes right until \(A\) resp. \(B\) is read, no output
  - goes back left to the beginning, no output
  - goes right, outputs \(A\) resp. \(B\) for every letter until \(A\) resp. \(B\) is read,
  - goes right and copies the input
Total vs. Partial Domain Implementations

Total domain
- Sequential transducers with bounded lookahead suffice
- Intuitive reason for bounded lookahead: If an arbitrary long lookahead is needed to determine the next output, then a deterministic machine may wait forever to output something valid.
- Result: a finite output sequence, but the infinite input sequence is valid.

Partial domain
- Deterministic two-way transducers suffice, sequential transducers do not
- Unbounded lookahead may be necessary.
Total vs. Partial Domain Implementations

**Total domain**

- Sequential transducers with bounded lookahead suffice
Total vs. Partial Domain Implementations

**Total domain**

- Sequential transducers with bounded lookahead suffice
- Intuitive reason for bounded lookahead
Total vs. Partial Domain Implementations

**Total domain**

- Sequential transducers with bounded lookahead suffice
- Intuitive reason for bounded lookahead
  - If an arbitrary long lookahead is needed to determine the next output,
**Total vs. Partial Domain Implementations**

**Total domain**

- Sequential transducers with bounded lookahead suffice
- Intuitive reason for bounded lookahead
  - If an arbitrary long lookahead is needed to determine the next output,
  - then a deterministic machine may wait forever to output something valid.

**Partial domain**

- Deterministic two-way transducers suffice, sequential transducers do not
- Unbounded lookahead may be necessary
Total vs. Partial Domain Implementations

**Total domain**
- Sequential transducers with bounded lookahead suffice
- Intuitive reason for bounded lookahead
  - If an arbitrary long lookahead is needed to determine the next output,
  - then a deterministic machine may wait forever to output something valid.
- Result: a finite output sequence, but the infinite input sequence is valid ✗
Total vs. Partial Domain Implementations

**Total domain**
- Sequential transducers with bounded lookahead suffice
- Intuitive reason for bounded lookahead
  - If an arbitrary long lookahead is needed to determine the next output,
  - then a deterministic machine may wait forever to output something valid.
- Result: a finite output sequence, but the infinite input sequence is valid.

**Partial domain**
- Deterministic two-way transducers suffice, sequential transducers do not
Total vs. Partial Domain Implementations

**Total domain**
- Sequential transducers with bounded lookahead suffice
- Intuitive reason for bounded lookahead
  - If an arbitrary long lookahead is needed to determine the next output,
  - then a deterministic machine may wait forever to output something valid.
- Result: a finite output sequence, but the infinite input sequence is valid

**Partial domain**
- Deterministic two-way transducers suffice, sequential transducers do not
- Unbounded lookahead may be necessary
Summary

Starting from a specification given by a non-deterministic automaton

Starting from a specification given by a deterministic automaton

Implementations for total domain
▶ sequential transducers suffice
▶ bounded lookahead suffices

Implementations for partial domain
▶ deterministic two-way transducers suffice
▶ unbounded lookahead may be necessary
## Summary

<table>
<thead>
<tr>
<th>Spec</th>
<th>Impl</th>
<th>Mealy machine</th>
<th>computable</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous w/ total domain</td>
<td>EXPTIME-c¹</td>
<td>EXPTIME-c²</td>
<td></td>
</tr>
<tr>
<td>synchronous w/ partial domain</td>
<td>EXPTIME-c¹</td>
<td>EXPTIME-c²</td>
<td></td>
</tr>
</tbody>
</table>

1. Starting from a specification given by a non-deterministic automaton
2. Starting from a specification given by a deterministic automaton
### Summary

<table>
<thead>
<tr>
<th>Spec</th>
<th>Impl</th>
<th>Mealy machine</th>
<th>computable</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous w/ total domain</td>
<td>EXPTIME-c(^1)</td>
<td>EXPTIME-c(^2)</td>
<td></td>
</tr>
<tr>
<td>synchronous w/ partial domain</td>
<td>EXPTIME-c(^1)</td>
<td>EXPTIME-c(^2)</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) Starting from a specification given by a non-deterministic automaton  
\(^2\) Starting from a specification given by a deterministic automaton

- Implementations for total domain
  - sequential transducers suffice
  - bounded lookahead suffices
### Summary

<table>
<thead>
<tr>
<th>Spec</th>
<th>Impl</th>
<th>Mealy machine</th>
<th>computable</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous w/ total domain</td>
<td>EXPTIME-c&lt;sub&gt;1&lt;/sub&gt;</td>
<td>EXPTIME-c&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>synchronous w/ partial domain</td>
<td>EXPTIME-c&lt;sub&gt;1&lt;/sub&gt;</td>
<td>EXPTIME-c&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
</tr>
</tbody>
</table>

1. Starting from a specification given by a non-deterministic automaton
2. Starting from a specification given by a deterministic automaton

- Implementations for total domain
  - sequential transducers suffice
  - bounded lookahead suffices
- Implementations for partial domain
  - deterministic two-way transducers suffice
  - unbounded lookahead may be necessary
Going Beyond Synchronous Specifications

It is decidable whether a synchronous specification can be implemented.

What about more powerful specifications?

Theorem (Filiot/W.). It is undecidable whether a given rational relation can be implemented.

Theorem (Filiot/W.). It is undecidable whether a continuous, computable, resp., sequential function can be synthesized from a given rational relation.

Finite word setting: Undecidable whether a sequential function can be synthesized. (Carayol/Löding’14)
Going Beyond Synchronous Specifications

▶ It is decidable whether a synchronous specification can be implemented.
Going Beyond Synchronous Specifications

- It is decidable whether a synchronous specification can be implemented.
- What about more powerful specifications?

Theorem (Filiot/W.). It is undecidable whether a given rational relation can be implemented.

Theorem (Filiot/W.). It is undecidable whether a continuous, computable, resp., sequential function can be synthesized from a given rational relation.

- Finite word setting: Undecidable whether a sequential function can be synthesized. (Carayol/Löding'14)
Going Beyond Synchronous Specifications

- It is decidable whether a synchronous specification can be implemented.
- What about more powerful specifications?

**Theorem (Filiot/W.).** It is undecidable whether a given rational relation can be implemented.
Going Beyond Synchronous Specifications

- It is decidable whether a synchronous specification can be implemented.
- What about more powerful specifications?

**Theorem (Filiot/W.).** It is undecidable whether a given rational relation can be implemented.

**Theorem (Filiot/W.).** It is undecidable whether a continuous, computable, resp., sequential function can be synthesized from a given rational relation.
Going Beyond Synchronous Specifications

- It is decidable whether a synchronous specification can be implemented.
- What about more powerful specifications?

**Theorem (Filiot/W.).** It is undecidable whether a given rational relation can be implemented.

**Theorem (Filiot/W.).** It is undecidable whether a continuous, computable, resp., sequential function can be synthesized from a given rational relation.

- Finite word setting: Undecidable whether a sequential function can be synthesized. (Carayol/Löding’14)
Undecidability Proof (similar to finite word setting)

Reduction from Post’s Correspondence Problem
Undecidability Proof (similar to finite word setting)

Reduction from Post’s Correspondence Problem

- A PCP instance $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$. 

A PCP instance has no solution

- $i_1 \ldots i_m \alpha \mapsto u_{i_1} \ldots u_{i_m} \beta$ if $\alpha$ contains $\infty$ many $a$

- $\not\mapsto v_{i_1} \ldots v_{i_m} \beta$ otherwise

With $i_1 \ldots i_m \in \{1, \ldots, n\}^*$ and $\alpha, \beta \in \{a, b\}^\omega$.

A PCP instance has a solution

- no implementation exists

- never known whether the input sequence has $\infty$ many $a$
Undecidability Proof (similar to finite word setting)

Reduction from Post’s Correspondence Problem

- A PCP instance $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$.
- Rational relation with domain $\{1, \ldots, n\}^* \{a, b\}^\omega$ and pairs $i_1 \cdots i_m\alpha \begin{cases} \hookrightarrow u_{i_1} \cdots u_{i_m}\beta & \text{if } \alpha \text{ contains } \infty \text{ many } a \\ \not\hookrightarrow v_{i_1} \cdots v_{i_m}\beta & \text{otherwise} \end{cases}$

with $i_1 \cdots i_m \in \{1, \ldots, n\}^*$ and $\alpha, \beta \in \{a, b\}^\omega$. 
Undecidability Proof (similar to finite word setting)

Reduction from Post’s Correspondence Problem

- A PCP instance $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$.
- Rational relation with domain $\{1, \ldots, n\}^* \{a, b\}^\omega$ and pairs
  
  $i_1 \cdots i_m \alpha \begin{cases} 
  \mapsto u_{i_1} \cdots u_{i_m} \beta & \text{if } \alpha \text{ contains } \infty \text{ many } a \\ 
  \not\mapsto v_{i_1} \cdots v_{i_m} \beta & \text{otherwise}
  \end{cases}$

  with $i_1 \cdots i_m \in \{1, \ldots, n\}^*$ and $\alpha, \beta \in \{a, b\}^\omega$.

PCP instance has no solution

- $i_1 \cdots i_m \alpha \mapsto u_{i_1} \cdots u_{i_m} \alpha$ is an implementation
- always $u_{i_1} \cdots u_{i_m} \neq v_{i_1} \cdots v_{i_m}$
Undecidability Proof (similar to finite word setting)

Reduction from Post’s Correspondence Problem

- A PCP instance $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$.
- Rational relation with domain $\{1, \ldots, n\}^* \{a, b\}^\omega$ and pairs
  
  $\begin{align*}
  i_1 \cdot \cdot i_m \alpha &\begin{cases} 
  \rightarrow u_{i_1} \cdots u_{i_m} \beta & \text{if } \alpha \text{ contains } \infty \text{ many } a \\
  \not\rightarrow v_{i_1} \cdots v_{i_m} \beta & \text{otherwise}
  \end{cases}
  \end{align*}$

  with $i_1 \cdots i_m \in \{1, \ldots, n\}^*$ and $\alpha, \beta \in \{a, b\}^\omega$.

PCP instance has no solution

- $i_1 \cdots i_m \alpha \rightarrow u_{i_1} \cdots u_{i_m} \alpha$ is an implementation
- always $u_{i_1} \cdots u_{i_m} \neq v_{i_1} \cdots v_{i_m}$

PCP instance has a solution

- no implementation exists
- never known whether the input sequence has $\infty$ many $a$
Work in Progress: Deterministic Rational Relations

Class between synchronous and rational relations.

Recognized by special kind of transducers

- state set is partitioned into input and output states
- transition function:

\[ Q_i \times \Sigma \rightarrow Q \cup Q_o \times \Gamma \rightarrow Q \]

Example.

\[
\begin{array}{ccc}
  & a/\varepsilon & b/\varepsilon \\
\hline
a/\varepsilon & \varepsilon/a & \varepsilon/b \\
\end{array}
\]

- recognizes \( f: u \# \alpha \mapsto \alpha, u \in \{a,b\}^*, \alpha \in \{a,b\} \omega \)

- \( f \) is not synchronous
Work in Progress: Deterministic Rational Relations

Class between synchronous and rational relations.
Work in Progress: Deterministic Rational Relations

Class between synchronous and rational relations.

Recognized by special kind of transducers

- state set is partitioned into input and output states
Class between synchronous and rational relations.

Recognized by special kind of transducers

- state set is partitioned into input and output states
- transition function: $Q_i \times \Sigma \rightarrow Q \cup Q_o \times \Gamma \rightarrow Q$
Work in Progress: Deterministic Rational Relations

Class between synchronous and rational relations.

Recognized by special kind of transducers

- state set is partitioned into input and output states
- transition function: \( Q_i \times \Sigma \to Q \cup Q_o \times \Gamma \to Q \)

Example.

![Diagram of a deterministic rational relation]

\[
\begin{array}{c}
Q_0 \\
\downarrow \\
0 \\
\downarrow \\
1 \\
\hline
a/\varepsilon & a/\varepsilon & \varepsilon/a \\
\hline \\
\#/\varepsilon & b/\varepsilon & \varepsilon/b \\
\hline \\
b/\varepsilon & \varepsilon/b & \varepsilon/a \\
\hline \\
\end{array}
\]

Example function:

\[ f: u \# \alpha \mapsto \alpha, u \in \{a, b\}^*, \alpha \in \{a, b\}^\omega \]

\( f \) is not synchronous.
Work in Progress: Deterministic Rational Relations

Class between synchronous and rational relations.

Recognized by special kind of transducers

- state set is partitioned into input and output states
- transition function: \( Q_i \times \Sigma \rightarrow Q \cup Q_o \times \Gamma \rightarrow Q \)

Example.

recognizes \( f : u \# \alpha \mapsto \alpha, \quad u \in \{a, b\}^*, \alpha \in \{a, b\}^\omega \)

\( f \) is not synchronous
Almost Sure Theorem. It is decidable whether a continuous function can be synthesized from a given deterministic rational relation. Almost Sure Theorem. Such a synthesized function is computable by a deterministic two-way transducer.
Almost Sure Theorem. It is decidable whether a continuous function can be synthesized from a given deterministic rational relation.
**Almost Sure Theorem.** It is decidable whether a continuous function can be synthesized from a given deterministic rational relation.

**Almost Sure Theorem.** Such a synthesized function is computable by a deterministic two-way transducer.
Open question
Open question

Is it decidable whether a synchronous relation with **partial domain** is implementable using only finite memory?
Open question

Is it decidable whether a synchronous relation with partial domain is implementable using only finite memory?

Example.

- Specification: \((a^*b\cdots, b\cdots) \rightarrow (a^*c\cdots, c\cdots)\)
Open question

Is it decidable whether a synchronous relation with \textbf{partial domain} is implementable using only finite memory?

\textbf{Example.}

▶ Specification: \((a^*b\cdots, b\cdots) \rightarrow (a^*c\cdots, c\cdots)\)

▶ Specification is implementable, e.g., by a finite-memory machine (sequential transducer) that computes the function

\[a^*b\cdots \mapsto b^\omega \quad a^*c\cdots \mapsto c^\omega\]
<table>
<thead>
<tr>
<th>Spec</th>
<th>Impl</th>
<th>Mealy machine</th>
<th>sequential transducer</th>
<th>computable</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous w/ total domain</td>
<td>EXPTIME-c(^1)</td>
<td>EXPTIME-c(^2)</td>
<td>EXPTIME-c(^2)</td>
<td></td>
</tr>
<tr>
<td>synchronous w/ partial domain</td>
<td>EXPTIME-c(^1)</td>
<td>open</td>
<td>EXPTIME-c(^2)</td>
<td></td>
</tr>
<tr>
<td>det. rational</td>
<td>open</td>
<td>open</td>
<td>EXPTIME-c(^1)</td>
<td></td>
</tr>
<tr>
<td>rational</td>
<td>undecidable</td>
<td>undecidable</td>
<td>undecidable</td>
<td>undecidable</td>
</tr>
</tbody>
</table>

\(^1\) non-deterministic specification  \(^2\) deterministic specification