A little bit infinite?
Adding data to finitely labelled structures

Thomas Schwentick

Bordeaux
February 2008
Contents

Introduction

Motivation from XML
Motivation from Verification
Data Model
Automata
Logic
Other Models
Conclusion
Introduction

- Motivation from XML
  - Motivation from Verification
- Data Model
- Automata
- Logic
- Other Models
- Conclusion
### Composers from Southwest

<table>
<thead>
<tr>
<th>COMPOSERS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Birth</td>
<td>Death</td>
</tr>
<tr>
<td>Ravel</td>
<td>Ciboure</td>
<td>Paris</td>
</tr>
<tr>
<td>Tournemire</td>
<td>Bordeaux</td>
<td>Arcachon</td>
</tr>
</tbody>
</table>

### PIECES

<table>
<thead>
<tr>
<th>PIECES</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Comp</td>
<td>Year</td>
<td>Instr</td>
</tr>
<tr>
<td>Boléro</td>
<td>Ravel</td>
<td>1928</td>
<td>Orch.</td>
</tr>
<tr>
<td>Douze Préludes</td>
<td>Tournemire</td>
<td>1932</td>
<td>Piano</td>
</tr>
<tr>
<td>La Valse</td>
<td>Ravel</td>
<td>1920</td>
<td>Orch.</td>
</tr>
</tbody>
</table>

### SQL Query

```sql
SELECT B.Name, B.Comp
FROM Composers A, Pieces B
WHERE A.Name = B.Comp AND A.Birth = "Bordeaux"
```
Relational Databases

**Composers from Southwest**

<table>
<thead>
<tr>
<th>COMPOSERS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Birth</td>
<td>Death</td>
</tr>
<tr>
<td>Ravel</td>
<td>Ciboure</td>
<td>Paris</td>
</tr>
<tr>
<td>Tournemire</td>
<td>Bordeaux</td>
<td>Arcachon</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIECES</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Comp</td>
<td>Year</td>
<td>Instr</td>
<td>Movem</td>
<td></td>
</tr>
<tr>
<td>Boléro</td>
<td>Ravel</td>
<td>1928</td>
<td>Orch.</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Douze Préludes</td>
<td>Tournemire</td>
<td>1932</td>
<td>Piano</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>La Valse</td>
<td>Ravel</td>
<td>1920</td>
<td>Orch.</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**SELECT** B.Name, B.Comp  
**FROM** Composers A, Pieces B  
**WHERE** A.Name = B.Comp AND  
A.Birth = "Bordeaux"

- **Relational data**: flat structure & data
- Queries rely on **structure** and **equality of data items**:

\[
Q(x_1, x_2) \equiv \\
\exists x_3, \ldots, x_5, y_1 \ldots, y_3 \\
\text{Pieces}(x_1, x_2, x_3, x_4, x_5) \land \\
\text{Composers}(y_1, y_2, y_3) \land \\
y_1 = x_2 \land y_3 \text{ = "Bordeaux"}
\]

A little bit infinite? Thomas Schwentick
Relational Databases

Composers from Southwest

<table>
<thead>
<tr>
<th>COMPOSERS</th>
<th>Name</th>
<th>Birth</th>
<th>Death</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ravel</td>
<td>Ciboure</td>
<td>Paris</td>
</tr>
<tr>
<td></td>
<td>Tourlemoine</td>
<td>Bordeaux</td>
<td>Arcachon</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIECES</th>
<th>Name</th>
<th>Comp</th>
<th>Year</th>
<th>Instr</th>
<th>Movem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Boléro</td>
<td>Ravel</td>
<td>1928</td>
<td>Orch.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Douze Préludes</td>
<td>Tourlemoine</td>
<td>1932</td>
<td>Piano</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>La Valse</td>
<td>Ravel</td>
<td>1920</td>
<td>Orch.</td>
<td>1</td>
</tr>
</tbody>
</table>

SELECT B.Name, B.Comp
FROM Composers A, Pieces B
WHERE A.Name = B.Comp AND
      A.Birth = "Bordeaux"

- **Relational data**: flat structure & data
- Queries rely on *structure* and *equality of data items*:
  \[
  Q(x_1, x_2) \equiv \\
  \exists x_3, \ldots, x_5, y_1 \ldots, y_3 \\
  \text{Pieces}(x_1, x_2, x_3, x_4, x_5) \land \\
  \text{Composers}(y_1, y_2, y_3) \land \\
  y_1 = x_2 \land y_3 = "Bordeaux"
  \]
- Integrity Constraints rely on *structure* and *equality of data items*:
  \[
  \forall x_1, \ldots, x_5, y_1, \ldots, y_5 \\
  (x_1 = y_1 \land x_2 = y_2) \rightarrow \\
  (x_3 = y_3 \land x_4 = y_5 \land x_5 = y_5)
  \]
Example Document

(Composer) (Name) Maurice Ravel (/Name)
  (Vita) (Born) (When) March 3, 1875 (/When) (Where) Ciboure (/Where) (/Born)
  (Pieces)
    (Piece) (PTitle) Boléro (/PTitle) (PYear) 1928 (/PYear)
      (Instrumentation) Orchestra (/Instrumentation) (/Movements) 1 (/Movements) (/Piece)
    (Piece) (PTitle) La Valse (/PTitle) (PYear) 1920 (/PYear)
      (Instrumentation) Orchestra (/Instrumentation) (/Movements) 1 (/Movements) (/Piece)
  (/Pieces)
(Composer)
  (Composer) (Name) Charles Tournemire (/Name)
  (Vita) (Born) (When) January 22, 1870 (/When) (Where) Bordeaux (/Where) (/Born)
  (Died) (When) November 4, 1939 (/When) (Where) Arcachon (/Where) (/Died) (/Vita)
  (Pieces)
    (Piece) (PTitle) Douze préludes-poèmes (/PTitle) (PYear) 1932 (/PYear)
      (Instrumentation) Piano (/Instrumentation) (/Movements) 12 (/Movements) (/Piece)
  (/Pieces)
(Composer)
XML

Example Tree

Composer

Name
Maurice Ravel
Born
When
1875
Where
Ciboure
Vita
Died
When
1937
Where
Paris
Piece
PTitle
Boléro
PYear
1928
Instruments
Orchestra
Movements
1

Composer

Name
Charles Tournemire
Born
When
1870
Where
Bordeaux
Vita
Died
When
1939
Where
Arcachon
Piece
PTitle
Douze préludes poèmes
PYear
1932
Instruments
Piano
Movements
12

A little bit infinite? Thomas Schwentick
XML

Example

Composer

Name

Maurice Ravel

When

1875

Vita

Born

Where

Ciboure

Died

When

1937

Where

Paris

Piece

PTitle

Boléro

PY ear

1928

Instruments

Orchestra

Movements

1

Piece

PTitle

La Valse

PY ear

1920

Instruments

Orchestra

Movements

1

Composer

Name

Charles Tournemire

When

1870

Vita

Born

Where

Bordeaux

Died

When

1939

Where

Arcachon

Piece

PTitle

Douze préludes poèmes

PY ear

1932

Instruments

Piano

Movements

12

XML: hierarchical structure & data

A little bit infinite? Thomas Schwentick
XML

Example

- **XML:** hierarchical structure & data
- **Data model:** an XML document can be viewed as an unranked tree in which
  - inner nodes correspond to *elements*
  - leaves correspond to *data*
    (attributes, text content)

A little bit infinite? Thomas Schwentick
XML: hierarchical structure & data

Data model: an XML document can be viewed as an unranked tree in which
▶ inner nodes correspond to elements
▶ leaves correspond to data (attributes, text content)

For many investigations,
▶ the set of tags is restricted
▶ data values can be ignored
XML:

- **XML: hierarchical structure & data**
- **Data model:** an XML document can be viewed as an unranked tree in which
  - inner nodes correspond to **elements**
  - leaves correspond to **data** (attributes, text content)
- For many investigations,
  - the set of tags is restricted
  - data values can be ignored

→ **Abstraction:** labeled trees over a **finite** alphabet
XML

- **XML**: hierarchical structure & data
- **Data model**: an XML document can be viewed as an unranked tree in which
  - inner nodes correspond to **elements**
  - leaves correspond to **data**
    - (attributes, text content)
- For many investigations,
  - the set of tags is restricted
  - data values can be ignored
  ➞ **Abstraction**:
    - labeled trees over a **finite** alphabet
- Works well for foundational studies on many aspects of
  - Validation
  - Navigation
  - Transformation
XML

- **XML**: hierarchical structure & data
- **Data model**: an XML document can be viewed as an unranked tree in which
  - inner nodes correspond to **elements**
  - leaves correspond to **data**
    (attributes, text content)
- For many investigations,
  - the set of tags is restricted
  - data values can be ignored

→ **Abstraction**:
  labeled trees over a **finite** alphabet

- Works well for foundational studies on many aspects of
  - Validation
  - Navigation
  - Transformation

→ **Foundational research on XML** has largely ignored data but concentrated on finitely labeled trees

A little bit infinite?  Thomas Schwentick
There is a need for data-aware foundational XML research:
There is a need for data-aware foundational XML research:

- **Schemas:**
  - Schemas for XML describe the allowed *structure of documents* and can specify *constraints on the data*
  - **Structure constraints** can be captured by regular tree languages (automata & logics available)
  - **Data constraints** include uniqueness, keys, foreign keys
There is a need for data-aware foundational XML research:

- **Schemas:**
  - Schemas for XML describe the allowed structure of documents and can specify constraints on the data.
  - Structure constraints can be captured by regular tree languages (automata & logics available).
  - Data constraints include uniqueness, keys, foreign keys.

- **XPath:**
  - The core of XPath allows to specify navigational queries (automata & logics available).
  - But: it also allows comparisons between data.
There is a need for data-aware foundational XML research:

- **Schemas:**
  - Schemas for XML describe the allowed *structure of documents* and can specify *constraints on the data*
  - **Structure constraints** can be captured by regular tree languages (automata & logics available)
  - **Data constraints** include uniqueness, keys, foreign keys

- **XPath:**
  - The core of XPath allows to specify navigational queries (automata & logics available)
  - But: it also allows comparisons between data

- **Other data-aware processing tasks:**
  - Querying: XQuery
  - Transformations: XSLT
  - Data Exchange [Arenas, Libkin 05]
An example scenario: **XML Query optimization**

Algorithmic problem:
An example scenario: **XML Query optimization**

- Algorithmic problem:
  - Given XPath expressions $q_1, q_2$ and a schema $S$
  - Decide whether, for each valid document $d$ (wrt $S$):
    
    \[ q_1(d) \subseteq q_2(d) \]
An example scenario: **XML Query optimization**

- **Algorithmic problem:**
  - Given XPath expressions \( q_1, q_2 \) and a schema \( S \)
  - Decide whether, for each valid document \( d \) (wrt \( S \)):
    \[
    q_1(d) \subseteq q_2(d)
    \]

- The XPath queries might combine navigation with conditions on data values:
  - \( q_1 \): select all composers who wrote a piece in the year they died
  - \( q_2 \): select all composers whose name is unique
An example scenario: **XML Query optimization**

* Algorithmic problem:
  - Given XPath expressions \( q_1, q_2 \) and a schema \( S \)
  - Decide whether, for each valid document \( d \) (wrt \( S \)):
    \[
    q_1(d) \subseteq q_2(d)
    \]

* The XPath queries might combine navigation with conditions on data values:
  - \( q_1 \): select all composers who wrote a piece in the year they died
  - \( q_2 \): select all composers whose name is unique

* The schema \( S \) might consist of
  - structural constraints \( \rightarrow \) regular tree language \( L \)
  - and data integrity constraints
    (e.g.: each composer name occurs at most once)
An example scenario: XML Query optimization

Algorithmic problem:
- Given XPath expressions $q_1$, $q_2$ and a schema $S$
- Decide whether, for each valid document $d$ (wrt $S$):
  $$ q_1(d) \subseteq q_2(d) $$

The XPath queries might combine navigation with conditions on data values:
- $q_1$: select all composers who wrote a piece in the year they died
- $q_2$: select all composers whose name is unique

The schema $S$ might consist of
- structural constraints $\rightarrow$ regular tree language $L$
- and data integrity constraints
  (e.g.: each composer name occurs at most once)

Most of XPath navigation can be modelled by two-variable logic

How to deal with data?
Contents

Introduction
  Motivation from XML
  Motivation from Verification
    Data Model
    Automata
    Logic
    Other Models
    Conclusion
A Toy Example from Verification

A printer and two processes

- Possibility actions:
  - \( r_i \): User \( i \) submits print request
  - \( s_i \): Printing of request of \( i \) starts
  - \( t_i \): Print job for user \( i \) terminates

A little bit infinite? Thomas Schwentick
A Toy Example from Verification

A printer and two processes

- Example properties that could be enforced:
  - **“Local property”**: processes never request a new print job before the last one has terminated, i.e.: for each \(i\) the subrun is of the form \((r_i s_i t_i)^*\),
  - **“Global property”**: a print job must be finished before the next one is started, i.e.: between a \(s_i\) and the subsequent \(t_i\) there is no \(s_j\) or \(t_j, j \neq i\)

- A printer and two processes
- Possible actions:
  - \(r_i\): User \(i\) submits print request
  - \(s_i\): Printing of request of \(i\) starts
  - \(t_i\): Print job for user \(i\) terminates
A Toy Example from Verification

A printer and two processes

- Possible actions:
  - \( r_i \): User \( i \) submits print request
  - \( s_i \): Printing of request of \( i \) starts
  - \( t_i \): Print job for user \( i \) terminates

Example properties that could be enforced:
- "Local property": processes never request a new print job before the last one has terminated, i.e.: for each \( i \) the subrun is of the form \((r_is_it_i)^*\),
- "Global property": a print job must be finished before the next one is started, i.e.: between a \( s_i \) and the subsequent \( t_i \) there is no \( s_j \) or \( t_j, j \neq i \)

Memory Allocation

- "Local property": A memory location should only be accessed after it is allocated and before it is freed
A Toy Example from Verification

A printer and two processes

- Example properties that could be enforced:
  - "Local property": processes never request a new print job before the last one has terminated, i.e.: for each \( i \) the subrun is of the form \((r_i s_i t_i)^*\),
  - "Global property": a print job must be finished before the next one is started, i.e.: between a \( s_i \) and the subsequent \( t_i \) there is no \( s_j \) or \( t_j \), \( j \neq i \)

Memory Allocation

- "Local property": A memory location should only be accessed after it is allocated and before it is freed

- \( k \) processes give rise to \( 3^k \) states
  (\( \rightarrow \) "state explosion")

A printer and two processes (cont.)

- A printer and two processes
- Possible actions:
  - \( r_i \): User \( i \) submits print request
  - \( s_i \): Printing of request of \( i \) starts
  - \( t_i \): Print job for user \( i \) terminates

A little bit infinite? Thomas Schwentick
A Toy Example from Verification

A printer and two processes

- A printer and two processes
- Possible actions:
  - \( r_i \): User \( i \) submits print request
  - \( s_i \): Printing of request of \( i \) starts
  - \( t_i \): Print job for user \( i \) terminates

Example properties that could be enforced:
- “Local property”: processes never request a new print job before the last one has terminated, i.e.: for each \( i \) the subrun is of the form \( (r_is_it_i)^* \),
- “Global property”: a print job must be finished before the next one is started, i.e.: between a \( s_i \) and the subsequent \( t_i \) there is no \( s_j \) or \( t_j \), \( j \neq i \)

Memory Allocation
- “Local property”: A memory location should only be accessed after it is allocated and before it is freed
- \( k \) processes give rise to \( 3^k \) states
  \( \rightarrow \) “state explosion”
- What if the number of processes is unknown?
A Toy Example from Verification

A printer and two processes

- A printer and two processes
- Possible actions:
  - $r_i$: User $i$ submits print request
  - $s_i$: Printing of request of $i$ starts
  - $t_i$: Print job for user $i$ terminates

Example properties that could be enforced:

- "Local property": processes never request a new print job before the last one has terminated, i.e.: for each $i$ the subrun is of the form $(r_is_i t_i)^*$,
- "Global property": a print job must be finished before the next one is started, i.e.: between a $s_i$ and the subsequent $t_i$ there is no $s_j$ or $t_j$, $j \neq i$

Memory Allocation

- "Local property": A memory location should only be accessed after it is allocated and before it is freed

- $k$ processes give rise to $3^k$ states
  (→ “state explosion”)

- What if the number of processes is unknown?
- What if the number of processes changes during the computation?
The Automata Approach to Model Checking

- Model checking:
  - System: $M$
  - Property: $\varphi$
  - Does $M \models \varphi$?
The Automata Approach to Model Checking

- **Model checking:**
  - System: $M$
  - Property: $\varphi$
  - Does $M \models \varphi$?

- **The automata approach:**
  - Model a "real life" system as a transition system with finite state space
    - Abstract away data values, process numbers, ...
  - Model executions of the system as infinite strings or trees
  - Specify properties in a logic (e.g., LTL/CTL) that allows translation into automata
  - Use decidability of non-emptiness for automata to obtain decidability of model checking

A little bit infinite? Thomas Schwentick
The Automata Approach to Model Checking

- **Model checking:**
  - System: $M$
  - Property: $\varphi$
  - Does $M \models \varphi$?

- The **automata approach:**
  - Model a "real life" system as a transition system with finite state space
    - Abstract away data values, process numbers,...
  - Model executions of the system as infinite strings or trees
  - Specify properties in a logic (e.g., LTL/CTL) that allows translation into automata
  - Use decidability of non-emptiness for automata to obtain decidability of model checking

- But sometimes the finite state space approach does not really work

A little bit infinite? Thomas Schwentick
The Automata Approach to Model Checking

- **Model checking:**
  - System: $M$
  - Property: $\varphi$
  - Does $M \models \varphi$?

- **The automata approach:**
  - Model a "real life" system as a transition system with finite state space
    - Abstract away data values, process numbers, ...
  - Model executions of the system as infinite strings or trees
  - Specify properties in a logic (e.g., LTL/CTL) that allows translation into automata
  - Use decidability of non-emptiness for automata to obtain decidability of model checking

- **But sometimes the finite state space approach does not really work**

- **Sources of infinity in software systems:**
  - **Data manipulation**: integers, lists, trees, more general pointer structures
  - **Control structures**: procedures, process creation
  - **Asynchronous communication**: unbounded FIFO queues
  - **Parameters**: number of processes, duration of delays
  - **Real-time**: discrete or dense domains

[Esparza]
The Automata Approach to Model Checking

- **Model checking:**
  - System: $M$
  - Property: $\varphi$
  - Does $M \models \varphi$?

- **The automata approach:**
  - Model a "real life" system as a transition system with finite state space
    - Abstract away data values, process numbers, ...
  - Model executions of the system as infinite strings or trees
  - Specify properties in a logic (e.g., LTL/CTL) that allows translation into automata
  - Use decidability of non-emptiness for automata to obtain decidability of model checking

- **But sometimes the finite state space approach does not really work**

- **Sources of infinity in software systems:**
  - **Data manipulation:** integers, lists, trees, more general pointer structures
  - **Control structures:** procedures, process creation
  - **Asynchronous communication:** unbounded FIFO queues
  - **Parameters:** number of processes, duration of delays
  - **Real-time:** discrete or dense domains

- There is a huge need for **Model Checking of infinite-state systems**
Current Approaches to Infinite-State Model Checking

- Infinite-State Model Checking has been an active and successful research area for many years

- **Typical approach (in a nutshell):**
  - Describe system states by some finite objects (strings, tuples of parameters)
  - Describe possible transitions from state to state
  - Device algorithms for checking reachability and/or repeated reachability

- **Examples:**
  - Timed automata [Alur, Dill 90]
  - Mutual exclusion protocols [Abdulla et al. 07]
  - Regular model checking [Bouajjani et al. 00]
Infinite-State Model Checking has been an active and successful research area for many years.

**Typical approach (in a nutshell):**
- Describe system states by some finite objects (strings, tuples of parameters)
- Describe possible transitions from state to state
- Device algorithms for checking reachability and/or repeated reachability

**Examples:**
- Timed automata [Alur, Dill 90]
- Mutual exclusion protocols [Abdulla et al. 07]
- Regular model checking [Bouajjani et al. 00]

**Achievements:**
- Model checking of linear time properties is in many cases possible.
Infinite-State Model Checking has been an active and successful research area for many years.

**Typical approach (in a nutshell):**
- Describe system states by some finite objects (strings, tuples of parameters)
- Describe possible transitions from state to state
- Device algorithms for checking reachability and/or repeated reachability

**Examples:**
- Timed automata [Alur, Dill 90]
- Mutual exclusion protocols [Abdulla et al. 07]
- Regular model checking [Bouajjani et al. 00]

**Achievements:**
- Model checking of linear time properties is in many cases possible

**Still missing:**
- Inter-state reasoning about data from infinite domains (e.g., for each $i$, each $r_i$ is followed by some $s_i$, for an unlimited number of processes)
- A generic framework for branching-time properties
A unifying approach

There are obvious similarities between the XML and the infinite-state model checking scenario:

- Traditional modeling uses finitely labeled structures:
  - strings, trees, Kripke structures
- There is a need to add data from infinite domains to the positions/nodes of such structures
- It should be possible to reason about inter-node relationships between data items
A unifying approach

- There are obvious similarities between the XML and the infinite-state model checking scenario:
  - Traditional modeling uses finitely labeled structures: strings, trees, Kripke structures
  - There is a need to add data from infinite domains to the positions/nodes of such structures
  - It should be possible to reason about inter-node relationships between data items

- A possible unifying approach:
  - Enhance finitely labeled structures by data
    - Various possibilities:
      - One (or more) relations per node
      - A vector of data values per node
      - One data item per node
      - ...and many more
A unifying approach

- There are obvious similarities between the XML and the infinite-state model checking scenario:
  - Traditional modeling uses finitely labeled structures:
    - strings, trees, Kripke structures
  - There is a need to add data from infinite domains to the positions/nodes of such structures
  - It should be possible to reason about inter-node relationships between data items

- A possible unifying approach:
  - Enhance finitely labeled structures by data
  - Various possibilities:
    - One (or more) relations per node
    - A vector of data values per node
    - One data item per node
    - ...and many more

- Parameters to choose:
  1. Underlying finitely labeled structures
  2. Amount and structure of data per node
  3. Operations and predicates on data
  4. Expressiveness of specification language
A unifying approach

- There are obvious similarities between the XML and the infinite-state model checking scenario:
  - Traditional modeling uses finitely labeled structures: strings, trees, Kripke structures
  - There is a need to add data from infinite domains to the positions/nodes of such structures
  - It should be possible to reason about inter-node relationships between data items

- A possible unifying approach:
  - Enhance finitely labeled structures by data
  - Various possibilities:
    - One (or more) relations per node
    - A vector of data values per node
    - One data item per node
    - ...and many more

- Parameters to choose:
  1. Underlying finitely labeled structures
  2. Amount and structure of data per node
  3. Operations and predicates on data
  4. Expressiveness of specification language

- Limitations:
  - To avoid undecidability of reasoning, parameters (1) - (4) have to be chosen very carefully
A unifying approach

- **There are obvious similarities between the XML and the infinite-state model checking scenario:**
  - Traditional modeling uses finitely labeled structures: strings, trees, Kripke structures
  - There is a need to add data from infinite domains to the positions/nodes of such structures
  - It should be possible to reason about inter-node relationships between data items

- **A possible unifying approach:**
  - **Enhance finitely labeled structures by data**
    - Various possibilities:
      - One (or more) relations per node
      - A vector of data values per node
      - One data item per node
      - ...and many more

- **Parameters to choose:**
  1. Underlying finitely labeled structures
  2. Amount and structure of data per node
  3. Operations and predicates on data
  4. Expressiveness of specification language

- **Limitations:**
  - To avoid undecidability of reasoning, parameters (1) - (4) have to be chosen very carefully

- **Related work:**
  - [Autebert et al. 80]
  - [Otto 85]: Regular and context-free languages over infinite alphabets (Symbols have structure)
  - [Henzinger 90]: Kripke structures with one data value per word
  - [Kaminski, Francez 90]: Strings over an infinite alphabet
  - More related work will be mentioned later
In this talk:

- We fix the structure and data parameters:
  1. Finite or infinite strings or trees as underlying finitely labeled structure
  2. One data item per node/position
  3. Only equality tests between data items
Data Strings and Data Trees

- In this talk:
  - We fix the structure and data parameters:
    1. Finite or infinite strings or trees as underlying finitely labeled structure
    2. One data item per node/position
    3. Only equality tests between data items
  - We try to find (4) expressive and decidable reasoning/specification mechanisms
## Data Strings and Data Trees

- **In this talk:**
  - **We fix the structure and data parameters:**
    1. Finite or infinite strings or trees as underlying finitely labeled structure
    2. One data item per node/position
    3. Only equality tests between data items
  - **We try to find** (4) expressive and decidable reasoning/specification mechanisms

### Example: data string

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Definition [Bouyer et al. 03]**

- **Data string**: Finite sequence over $\sum \times D$, where
  - $\sum$ finite (here: \{r, s, t\})
  - $D$ infinite (here: $\mathbb{N}$)
Regular String Languages

- Data strings extend strings
- **Regular string languages** are a very powerful concept:
  1. **Expressiveness**: They capture the desired languages for many kinds of applications
  2. **Decidability**: Automated semantic analysis possible through automata
  3. **Efficiency**: Model checking in linear time.
  4. **Closure properties**: It is hard to find a natural operation under which they are not (effectively) closed
  5. **Robustness**: Tons of characterizations
Regular String Languages

- Data strings extend strings
- **Regular string languages** are a very powerful concept:
  1. **Expressiveness**: They capture the desired languages for many kinds of applications
  2. **Decidability**: Automated semantic analysis possible through automata
  3. **Efficiency**: Model checking in linear time.
  4. **Closure properties**: It is hard to find a natural operation under which they are not (effectively) closed
  5. **Robustness**: Tons of characterizations

→ Regular string languages offer an ideal framework to deal with string languages:
  - Declarative specifications...
  - ..can be translated into automata...
  - ...which can be efficiently
    - evaluated,
    - manipulated and
    - analyzed semantically

A little bit infinite? Thomas Schwentick
Regular String Languages

- Data strings extend strings
- **Regular string languages** are a very powerful concept:
  
  1. **Expressiveness**: They capture the desired languages for many kinds of applications
  2. **Decidability**: Automated semantic analysis possible through automata
  3. **Efficiency**: Model checking in linear time.
  4. **Closure properties**: It is hard to find a natural operation under which they are not (effectively) closed
  5. **Robustness**: Tons of characterizations

→ Regular string languages offer an ideal framework to deal with string languages:
  
  - Declarative specifications...
  - ..can be translated into automata...
  - ...which can be efficiently evaluated, manipulated and analyzed semantically

- Furthermore: There exist canonical generalizations of regular languages for a variety of data types:
  
  - Infinite strings, (infinite) trees, pictures,...
Regular String Languages

- Data strings extend strings
- **Regular string languages** are a very powerful concept:
  
  1. **Expressiveness**: They capture the desired languages for many kinds of applications
  2. **Decidability**: Automated semantic analysis possible through automata
  3. **Efficiency**: Model checking in linear time.
  4. **Closure properties**: It is hard to find a natural operation under which they are not (effectively) closed
  5. **Robustness**: Tons of characterizations

→ Regular string languages offer an ideal framework to deal with string languages:
  
  - Declarative specifications...
  - ..can be translated into automata...
  - ...which can be efficiently
    - evaluated,
    - manipulated and
    - analyzed semantically

- **Furthermore**: There exist canonical generalizations of regular languages for a variety of data types:
  
  - Infinite strings, (infinite) trees, pictures,...

→ **Obvious question:**

  - Is there a corresponding canonical concept of “regular data languages”?
**Bad news:** There does not seem to be a canonical notion of regular data languages.
Regular Data Languages?

- **Bad news:** There does **not** seem to be a canonical notion of regular data languages

- **Good news:** We can mimic the regular languages framework:
  - Declarative specifications...
  - ..can be translated into automata...
  - ...which can be **effectively**
    - evaluated,
    - manipulated
    - analyzed semantically
• **Bad news:** There does not seem to be a canonical notion of regular data languages

• **Good news:** We can mimic the regular languages framework:
  ▶ Declarative specifications...
  ▶ ..can be translated into automata...
  ▶ ...which can be **effectively**
    ■ evaluated,
    ■ manipulated
    ■ analyzed semantically

• **This talk is about the search for a good framework to deal with (string or tree) data languages:**
Regular Data Languages?

- **Bad news:** There does not seem to be a canonical notion of regular data languages

- **Good news:** We can mimic the regular languages framework:
  ▶ Declarative specifications...
  ▶ ...can be translated into automata...
  ▶ ...which can be **effectively**
    ■ evaluated,
    ■ manipulated
    ■ analyzed semantically

- **This talk is about the search for a good framework to deal with (string or tree) data languages:**
  ▶ Automata for data languages
  ▶ Logic-based specification languages
  ▶ Their (potential) use for XML and Model Checking
  ▶ Other approaches
Example properties of data strings

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>r</strong></td>
<td><strong>r</strong></td>
<td><strong>s</strong></td>
<td><strong>r</strong></td>
<td><strong>r</strong></td>
<td><strong>t</strong></td>
<td><strong>r</strong></td>
<td><strong>s</strong></td>
<td><strong>t</strong></td>
<td><strong>s</strong></td>
<td><strong>t</strong></td>
<td><strong>s</strong></td>
<td><strong>t</strong></td>
<td><strong>t</strong></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

A little bit infinite? Thomas Schwentick
Example properties of data strings

Example

\[
\begin{array}{ccccccccccc}
  & r & r & s & r & r & t & t & s & r & t & s & t & s & t & s & t \\
2 & 5 & 5 & 3 & 8 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

A class with class string \(rstrst\)
Example properties of data strings

A class with class string $rstrst$

Examples

(L1) No two $a$-positions do have the same data value (unary key constraint)

(L2) There are two $a$-positions with the same data value

(L3) For each $a$-position there is a $b$-position with the same data value (unary inclusion constraint)

(L4) A print job of a user has to be printed before the next one can be requested (“local safety”)

(L5) Each print request of a user is eventually followed by a print (“local liveness”)

→ (L1) - (L5) are “local properties” of the class strings

(L6) Between two successive print jobs of the same user some other user’s job has to be printed (“global safety”)

(L7) After each printed job a job of some other user is eventually printed (“global liveness”)
Contents

Introduction
Data Model
Automata
  ➢ Register Automata
    Pebble Automata
    Class Memory Automata
Logic
Other Models
Conclusion
- **A natural idea:**
  Equip finite automata with registers that can store data values.
A natural idea:

Equip finite automata with registers that can store data values

→ Register Automata
A natural idea:

Equip finite automata with registers that can store data values

→ Register Automata

(“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)
A natural idea:

Equip finite automata with registers that can store data values

Register Automata

("Finite Memory Automata" in [Kaminski, Francez 90], but w/o labels)

Example

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

Stated differently:

**No two successive s-positions carry the same data value**

Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

\[
R_1 \quad R_2
\]
Register Automata (1/3)

- **A natural idea:** Equip finite automata with registers that can store data values

  → **Register Automata**

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

---

**Example**

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Stated differently:

  **No two successive s-positions carry the same data value**

- Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

\[
\begin{array}{cccccccccccccccc}
  r & r & s & r & r & t & r & s & t & s & r & t & s & t & s & t \\
  2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

| \( R_1 \) | \( \perp \) |
| \( R_2 \) | 2 |
• **A natural idea:**
  Equip finite automata with registers that can store data values

→ **Register Automata**

• (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

---

### Example

• Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

• Stated differently: **No two successive \( s \)-positions carry the same data value**

• Solution: store the data value of the previous \( s \)-position in register 1 and check that it does not occur at the next \( s \)-position

```
\[ R_1 \downarrow \\
R_2 5 \]
```

```
r r s r r t r s t s r t s t s t
2 5 5 3 8 5 5 2 2 8 4 8 3 3 4 4 5 5
```
Register Automata (1/3)

- A natural idea:
  Equip finite automata with registers that can store data values
  ➞ Register Automata
- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

Example

- Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed
- Stated differently:
  No two successive \(s\)-positions carry the same data value
- Solution: store the data value of the previous \(s\)-position in register 1 and check that it does not occur at the next \(s\)-position

\[
\begin{array}{cccccccccccc}
  r & r & s & r & r & t & r & s & t & s & t & s & t & s & t \\
  2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 3 & 4 & 4 & 5 & 5
\end{array}
\]

\[
\begin{array}{c}
R_1 \\
5
\end{array}
\]

\[
\begin{array}{c}
R_2 \\
\bot
\end{array}
\]

A little bit infinite? Thomas Schwentick
A natural idea:

Equip finite automata with registers that can store data values

Register Automata

("Finite Memory Automata" in [Kaminski, Francez 90], but w/o labels)

Example

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

Stated differently:

**No two successive s-positions carry the same data value**

Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

\[
\begin{array}{ccccccccccccccc}
\text{r} & \text{r} & \text{s} & \text{r} & \text{r} & \text{t} & \text{r} & \text{s} & \text{t} & \text{s} & \text{r} & \text{t} & \text{s} & \text{t} & \text{s} & \text{t} \\
2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

\[
\begin{array}{cccc}
R_1 & 5 \\
R_2 & 3 \\
\end{array}
\]
Register Automata (1/3)

- A natural idea:
  Equip finite automata with registers that can store data values

→ Register Automata

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

Example

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Stated differently: **No two successive s-positions carry the same data value**

- Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
R_1 & 5 \\
R_2 & 8 \\
\end{array}
\]
Register Automata (1/3)

- **A natural idea:** Equip finite automata with registers that can store data values

  ➞ **Register Automata**

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

---

### Example

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Stated differently: **No two successive s-positions carry the same data value**

- Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

  \[
  r \ r \ s \ r \ r \ t \ r \ s \ t \ s \ r \ t \ s \ t \ s \ t \ s \ t \\
  2 \ 5 \ 5 \ 3 \ 8 \ 5 \ 5 \ 2 \ 2 \ 8 \ 4 \ 8 \ 3 \ 3 \ 4 \ 4 \ 5 \ 5 \\
  \]

  \[
  \begin{array}{c}
  R_1 \\
  5 \\
  \hline
  R_2 \\
  8 \\
  \end{array} \\
  \]

A little bit infinite? Thomas Schwentick
Register Automata (1/3)

- **A natural idea:** Equip finite automata with registers that can store data values

  ➞ Register Automata

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

**Example**

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Stated differently: **No two successive s-positions carry the same data value**

- Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

```
r r s r r t r s t s r t s t s t  
2 5 5 3 8 5 2 2 8 4 8 3 3 4 4 5 5
```

<table>
<thead>
<tr>
<th></th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

A little bit infinite? Thomas Schwentick
Register Automata (1/3)

- **A natural idea:**
  Equip finite automata with registers that can store data values

  ➞ Register Automata

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

### Example

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Stated differently:
  - **No two successive** $s$-**positions carry the same data value**
- Solution: store the data value of the previous $s$-position in register 1 and check that it does not occur at the next $s$-position

```
  r  r  s  r  r  t  r  s  t  s  r  t  s  t  s  t  s  t  s  t
  2  5  5  3  8  5  5  2  2  8  4  8  3  3  4  4  5  5
  R_1 2
  R_2 8
```

A little bit infinite? Thomas Schwentick
A natural idea:
Equip finite automata with registers that can store data values
→ Register Automata

("Finite Memory Automata" in [Kaminski, Francez 90], but w/o labels)

Example

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

Stated differently:
**No two successive s-positions carry the same data value**

Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

```
  r r s r r t r s t s t s t
  2 5 5 3 8 5 5 2 2 8 4 8 3 3 4 4 5 5

R_1 2
R_2 8
```
A natural idea:
Equip finite automata with registers that can store data values

Register Automata

("Finite Memory Automata" in [Kaminski, Francez 90], but w/o labels)

Example

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

Stated differently:
**No two successive s-positions carry the same data value**

Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position
Register Automata (1/3)

- **A natural idea:** Equip finite automata with registers that can store data values
  
  → Register Automata

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

### Example

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Stated differently: **No two successive s-positions carry the same data value**

- Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

```
r r s r r t r s t s r t s t s t
2 5 5 3 8 5 5 2 2 8 4 8 3 3 4 4 5 5
```

<table>
<thead>
<tr>
<th>R1</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>4</td>
</tr>
</tbody>
</table>
Register Automata (1/3)

- A natural idea:
  Equip finite automata with registers that can store data values
  ➞ Register Automata

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

Example

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Stated differently:
  **No two successive s-positions carry the same data value**

- Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

```
  r  r  s  r  r  t  r  s  t  s  r  t  s  t  s  t  s  t
  2  5  5  3  8  5  5  2  2  8  4  8  3  3  4  4  5  5

  R1 8
  R2 4
```

A little bit infinite? Thomas Schwentick
Register Automata (1/3)

- **A natural idea:** Equip finite automata with registers that can store data values

→ **Register Automata**

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

---

**Example**

- Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed

- Stated differently: **No two successive s-positions carry the same data value**

- Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

```
   r r s r r t r s t s r t s t s t
2 5 5 3 8 5 5 2 2 8 4 8 3 3 4 4 5 5
```

```
R_1 3
R_2 4
```

A little bit infinite? Thomas Schwentick
Register Automata (1/3)

- **A natural idea:**
  Equip finite automata with registers that can store data values

  ➞ Register Automata

- (“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

**Example**

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Stated differently: **No two successive s-positions carry the same data value**
- Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

\[
\begin{align*}
R_1 & = 3 \\
R_2 & = 4
\end{align*}
\]

A little bit infinite? Thomas Schwentick
Register Automata (1/3)

- **A natural idea:** Equip finite automata with registers that can store data values
  
  \[\text{Register Automata}\]

- ("Finite Memory Automata" in [Kaminski, Francez 90], but w/o labels)

---

**Example**

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Stated differently: **No two successive \(s\)-positions carry the same data value**

- Solution: store the data value of the previous \(s\)-position in register 1 and check that it does not occur at the next \(s\)-position

\[
\begin{array}{cccccccccccccc}
\text{r} & \text{r} & \text{s} & \text{r} & \text{r} & \text{t} & \text{r} & \text{s} & \text{t} & \text{s} & \text{t} & \text{s} & \text{t} & \text{s} & \text{t} \\
2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

\[
\begin{array}{cc}
R_1 & 4 \\
R_2 & \perp \\
\end{array}
\]
A natural idea: Equip finite automata with registers that can store data values

Example

Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed

Stated differently: No two successive s-positions carry the same data value

Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position
A natural idea: Equip finite automata with registers that can store data values

→ Register Automata

("Finite Memory Automata" in [Kaminski, Francez 90], but w/o labels)

Example

Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed

Stated differently: No two successive s-positions carry the same data value

Solution: store the data value of the previous s-position in register 1 and check that it does not occur at the next s-position

\[r \quad r \quad s \quad r \quad r \quad t \quad r \quad s \quad t \quad s \quad r \quad t \quad s \quad t \quad s \quad t \quad s \quad t\]

\[2 \quad 5 \quad 5 \quad 3 \quad 8 \quad 5 \quad 5 \quad 2 \quad 2 \quad 8 \quad 4 \quad 8 \quad 3 \quad 3 \quad 4 \quad 4 \quad 5 \quad 5\]

\[\begin{array}{c}
R_1 \quad 5 \\
R_2 \quad \bot
\end{array}\]
A natural idea:
Equip finite automata with registers that can store data values

→ Register Automata

(“Finite Memory Automata” in [Kaminski, Francez 90], but w/o labels)

Example

Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed

Stated differently:
No two successive \( s \)-positions carry the same data value

Solution: store the data value of the previous \( s \)-position in register 1 and check that it does not occur at the next \( s \)-position

\[
\begin{array}{cccccccccccc}
  r & r & s & r & r & t & r & s & t & s & r & t & s & t & s & t & s & t & s & t \\
  2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

\[
\begin{array}{c|c}
  R_1 & 5 \\
  R_2 & \bot \\
\end{array}
\]
Theorem 1 [Kaminski, Francez 90]

(a) Non-emptiness for register automata is decidable

(b) Testing $L(A_1) \subseteq L(A_2)$ is decidable as long as $A_2$ has $\leq 2$ registers
### Theorem 1 [Kaminski, Francez 90]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>Non-emptiness for register automata is decidable</td>
</tr>
<tr>
<td>(b)</td>
<td>Testing $L(A_1) \subseteq L(A_2)$ is decidable as long as $A_2$ has $\leq 2$ registers</td>
</tr>
</tbody>
</table>

### Proof idea

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>Crux: if there is a string in $L(A)$, then there is one with $\leq</td>
</tr>
</tbody>
</table>
**Theorem 1 [Kaminski, Francez 90]**

(a) Non-emptiness for register automata is decidable

(b) Testing $L(A_1) \subseteq L(A_2)$ is decidable as long as $A_2$ has $\leq 2$ registers

**Proof idea**

(a) Crux: if there is a string in $L(A)$, then there is one with $\leq |Q| + 1$ different data values

- Register automata can test global regular properties
  - That’s simple: just ignore the data values
**Theorem 1 [Kaminski, Francez 90]**

<table>
<thead>
<tr>
<th>(a) Non-emptiness for register automata is decidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Testing $L(A_1) \subseteq L(A_2)$ is decidable as long as $A_2$ has $\leq 2$ registers</td>
</tr>
</tbody>
</table>

**Proof idea**

| (a) Crux: if there is a string in $L(A)$, then there is one with $\leq |Q| + 1$ different data values |

- Register automata can test global regular properties
  - That’s simple: just ignore the data values

**Theorem 2 [Neven et al. 01]**

- No register automaton can test (L4): “A print job of a user has to be printed before the next one can be requested”
### Theorem 1 [Kaminski, Francez 90]

(a) Non-emptiness for register automata is decidable

(b) Testing $L(A_1) \subseteq L(A_2)$ is decidable as long as $A_2$ has $\leq 2$ registers

#### Proof idea

(a) Crux: if there is a string in $L(A)$, then there is one with $\leq |Q| + 1$ different data values

- Register automata can test global regular properties
  - That’s simple: just ignore the data values

### Theorem 2 [Neven et al. 01]

- No register automaton can test (L4): “A print job of a user has to be printed before the next one can be requested”

#### Proof idea

- Assume some 3-register automaton $A$ tests (L4)
### Theorem 1 [Kaminski, Francez 90]

<table>
<thead>
<tr>
<th>(a)</th>
<th>Non-emptiness for register automata is decidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b)</td>
<td>Testing $L(A_1) \subseteq L(A_2)$ is decidable as long as $A_2$ has $\leq 2$ registers</td>
</tr>
</tbody>
</table>

#### Proof idea

- **(a)** Crux: if there is a string in $L(A)$, then there is one with $\leq |Q| + 1$ different data values
- **(b)** Register automata can test global regular properties
  - That’s simple: just ignore the data values

### Theorem 2 [Neven et al. 01]

- No register automaton can test (L4): “A print job of a user has to be printed before the next one can be requested”

#### Proof idea

- Assume some 3-register automaton $A$ tests (L4)
- Consider the following input:
  
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>$r$</td>
<td>$r$</td>
<td>$r$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

A little bit infinite? Thomas Schwentick
### Theorem 1 [Kaminski, Francez 90]

| (a) Non-emptiness for register automata is decidable |
| (b) Testing $L(A_1) \subseteq L(A_2)$ is decidable as long as $A_2$ has $\leq 2$ registers |

### Proof idea

| (a) Crux: if there is a string in $L(A)$, then there is one with $\leq |Q| + 1$ different data values |
| Register automata can test global regular properties |
| That's simple: just ignore the data values |

### Theorem 2 [Neven et al. 01]

- No register automaton can test (L4): “A print job of a user has to be printed before the next one can be requested”

### Proof idea

- Assume some 3-register automaton $A$ tests (L4)
- Consider the following input:

  $\begin{array}{cccccc}
  r & r & r & r & r \\
  1 & 2 & 3 & 4 & 1 \\
  \end{array}$

  $\begin{array}{cc}
  R_1 & 4 \\
  R_2 & 2 \\
  R_3 & 3 \\
  \end{array}$

A little bit infinite? Thomas Schwentick
Register Automata (2/3)

Theorem 1 [Kaminski, Francez 90]
(a) Non-emptiness for register automata is decidable
(b) Testing \( L(A_1) \subseteq L(A_2) \) is decidable as long as \( A_2 \) has \( \leq 2 \) registers

Proof idea
(a) Crux: if there is a string in \( L(A) \), then there is one with \( \leq |Q| + 1 \) different data values

- Register automata can test global regular properties
  - That’s simple: just ignore the data values

Theorem 2 [Neven et al. 01]
- No register automaton can test (L4): “A print job of a user has to be printed before the next one can be requested”

Proof idea
- Assume some 3-register automaton \( A \) tests (L4)
- Consider the following input:

  \[
  \begin{array}{cccccc}
  r & r & r & r & r \\
  1 & 2 & 3 & 4 & 1 \\
  \end{array}
  \]

- \( A \) cannot detect that process 1 has a pending print job
### Theorem 1 [Kaminski, Francez 90]

(a) Non-emptiness for register automata is decidable

(b) Testing $L(A_1) \subseteq L(A_2)$ is decidable as long as $A_2$ has $\leq 2$ registers

#### Proof idea

(a) Crux: if there is a string in $L(A)$, then there is one with $\leq |Q| + 1$ different data values

- Register automata can test global regular properties
  - That’s simple: just ignore the data values

### Theorem 2 [Neven et al. 01]

- No register automaton can test (L4): “A print job of a user has to be printed before the next one can be requested”

#### Proof idea

- Assume some 3-register automaton $A$ tests (L4)
- Consider the following input:

  $r \ r \ r \ r \ r
  1 \ 2 \ 3 \ 4 \ 1$

- $A$ cannot detect that process 1 has a pending print job

  $A$ cannot detect that process 1 has a pending print job

  $R_1 \ 4$
  $R_2 \ 2$
  $R_3 \ 3$

  $A$ cannot detect that process 1 has a pending print job

  $R_1 \ 4$
  $R_2 \ 2$
  $R_3 \ 3$

- Easy to generalize for arbitrary number of registers
Theorem 3 [Kaminski, Francez 90]

- Testing whether a register automaton accepts every data string is undecidable
Register Automata (3/3)

Theorem 3 [Kaminski, Francez 90]

- Testing whether a register automaton accepts every data string is undecidable

Summary of properties of register automata:

<table>
<thead>
<tr>
<th>Property</th>
<th>RegisterA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressiveness</strong></td>
<td>$(L2),(L6),(L7)$</td>
</tr>
<tr>
<td><strong>Decidability</strong></td>
<td>✓</td>
</tr>
<tr>
<td>Non-emptiness</td>
<td>✓</td>
</tr>
<tr>
<td>Containment</td>
<td>–</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>✓</td>
</tr>
<tr>
<td>Data complexity word problem</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Closure properties</strong></td>
<td>✓</td>
</tr>
<tr>
<td>Union</td>
<td>✓</td>
</tr>
<tr>
<td>Intersection</td>
<td>✓</td>
</tr>
<tr>
<td>Complement</td>
<td>–</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td>–</td>
</tr>
</tbody>
</table>

A little bit infinite? Thomas Schwentick
Register Automata (3/3)

Theorem 3 [Kaminski, Francez 90]

- Testing whether a register automaton accepts every data string is undecidable

Summary of properties of register automata:

<table>
<thead>
<tr>
<th>Property</th>
<th>RegisterA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressiveness</strong></td>
<td>(L2),(L6),(L7)</td>
</tr>
<tr>
<td><strong>Decidability</strong></td>
<td>✓</td>
</tr>
<tr>
<td>Non-emptiness</td>
<td>✓</td>
</tr>
<tr>
<td>Containment</td>
<td>–</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>✓</td>
</tr>
<tr>
<td>Data complexity word problem</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Closure properties</strong></td>
<td>✓</td>
</tr>
<tr>
<td>Union</td>
<td>✓</td>
</tr>
<tr>
<td>Intersection</td>
<td>✓</td>
</tr>
<tr>
<td>Complement</td>
<td>–</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td>–</td>
</tr>
</tbody>
</table>

Variants of the basic RA model:

- 1-way and 2-way
- Deterministic and non-deterministic
- Alternating [Neven et al. 01, Demri Lazic 06]
- Look-ahead automata [Zeitlin 06]
- “Unification based” [Tal 99]

A little bit infinite?

Thomas Schwentick
Contents

Introduction
Data Model
Automata
  Register Automata
  ▶ Pebble Automata
    Class Memory Automata
Logic
Other Models
Conclusion

A little bit infinite?  Thomas Schwentick  tu.
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata
A different approach: instead of registers use pebbles (pointers/heads)

Restrict movement and placement of pebbles:
- Pebbles are numbered $1, 2, \ldots, k$
- Only pebble with highest number $i$ can be moved or lifted
- Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered \(1, 2, \ldots, k\)
  - Only pebble with highest number \(i\) can be moved or lifted
  - Only pebble with number \(i + 1\) can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive \(s\)-positions carry the same data value
Solution: for each \(s\)-position check that the previous \(s\)-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user's job has to be printed
Again stated differently: no two successive $s$-positions carry the same data value
Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number \(i\) can be moved or lifted
  - Only pebble with number \(i + 1\) can be placed

\[\text{Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed}\]

- Again stated differently: **no two successive** \(s\)-**positions carry the same data value**
- **Solution:** for each \(s\)-position check that the previous \(s\)-position has a different data value
Pebble automata (1/3)

• A different approach: instead of registers use pebbles (pointers/heads)
• Restrict movement and placement of pebbles:
  ▶ Pebbles are numbered 1, 2, ..., k
  ▶ Only pebble with highest number i can be moved or lifted
  ▶ Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive s-positions carry the same data value

Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive $s$-positions carry the same data value
Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive s-positions carry the same data value
Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for $(L6)$:
Between two successive print jobs of the same user some other user’s job has to be printed

Example: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
- Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
- Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
- Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive $s$-positions carry the same data value
Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

---

**Example**

```
  r  r  s  r  r  t  r  s  t  s  t  s  t  s  t
  2  5  5  3  8  5  5  2  2  8  4  8  3  3  4  4  5  5
```

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive s-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive s-positions carry the same data value

Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  ▶ Pebbles are numbered $1, 2, \ldots, k$
  ▶ Only pebble with highest number $i$ can be moved or lifted
  ▶ Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
A different approach: instead of registers use pebbles (pointers/heads)

Restrict movement and placement of pebbles:
- Pebbles are numbered $1, 2, \ldots, k$
- Only pebble with highest number $i$ can be moved or lifted
- Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive s-positions carry the same data value
- Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive $s$-positions carry the same data value
Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

---

**Example**

```
  r r s r r t r s t s t s t
  2 5 5 3 8 5 5 2
```

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6): *Between two successive print jobs of the same user some other user’s job has to be printed*

Again stated differently: *no two successive $s$-positions carry the same data value*

**Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered \(1, 2, \ldots, k\)
  - Only pebble with highest number \(i\) can be moved or lifted
  - Only pebble with number \(i + 1\) can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive \(s\)-positions carry the same data value
- Solution: for each \(s\)-position check that the previous \(s\)-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

### Example

|   |   | r | r | s | r | r | t | r | s | t | s | t | s | t | s | t | t |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   |   | 2 | 5 | 5 | 3 | 8 | 5 | 5 | 2 | 2 | 8 | 4 | 8 | 3 | 3 | 4 | 4 | 5 | 5 |

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive s-positions carry the same data value**
- **Solution:** for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed

- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
A different approach: instead of registers use pebbles (pointers/heads)

Restrict movement and placement of pebbles:
- Pebbles are numbered $1, 2, \ldots, k$
- Only pebble with highest number $i$ can be moved or lifted
- Only pebble with number $i + 1$ can be placed

Example

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

Again stated differently: **no two successive $s$-positions carry the same data value**

**Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed.

Again stated differently: no two successive $s$-positions carry the same data value.

Solution: for each $s$-position check that the previous $s$-position has a different data value.
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

**Example automaton for (L6):**

Between two successive print jobs of the same user some other user’s job has to be printed

- Again stated differently: no two successive $s$-positions carry the same data value
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata

Example

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

→ Pebble automata

Example

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive \( s \)-positions carry the same data value
- **Solution:** for each \( s \)-position check that the previous \( s \)-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive $s$-positions carry the same data value
Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive s-positions carry the same data value

Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata

Example

```
  r  r  s  r  r  t  r  s  t  s  r  t  s  t  s  t
  2  5  5  3  8  5  5  2  2  8  4  8  3  3  4  4  5  5
```

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive s-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)

- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata

Example

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>r</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Again stated differently: **No two successive** $s$-**positions carry the same data value**

- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value

Example

<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>r</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata

Example

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>r</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata

### Example

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>$r$</td>
<td>$s$</td>
<td>$r$</td>
<td>$r$</td>
<td>$t$</td>
<td>$r$</td>
<td>$s$</td>
<td>$t$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Example automaton for (L6): Between two successive print jobs of the same user some other user's job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
A different approach: instead of registers use pebbles (pointers/heads)

Restrict movement and placement of pebbles:
- Pebbles are numbered $1, 2, \ldots, k$
- Only pebble with highest number $i$ can be moved or lifted
- Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive s-positions carry the same data value
Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed

- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
A different approach: instead of registers use pebbles (pointers/heads)

Restrict movement and placement of pebbles:
- Pebbles are numbered $1, 2, \ldots, k$
- Only pebble with highest number $i$ can be moved or lifted
- Only pebble with number $i + 1$ can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive s-positions carry the same data value

Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered \(1, 2, \ldots, k\)
  - Only pebble with highest number \(i\) can be moved or lifted
  - Only pebble with number \(i + 1\) can be placed

Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed

Example: for each \(s\)-position check that the previous \(s\)-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)

- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive s positions carry the same data value
Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed

- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive $s$-positions carry the same data value
Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered \(1, 2, \ldots, k\)
  - Only pebble with highest number \(i\) can be moved or lifted
  - Only pebble with number \(i + 1\) can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive \(s\)-positions carry the same data value

Solution: for each \(s\)-position check that the previous \(s\)-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata

### Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>r</th>
<th>r</th>
<th>s</th>
<th></th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive s-positions carry the same data value**
- **Solution:** for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

**Example**

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Pebble automata

Example

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>$r$</td>
<td>$s$</td>
<td>$r$</td>
<td>$r$</td>
<td>$t$</td>
<td>$r$</td>
<td>$s$</td>
<td>$t$</td>
<td>$s$</td>
<td>$t$</td>
<td>$s$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution**: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

```
  r  r  s  r  r  t  r  s  t  8  r  t  s  t  s  t  s  t
  2  5  5  3  8  5  5  2  2  8  4  8  3  3  4  4  5  5
```

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive s-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

---

Example

```
 r  r  s  r  r  t  r  s  t  s  r  t  s  t  s  t  r  t  s  t  s  t  s  t
2 5 5 3 8 5 5 2 2 8 4 8 3 3 4 4 5 5
```

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive s-positions carry the same data value**
- **Solution:** for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

=> Pebble automata

Example

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

-- Pebble automata

Example

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>r</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered \(1, 2, \ldots, k\)
  - Only pebble with highest number \(i\) can be moved or lifted
  - Only pebble with number \(i + 1\) can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive \(s\)-positions carry the same data value
Solution: for each \(s\)-position check that the previous \(s\)-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)

- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive s-positions carry the same data value

Solution: for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

<table>
<thead>
<tr>
<th>$r$</th>
<th>$r$</th>
<th>$s$</th>
<th>$r$</th>
<th>$r$</th>
<th>$t$</th>
<th>$r$</th>
<th>$s$</th>
<th>$t$</th>
<th>$s$</th>
<th>$t$</th>
<th>$s$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata

Example

<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

<table>
<thead>
<tr>
<th>$r$</th>
<th>$r$</th>
<th>$s$</th>
<th>$r$</th>
<th>$r$</th>
<th>$t$</th>
<th>$r$</th>
<th>$s$</th>
<th>$t$</th>
<th>$s$</th>
<th>$t$</th>
<th>$s$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
- Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
A different approach: instead of registers use pebbles (pointers/heads)

Restrict movement and placement of pebbles:
- Pebbles are numbered $1, 2, \ldots, k$
- Only pebble with highest number $i$ can be moved or lifted
- Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered 1, 2, ..., k
  - Only pebble with highest number i can be moved or lifted
  - Only pebble with number i + 1 can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive s-positions carry the same data value

Solution: for each s-position check that the previous s-position has a different data value
A different approach: instead of registers use pebbles (pointers/heads)

Restrict movement and placement of pebbles:

- Pebbles are numbered $1, 2, \ldots, k$
- Only pebble with highest number $i$ can be moved or lifted
- Only pebble with number $i + 1$ can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

```
  r r s r r t r s t s r t s t s t
2 5 5 3 8 5 5 2 2 8 4 8 3 3 4 4 5 5 1
```

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive s-positions carry the same data value**
- **Solution:** for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

- Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>r</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive s-positions carry the same data value**
- **Solution:** for each s-position check that the previous s-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):

Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered \(1, 2, \ldots, k\)
  - Only pebble with highest number \(i\) can be moved or lifted
  - Only pebble with number \(i + 1\) can be placed

---

**Example**

```
\[ \begin{array}{cccccccccccc}
r & r & s & r & r & t & r & s & t & s & t & s \\
2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 \\
\end{array} \]
```

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive \(s\)-positions carry the same data value**
- **Solution:** for each \(s\)-position check that the previous \(s\)-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

### Example

<table>
<thead>
<tr>
<th>$r$</th>
<th>$r$</th>
<th>$s$</th>
<th>$r$</th>
<th>$r$</th>
<th>$t$</th>
<th>$r$</th>
<th>$s$</th>
<th>$t$</th>
<th>$s$</th>
<th>$t$</th>
<th>$s$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

- Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**
- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed

Again stated differently: no two successive $s$-positions carry the same data value

Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

→ Pebble automata

Example

```
  r   r   s   r   r   t   r   s   t   s   t   s   t
  2   5   5   3   8   5   5   2   2   8   4   8   3   3   4   4   5   5   1
```

- Example automaton for (L6): Between two successive print jobs of the same user some other user’s job has to be printed
- Again stated differently: no two successive $s$-positions carry the same data value
- Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, \ldots, k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example automaton for (L6):
Between two successive print jobs of the same user some other user’s job has to be printed
Again stated differently: no two successive $s$-positions carry the same data value
Solution: for each $s$-position check that the previous $s$-position has a different data value
Pebble automata (1/3)

- A different approach: instead of registers use pebbles (pointers/heads)
- Restrict movement and placement of pebbles:
  - Pebbles are numbered $1, 2, ..., k$
  - Only pebble with highest number $i$ can be moved or lifted
  - Only pebble with number $i + 1$ can be placed

Example

Example automaton for (L6): **Between two successive print jobs of the same user some other user’s job has to be printed**

- Again stated differently: **no two successive $s$-positions carry the same data value**
- **Solution:** for each $s$-position check that the previous $s$-position has a different data value
Pebble automata are a fairly powerful model:

- E.g., they can express all example properties (L1) – (L7)
Pebble automata are a fairly powerful model:

- E.g., they can express all example properties (L1) – (L7)
- They can even express all properties that can be described by first-order logic
Pebble automata are a fairly powerful model:
- E.g., they can express all example properties (L1) – (L7)
- They can even express all properties that can be described by first-order logic
- Unfortunately: first-order logic on data strings is undecidable (see below)
- Even non-emptiness of pebble automata is undecidable
Pebble automata are a fairly powerful model:

- E.g., they can express all example properties (L1) – (L7)
- They can even express all properties that can be described by first-order logic
- Unfortunately: first-order logic on data strings is undecidable (see below)
- Even non-emptiness of pebble automata is undecidable

On the other hand the model is quite robust:

- One-way and two-way, deterministic and non-deterministic pebble automata are equally expressive
Pebble automata are a fairly powerful model:

- E.g., they can express all example properties (L1) – (L7)
- They can even express all properties that can be described by first-order logic
- Unfortunately: first-order logic on data strings is undecidable (see below)
- Even non-emptiness of pebble automata is undecidable

On the other hand the model is quite robust:

- One-way and two-way, deterministic and non-deterministic pebble automata are equally expressive

### Expressiveness

<table>
<thead>
<tr>
<th></th>
<th>RegisterA</th>
<th>PebbleA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L2),(L6),(L7)</td>
<td>(L1)–(L7)</td>
<td></td>
</tr>
</tbody>
</table>

### Decidability

<table>
<thead>
<tr>
<th>Property</th>
<th>RegisterA</th>
<th>PebbleA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-emptiness</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Containment</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

### Efficiency

<table>
<thead>
<tr>
<th>Property</th>
<th>RegisterA</th>
<th>PebbleA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data complexity word pr.</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Closure properties

<table>
<thead>
<tr>
<th>Property</th>
<th>RegisterA</th>
<th>PebbleA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intersection</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complement</td>
<td>–</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Robustness

<table>
<thead>
<tr>
<th></th>
<th>RegisterA</th>
<th>PebbleA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>–</td>
<td>✓</td>
</tr>
</tbody>
</table>
Pebble Automata (3/3)

A little bit infinite? Thomas Schwentick.
Contents

Introduction
Data Model
Automata
  Register Automata
  Pebble Automata
Class Memory Automata
Logic
Other Models
Conclusion
Intermediate state of affairs:

Register Automata:

- Decidable Non-emptiness: 😃
- Not expressive enough: 😞
Intermediate state of affairs:

Register Automata:
- Decidable Non-emptiness: 😊
- Not expressive enough: 😞

Pebble Automata:
- Very expressive: 😊
- Undecidable Non-emptiness: 😞
Class Memory Automata (1/5)

- **Intermediate state of affairs:**
  - **Register Automata:**
    - Decidable Non-emptiness: 😊
    - Not expressive enough: 😞
  - **Pebble Automata:**
    - Very expressive: 😊
    - Undecidable Non-emptiness: 😞

- **New approach:**
  - Combine a global automaton with one automaton per class
Class Memory Automata (1/5)

- **Intermediate state of affairs:**
  - **Register Automata:**
    - Decidable Non-emptiness: 😊
    - Not expressive enough: 😞
  - **Pebble Automata:**
    - Very expressive: 😊
    - Undecidable Non-emptiness: 😞

- **New approach:**
  - Combine a global automaton with one automaton per class
  - More precisely:
    - Transitions depend on
      - the input symbol (from the finite set of labels)
      - the current state
      - the state assumed last time in the class of the input data value
Class Memory Automata (1/5)

- Intermediate state of affairs:
  - Register Automata:
    - Decidable Non-emptiness: 😊
    - Not expressive enough: 😞
  - Pebble Automata:
    - Very expressive: 😊
    - Undecidable Non-emptiness; 😞

- New approach:
  - Combine a global automaton with one automaton per class
  - More precisely:
    - Transitions depend on
      - the input symbol (from the finite set of labels)
      - the current state
      - the state assumed last time in the class of the input data value
    - The automaton accepts if
      - the last state is in an accepting set $F_g$
      - and for each class, the last state is in a set $F_l$
Intermediate state of affairs:

- **Register Automata:**
  - Decidable Non-emptiness: 😊
  - Not expressive enough: 😞

- **Pebble Automata:**
  - Very expressive: 😊
  - Undecidable Non-emptiness: 😞

New approach:

- Combine a global automaton with one automaton per class

- More precisely:
  - Transitions depend on
    - the input symbol (from the finite set of labels)
    - the current state
    - the state assumed last time in the class of the input data value
  - The automaton accepts if
    - the last state is in an accepting set $F_g$
    - and for each class, the last state is in a set $F_l$

→ Class Memory Automata

[Bojańczyk et al. 06, Björklund, S 07]
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*srd^*)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>r</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccc}
 r & r & s & r & r & t & r & s & t & s & r & t & s & t & s & t \\
 2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*s r^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

|   | r |   | s | r | r | t |   | r | s | t |   | s | t |   | s | t |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   | 2 | 5 | 5 | 3 | 8 | 5 | 5 | 2 | 2 | 8 | 4 | 8 | 3 | 4 | 4 | 5 |

\begin{array}{cc}
\bot & t \\
\bot & r \\
\end{array}

- States are of the form \(p \underbrace{\quad q}\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^* sr^* t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

```
2 5 5 3 8 5 5 2 2 8 4 8 3 3 4 4 5 5
```

- States are of the form \(p/q\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

### Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

```
2 5 5 3 8 5 2 2 8 4 8 3 3 4 4 5 5
```

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
### Class Memory Automata (2/5)

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| - Class memory automaton for the set of data strings  
  - with global pattern \((r^*sr^*t)^*\),  
  - with local pattern \((rst)^*\) (for each class),  
  - where at most one (singular) process prints more than once |

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>r</td>
<td>s</td>
<td>r</td>
<td>r</td>
<td>t</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>s</td>
<td>t</td>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>t</td>
<td>t'</td>
<td>s'</td>
<td>s'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- States are of the form \(p/q\), where  
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)  
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

```
2 5 5 3 8 5 5 2 2 8 4 8 3 3 4 4 5 5
```

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccc}
  r & r & s & r & r & t & r & s & t & s & r & t & s & t & s & t \\
  2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

- States are of the form \([p, q] \) where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccccccc}
& r & r & s & r & r & t & r & s & t & s & r & t & s & t & s & t \\
2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccc}
  r & r & s & r & r & t & r & s & t & s & t & s & t & s & t \\
  2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

- States are of the form \(p\_q\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

- States are of the form \(p\ q\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^* s r^* t)^*\),
  - with local pattern \((r s t)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccc}
  & r & r & s & r & r & t & r & s & t & s & r & t & s & t \\
 2 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
**Class Memory Automata (2/5)**

### Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccccc}
& r & r & s & r & r & t & r & s & t & s & t & s & t & s & t \\
2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\downarrow & t & t' & s' & s' & s' & t' \\
\downarrow & r & r' & s & r & s & t' \\
\end{array}
\]

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccc}
  r & r & s & r & r & t & r & s & t & s & t & s & t \\
  2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 3 & 3 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
  \perp & t & t' & s' & s' & t' & t' \\
  \perp & r & \check{r} & \check{s} & r & \check{t} & \check{t} \\
\end{array}
\]

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccccccc}
  r & r & s & r & r & t & r & s & t & s & r & t & s & t & s & t \\
  2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 4 & 4 & 5 \quad 5
\end{array}
\]

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Example

- Class memory automaton for the set of data strings
  - with global pattern $(r^*sr^*t)^*$,
  - with local pattern $(rst)^*$ (for each class),
  - where at most one (singular) process prints more than once

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>r</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \downarrow t \quad t' \quad s' \quad s' \quad t' \quad t' \quad s' \quad t' \]
\[ \downarrow r \quad r' \quad s' \quad r \quad r' \quad s \quad t \]

- States are of the form $[p \quad q]$, where
  - $p$ remembers whether the singular process already has appeared and whether $s$ or $t$ has been seen last: $s, t, s', t'$
  - $q$ is just the last symbol, (dotted if from the singular process)
### Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

```
   r  r  s  r  r  t  r  s  t  s  r  t  s  t  s  t
  2  5  5  3  8  5  5  2  2  8  4  8  3  3  4  4  5  5
```

- States are of the form \(p/q\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

```
  r   r   s   r   r   t   r   s   t   s   r   s   t   s   t
  2   5   5   3   8   5   2   2   8   4   8   3   3   4   4   5   5
```

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^* s r^* t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>r</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
</table>
| 2 | 5 | 5 | 3 | 8 | 5 | 5 | 2 | 2 | 8 | 4 | 8 | 3 | 4 | 4 | 5 | 5

\(\perp\) \(\perp\) \(t\) \(t'\) \(s'\) \(s'\) \(s'\) \(t'\) \(t'\) \(s'\) \(t'\) \(s'\) \(s'\) \(t'\) \(s'\) \(t'\) \(s'\) \(t'\)

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

| 2 | 5 | 5 | 3 | 8 | 5 | 5 | 2 | 2 | 8 | 4 | 8 | 3 | 3 | 4 | 4 | 5 | 5 |

- States are of the form \(\frac{p}{q}\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

- States are of the form \(p/q\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

• Class memory automaton for the set of data strings
  • with global pattern \((r^*sr^*t)^*\),
  • with local pattern \((rst)^*\) (for each class),
  • where at most one (singular) process prints more than once

\[
\begin{array}{cccccccccccccccc}
\text{r} & \text{r} & \text{s} & \text{r} & \text{r} & \text{s} & \text{t} & \text{r} & \text{s} & \text{t} & \text{s} & \text{t} & \text{s} & \text{t} \\
2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 4 & 4 & 5 & 5
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
\text{t} & \text{t'} & \text{s'} & \text{s'} & \text{t'} & \text{t'} & \text{s'} & \text{s'} & \text{s'} & \text{t'} & \text{s'} & \text{t'} & \text{t'} & \text{s'} & \text{s'} & \text{s'} & \text{s'} & \text{s'}
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
\text{r} & \text{r'} & \text{s'} & \text{r} & \text{r'} & \text{r'} & \text{s'} & \text{s'} & \text{s'} & \text{t'} & \text{s'} & \text{t'} & \text{t'} & \text{s'} & \text{s'} & \text{s'} & \text{s'} & \text{s'}
\end{array}
\]

• States are of the form \([p,q]\), where
  • \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  • \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
### Class Memory Automata (2/5)

#### Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

<table>
<thead>
<tr>
<th>2</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>8</th>
<th>5</th>
<th>2</th>
<th>2</th>
<th>8</th>
<th>4</th>
<th>8</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>t</td>
<td>t'</td>
<td>s'</td>
<td>s'</td>
<td>t'</td>
<td>s'</td>
<td>t'</td>
<td>s'</td>
<td>t'</td>
<td>s'</td>
<td>t'</td>
<td>s'</td>
<td>t'</td>
<td>s'</td>
<td>t'</td>
<td>s'</td>
</tr>
<tr>
<td>⊥</td>
<td>r</td>
<td>ṛ</td>
<td>ṕ</td>
<td>r</td>
<td>ṛ</td>
<td>ṕ</td>
<td>r</td>
<td>ṛ</td>
<td>ṕ</td>
<td>r</td>
<td>ṛ</td>
<td>ṕ</td>
<td>r</td>
<td>ṛ</td>
<td>ṕ</td>
<td>r</td>
</tr>
</tbody>
</table>

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
## Class Memory Automata (2/5)

### Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

<table>
<thead>
<tr>
<th>r</th>
<th>r</th>
<th>s</th>
<th>r</th>
<th>r</th>
<th>t</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \bot \quad t \quad t' \quad s' \quad s' \quad t' \quad s' \quad t' \quad s' \quad t' \quad s' \quad t' \quad s' \quad t' \quad s' \quad t' \]

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

![Diagram of class memory automaton with states and transitions]

- States are of the form \(p/q\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)

- At the end,
  - the last state should be of the form \(t\) or \(t'\)
Class Memory Automata (2/5)

Example

- Class memory automaton for the set of data strings
  - with global pattern \((r^*sr^*t)^*\),
  - with local pattern \((rst)^*\) (for each class),
  - where at most one (singular) process prints more than once

\[
\begin{array}{ccccccccccccc}
  r & r & s & r & r & t & r & s & t & s & r & t & s & t & s & t \\
  2 & 5 & 5 & 3 & 8 & 5 & 5 & 2 & 2 & 8 & 4 & 8 & 3 & 3 & 4 & 4 & 5 & 5 \\
\end{array}
\]

- States are of the form \([p, q]\), where
  - \(p\) remembers whether the singular process already has appeared and whether \(s\) or \(t\) has been seen last: \(s, t, s', t'\)
  - \(q\) is just the last symbol, (dotted if from the singular process)

- At the end,
  - the last state should be of the form \([t, t']\) or \([t', t']\)
  - each class should have a last state of the form \([t, t']\) or \([t', t']\)
Class Memory Automata (3/5)

- Class memory automata can express all properties (L1) – (L7)
- Later on we will see a precise characterization of their expressive power in terms of logic
Class Memory Automata (3/5)

- Class memory automata can express all properties (L1) – (L7)
- Later on we will see a precise characterization of their expressive power in terms of logic

Theorem 4

(a) Non-emptiness for class memory automata is decidable
(b) $\text{RegA} \subsetneq \text{ClassMA}$
Class Memory Automata (3/5)

- Class memory automata can express all properties (L1) – (L7)
- Later on we will see a precise characterization of their expressive power in terms of logic

Theorem 4

(a) Non-emptiness for class memory automata is decidable

(b) RegA ⊂ ClassMA

- The **complexity of Non-Emptiness** for class memory automata is **open**
Class Memory Automata (3/5)

- Class memory automata can express all properties (L1) – (L7)
- Later on we will see a precise characterization of their expressive power in terms of logic

Theorem 4

(a) Non-emptiness for class memory automata is decidable
(b) \[ \text{RegA} \subsetneq \text{ClassMA} \]

- The complexity of Non-Emptiness for class memory automata is open
- But there is little doubt that it is extremely bad:
  - Equivalent to Petri Net Reachability
  - Not even known to be primitive recursive
Class Memory Automata (3/5)

- Class memory automata can express all properties (L1) – (L7)
- Later on we will see a precise characterization of their expressive power in terms of logic

**Theorem 4**

(a) Non-emptiness for class memory automata is decidable

(b) $\text{RegA} \subsetneq \text{ClassMA}$

- The **complexity of Non-Emptiness** for class memory automata is open
- But there is little doubt that it is **extremely bad**:
  - Equivalent to Petri Net Reachability
  - Not even known to be primitive recursive

**Proof idea for (a) [Bojańczyk et al. 06a]**

- In a nutshell:
  - “Simulate” a class memory automaton $A$ by a (non-data) **Multicounter Automaton**:
    - String automaton $A'$ with several counters
    - $A'$ has one counter $C_q$ per state $q$ of $A$
    - $C_q$ counts the number of classes in state $q$
    - Zero tests are only needed at the end of the computation: $C_p = 0$, for $p \notin F_l$

A little bit infinite? Thomas Schwentick
Class Memory Automata (3/5)

- Class memory automata can express all properties (L1) – (L7)
- Later on we will see a precise characterization of their expressive power in terms of logic

### Theorem 4

(a) Non-emptiness for class memory automata is decidable

(b) \( \text{RegA} \subsetneq \text{ClassMA} \)

- The **complexity of Non-Emptiness** for class memory automata is **open**
- But there is little doubt that it is **extremely bad**:
  - Equivalent to Petri Net Reachability
  - Not even known to be primitive recursive

### Proof idea for (a) [Bojańczyk et al. 06a]

- In a nutshell:
  - “Simulate” a class memory automaton \( \mathcal{A} \) by a (non-data) **Multicounter Automaton**:
    - String automaton \( \mathcal{A}' \) with several counters
    - \( \mathcal{A}' \) has one counter \( C_q \) per state \( q \) of \( \mathcal{A} \)
    - \( C_q \) counts the number of classes in state \( q \)
    - Zero tests are only needed at the end of the computation: \( C_p = 0 \), for \( p \notin \mathcal{F}_l \)
  - Non-emptiness for multi-counter automata is decidable
    - Equivalent to Petri Net Reachability
    - Not even known to be primitive recursive

- And:
  - \( L(\mathcal{A}) \neq \emptyset \iff L(\mathcal{A}') \neq \emptyset \)

A little bit infinite? Thomas Schwentick

[Mayr 81]
**Proof sketch for (b) [Björklund, S 07]**

- **Strictness**: RAs cannot express (L1)
Class Memory Automata (4/5)

Proof sketch for (b) [Björklund, S 07]
- Strictness: RAs can not express (L1)
- Isn’t $RegA \subseteq ClassMA$ obvious?
### Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs cannot express (L1)
- **Isn't** \( \text{RegA} \subseteq \text{ClassMA} \) **obvious?**
- **Not entirely,** consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing \( s \) \( d \) know what happened since \( s \) \( d \) occurred last time?

A little bit infinite?
Proof sketch for (b) [Björklund, S 07]

- Strictness: RAs can not express (L1)
- Isn’t \( \text{RegA } \subseteq \text{ClassMA} \) obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing \( s_d \) know what happened since \( s_d \) occurred last time?
- **Idea:** \( A \) “colors” positions by \(+,+,−,−,+−,−+\) such that:
  - If an \( s \)-position has \(+\) the next \( s \)-position has \(−\) (and \(−\) → \(+\))
  - If an \( s \)-position has \(+\) the next \( s \)-position in the same class has \(+\)
Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs can not express (L1)
- Isn’t \( \text{RegA} \subseteq \text{ClassMA} \) obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing a state know what happened since occurred last time?

- **Idea:** A “colors” positions by such that:
  - If an \( s \)-position has the next
  - \( s \)-position has (and →)
  - If an \( s \)-position has the next \( s \)-position in the same class has

Proof sketch for (b) (cont.)

- Of course: **if such a coloring exists, (L6) holds:** the next \( s \)-position is never the next \( s \)-position in the same class

- If (L6) holds such a coloring can be constructed by applying the following rules:
Proof sketch for (b) [Björklund, S 07]

• Strictness: RAs can not express (L1)
• Isn’t RegA $\subseteq$ ClassMA obvious?
• Not entirely, consider (L6): No two successive prints by the same process
  ▶ The register automaton for (L6) only needs one state plus a sink state
  ▶ How shall a ClassMA seeing $s_d$ know what happened since $s_d$ occurred last time?

• Idea: $A$ “colors” positions by $+, -, +, -$ such that:
  ▶ If an $s$-position has $+$ the next $s$-position has $-$ (and $-$ $\rightarrow$ $+$)
  ▶ If an $s$-position has $+$ the next $s$-position in the same class has $+$

Proof sketch for (b) (cont.)

• Of course: if such a coloring exists, (L6) holds: the next $s$-position is never the next $s$-position in the same class

\[
\begin{array}{cccccccccc}
S & S & S & S & S & S & S \\
2 & 3 & 5 & 3 & 2 & 5 & 2 \\
\end{array}
\]

• If (L6) holds such a coloring can be constructed by applying the following rules:
  (1) Assign $+$ to the very last $s$ ✓
  (2) If no other rule applies: assign $+$ to the rightmost $s$ without upper color ✓
Class Memory Automata (4/5)

Proof sketch for (b) [Björklund, S 07]

- **Strictness**: RAs can not express (L1)
- Isn’t RegA $\subseteq$ ClassMA obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing $s_d$ know what happened since $s_d$ occurred last time?
- **Idea**: $A$ “colors” positions by $++, +-, -, --, +-, +-, ++$ such that:
  - If an $s$-position has $+$ the next $s$-position has $-$ (and $-\rightarrow +$)
  - If an $s$-position in the same class has $+$ the next $s$-position $\ldots$

Proof sketch for (b) (cont.)

- Of course: **if such a coloring exists, (L6) holds**: the next $s$-position is never the next $s$-position in the same class

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$+$</td>
<td>$-$</td>
<td>$+$</td>
<td>$+$</td>
<td>$-$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
</tr>
</tbody>
</table>

- If (L6) holds such a coloring can be **constructed** by applying the following rules:
  1. Assign $+$ to the very last $s$
  2. If no other rule applies: assign $+$ to the rightmost $s$ without upper color
  3. Whenever $x$ is assigned to an $s$-position assign $\bar{x}$ to its left $s$-neighbour and $x$ to the left $s$-neighbour in its class $\checkmark$

A little bit infinite?  

Thomas Schwentick  

Folie 30
### Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs can not express (L1)
- Isn’t $\text{RegA} \subseteq \text{ClassMA}$ obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing $s$ know what happened since $d$ occurred last time?
- **Idea:** $\mathcal{A}$ “colors” positions by $\begin{array}{c} +, \uparrow, +, \downarrow, -, \downarrow \end{array}$ such that:
  - If an $s$-position has $\begin{array}{c} + \end{array}$ the next $s$-position has $\begin{array}{c} - \end{array}$ (and $\begin{array}{c} - \end{array} \rightarrow \begin{array}{c} + \end{array}$)
  - If an $s$-position has $\begin{array}{c} + \end{array}$ the next $s$-position in the same class has $\begin{array}{c} + \end{array}$

### Proof sketch for (b) (cont.)

- **Of course:** if such a coloring exists, (L6) **holds:** the next $s$-position is never the next $s$-position in the same class

  $\begin{array}{cccccccc} s & s & s & s & s & s & s & s \\ 2 & 3 & 2 & 5 & 3 & 2 & 5 & 2 \end{array}$

  $\begin{array}{cccccccc} + & - & + & + \\ + & - & - \end{array}$

- If (L6) holds such a coloring can be **constructed** by applying the following rules:
  1. Assign $\begin{array}{c} + \end{array}$ to the very last $s$
  2. If no other rule applies: assign $\begin{array}{c} + \end{array}$ to the rightmost $s$ without upper color
  3. Whenever $\begin{array}{c} \bar{x} \end{array}$ is assigned to an $s$-position assign $\begin{array}{c} \bar{x} \end{array}$ to its left $s$-neighbour and $\begin{array}{c} \bar{x} \end{array}$ to the left $s$-neighbour in its class
  4. Whenever $\begin{array}{c} \bar{x} \end{array}$ is assigned to an $s$-position assign $\begin{array}{c} \bar{x} \end{array}$ to its right $s$-neighbour ✓

"A little bit infinite?"
Proof sketch for (b) [Björklund, S 07]

- Strictness: RAs can not express (L1)
- Isn’t $\text{RegA} \subseteq \text{ClassMA}$ obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing $s_d$ know what happened since $s_d$ occurred last time?
- **Idea:** $\mathcal{A}$ “colors” positions by $+, +, −, −, −, +, +, −, −, −$ such that:
  - If an $s$-position has $+$ the next $s$-position has $−$ (and $− → +$)
  - If an $s$-position has $+$ the next $s$-position in the same class has $+$

Proof sketch for (b) (cont.)

- Of course: **if such a coloring exists, (L6) holds:** the next $s$-position is never the next $s$-position in the same class

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
</tbody>
</table>

- If (L6) holds such a coloring can be constructed by applying the following rules:
  1. Assign $+$ to the very last $s$
  2. If no other rule applies: assign $+$ to the rightmost $s$ without upper color
  3. Whenever $\bar{x}$ is assigned to an $s$-position assign $\bar{x}$ to its left $s$-neighbour and $\bar{x}$ to the left $s$-neighbour in its class
  4. Whenever $\bar{x}$ is assigned to an $s$-position assign $\bar{x}$ to its right $s$-neighbour

A little bit infinite? Thomas Schwentick
Proof sketch for (b) [Björklund, S 07]

- Strictness: RAs can not express (L1)
- Isn’t RegA ⊆ ClassMA obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing $s$ know what happened since $d$ occurred last time?
- **Idea:** $A$ “colors” positions by $\{+$, $\mp$, $-$, $\pm$\} such that:
  - If an $s$-position has $+$ the next $s$-position has $-$ (and $-$ $\rightarrow$ $+$)
  - If an $s$-position has $+$ the next $s$-position in the same class has $+$

Proof sketch for (b) (cont.)

- Of course: **if such a coloring exists, (L6) holds:** the next $s$-position is never the next $s$-position in the same class
- If (L6) holds such a coloring can be constructed by applying the following rules:
  1. Assign $+$ to the very last $s$
  2. If no other rule applies: assign $+$ to the rightmost $s$ without upper color
  3. Whenever $x$ is assigned to an $s$-position assign $\bar{x}$ to its left $s$-neighbour and $x$ to the left $s$-neighbour in its class
  4. Whenever $x$ is assigned to an $s$-position assign $\bar{x}$ to its right $s$-neighbour

A little bit infinite?
### Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs cannot express (L1)
- Isn’t $\text{RegA} \subseteq \text{ClassMA}$ obvious?
- Not entirely, consider (L6): No two successive prints by the same process
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing $s_d$ know what happened since $s_d$ occurred last time?
- **Idea:** $\mathcal{A}$ “colors” positions by $++$, $+\ -\$, $-\ +\$, $--\$ such that:
  - If an $s$-position has $+$ the next $s$-position has $-$ (and $-$ $\rightarrow$ $+$)
  - If an $s$-position has $+$ the next $s$-position in the same class has $+$

### Proof sketch for (b) (cont.)

- Of course: if such a coloring exists, (L6) holds: the next $s$-position is never the next $s$-position in the same class

<table>
<thead>
<tr>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_2$</th>
<th>$s_5$</th>
<th>$s_3$</th>
<th>$s_2$</th>
<th>$s_5$</th>
<th>$s_2$</th>
<th>$s_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$-$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
</tr>
<tr>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
<td>$-$</td>
<td>$+$</td>
</tr>
</tbody>
</table>

- If (L6) holds such a coloring can be constructed by applying the following rules:
  1. Assign $+$ to the very last $s$
  2. If no other rule applies: assign $+$ to the rightmost $s$ without upper color
  3. Whenever $x$ is assigned to an $s$-position assign $x$ to its left $s$-neighbour and $\bar{x}$ to the left $s$-neighbour in its class
  4. Whenever $\bar{x}$ is assigned to an $s$-position assign $\bar{x}$ to its right $s$-neighbour
Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs can not express (L1)
- Isn’t $\text{RegA} \subseteq \text{ClassMA}$ obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing $s$ know what happened since $d$ occurred last time?
- **Idea:** $\mathcal{A}$ “colors” positions by $\mathbb{N}$ such that:
  - If an $s$-position has $+$ the next $s$-position has $-$ (and $-$ $\rightarrow$ $+$)
  - If an $s$-position has $+$ the next $s$-position in the same class has $+$

---

Proof sketch for (b) (cont.)

- **Of course:** if such a coloring exists, (L6) holds: the next $s$-position is never the next $s$-position in the same class

\[
\begin{array}{cccccccc}
\hline
s_2 & s_3 & s_2 & s_3 & s_2 & s_3 & s_2 & s_3 \\
\hline
+ & - & - & + & + & + & - & - \\
\end{array}
\]

- **If (L6) holds** such a coloring can be constructed by applying the following rules:
  1. Assign $+$ to the very last $s$
  2. If no other rule applies: assign $+$ to the rightmost $s$ without upper color
  3. Whenever $\bar{x}$ is assigned to an $s$-position assign $\bar{x}$ to its left $s$-neighbour and $x$ to the left $s$-neighbour in its class
  4. Whenever $\bar{x}$ is assigned to an $s$-position assign $\bar{x}$ to its right $s$-neighbour

A little bit infinite? Thomas Schwentick
Class Memory Automata (4/5)

Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RA cannot express (L1)
- **Isn’t** \( \text{RegA} \subseteq \text{ClassMA} \) obvious?
- **Not entirely,** consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing \( s \) know what happened since \( d \) occurred last time?
- **Idea:** \( A \) “colors” positions by \( \begin{bmatrix} + & + & - & - & \end{bmatrix} \) such that:
  - If an \( s \)-position has \( + \) the next \( s \)-position has \( - \) (and \( - \) → \( + \))
  - If an \( s \)-position in the same class has \( + \) the next

Proof sketch for (b) (cont.)

- **Of course:** if such a coloring exists, (L6) holds: the next \( s \)-position is never the next \( s \)-position in the same class

\[
\begin{array}{cccccccc}
S_2 & S_3 & S_2 & S_5 & S_3 & S_2 & S_5 & S_3 \\
\hline
\begin{bmatrix} + & - & - & + & + \\ - & + & + & - & - & + \end{bmatrix}
\end{array}
\]

- **If** (L6) holds such a coloring can be constructed by applying the following rules:
  1. Assign \( + \) to the very last \( s \)
  2. If no other rule applies: assign \( + \) to the rightmost \( s \) without upper color
  3. Whenever \( \overline{x} \) is assigned to an \( s \)-position assign \( \overline{x} \) to its left \( s \)-neighbour and \( \overline{x} \) to the left \( s \)-neighbour in its class
  4. Whenever \( \overline{x} \) is assigned to an \( s \)-position assign \( \overline{x} \) to its right \( s \)-neighbour

A little bit infinite? Thomas Schwentick tu.
Class Memory Automata (4/5)

Proof sketch for (b) [Björklund, S 07]

- Strictness: RAs can not express (L1)
- Isn’t RegA ⊆ ClassMA obvious?
- Not entirely, consider (L6): No two successive prints by the same process
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing what happened since occurred last time?
- Idea: A “colors” positions by such that:
  - If an s-position has the next
    - s-position has (and → )
  - If an s-position has the next
    - s-position in the same class has

Proof sketch for (b) (cont.)

- Of course: if such a coloring exists, (L6) holds: the next s-position is never the next s-position in the same class

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
</tr>
</tbody>
</table>

- If (L6) holds such a coloring can be constructed by applying the following rules:
  1. Assign to the very last s
  2. If no other rule applies: assign + to the rightmost s without upper color
  3. Whenever is assigned to an s-position assign to its left s-neighbour and to the left s-neighbour in its class ✓
  4. Whenever is assigned to an s-position assign to its right s-neighbour

A little bit infinite? Thomas Schwentick
### Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs can not express (L1)
- **Isn't** RegA $\subseteq$ ClassMA obvious?
- **Not entirely,** consider (L6):
  - **No two successive prints by the same process**
    - The register automaton for (L6) only needs one state plus a sink state.
    - How shall a ClassMA seeing $s_d$ know what happened since $s_d$ occurred last time?

- **Idea:** $\mathcal{A}$ “colors” positions by $++, +-, -, +, −−$ such that:
  - If an $s$-position has $++$ the next $s$-position has $-$ (and $−→+$)
  - If an $s$-position has $+$ the next $s$-position in the same class has $+$

### Proof sketch for (b) (cont.)

- **Of course:** if such a coloring exists, (L6) **holds:** the next $s$-position is never the next $s$-position in the same class.

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
<th>$s_6$</th>
<th>$s_7$</th>
<th>$s_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2$</td>
<td>$3$</td>
<td>$2$</td>
<td>$5$</td>
<td>$3$</td>
<td>$2$</td>
<td>$5$</td>
<td>$2$</td>
</tr>
</tbody>
</table>

- **If (L6) holds such a coloring can be constructed** by applying the following rules:
  1. Assign $+$ to the very last $s$.
  2. If no other rule applies: assign $+$ to the rightmost $s$ without upper color.
  3. Whenever $x$ is assigned to an $s$-position assign $\bar{x}$ to its left $s$-neighbour and $x$ to the left $s$-neighbour in its class.
  4. Whenever $\bar{x}$ is assigned to an $s$-position assign $x$ to its right $s$-neighbour.

---

A little bit infinite? Thomas Schwentick
Proof sketch for (b) [Björklund, S 07]

- Strictness: RAs can not express (L1)
- Isn’t $\text{RegA} \subseteq \text{ClassMA}$ obvious?
- Not entirely, consider (L6): No two successive prints by the same process
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing $s_d$ know what happened since $s_d$ occurred last time?

- Idea: $\mathcal{A}$ “colors” positions by $\begin{array}{c} +, +, -, -, - \\ +, - +, - \\ +, - +, - \\ - +, - +, - \end{array}$ such that:
  - If an $s$-position has $\begin{array}{c} + \\ + \\ - \\ + \\ + \end{array}$ the next $s$-position has $\begin{array}{c} - \\ - \rightarrow + \\ + \end{array}$ (and $\begin{array}{c} - \\ - \rightarrow + \\ + \end{array}$)
  - If an $s$-position has $\begin{array}{c} + \\ + \\ + \end{array}$ the next $s$-position in the same class has $\begin{array}{c} + \\ + \end{array}$

Proof sketch for (b) (cont.)

- Of course: if such a coloring exists, (L6) holds: the next $s$-position is never the next $s$-position in the same class
  - $\begin{array}{cccccccc} s_2 & s_3 & s_2 & s_3 & s_2 & s_3 & s_2 & s_3 \\ + & + & - & - & + & + & - & - \end{array}$

- If (L6) holds such a coloring can be constructed by applying the following rules:
  1. Assign $\begin{array}{c} + \end{array}$ to the very last $s$
  2. If no other rule applies: assign $\begin{array}{c} + \end{array}$ to the rightmost $s$ without upper color
  3. Whenever $\begin{array}{c} x \end{array}$ is assigned to an $s$-position assign $\begin{array}{c} \bar{x} \end{array}$ to its left $s$-neighbour and $\begin{array}{c} \bar{x} \end{array}$ to the left $s$-neighbour in its class
  4. Whenever $\begin{array}{c} x \end{array}$ is assigned to an $s$-position assign $\begin{array}{c} \bar{x} \end{array}$ to its right $s$-neighbour
Proof sketch for (b) [Björklund, S 07]

- Strictness: RAs can not express (L1)
- Isn’t $\text{RegA} \subseteq \text{ClassMA}$ obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing $s \rightarrow d$ know what happened since $d \rightarrow s$ occurred last time?
- **Idea:** $\mathcal{A}$ “colors” positions by $+, -, ++, --$ such that:
  - If an $s$-position has $+$ the next $s$-position has $-$ (and $-$ $\rightarrow$ $+$)
  - If an $s$-position in the same class has $+$ the next $s$-position has $+$

Proof sketch for (b) (cont.)

- Of course: **if such a coloring exists, (L6) holds:** the next $s$-position is never the next $s$-position in the same class

\[
\begin{array}{cccccccc}
S & S & S & S & S & S & S & S \\
2 & 3 & 2 & 5 & 3 & 2 & 5 & 3 \\
\hline
- & + & + & - & - & + & + & + \\
+ & - & - & + & + & - & - & -
\end{array}
\]

- **If (L6) holds such a coloring can be constructed** by applying the following rules:
  1. Assign $+$ to the very last $s$
  2. If no other rule applies: assign $+$ to the rightmost $s$ without upper color
  3. Whenever $\bar{x}$ is assigned to an $s$-position assign $x$ to its left $s$-neighbour and $\bar{x}$ to the left $s$-neighbour in its class
  4. Whenever $\bar{x}$ is assigned to an $s$-position assign $\bar{x}$ to its right $s$-neighbour
### Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs can not express (L1)
- Isn’t \( \text{RegA} \subseteq \text{ClassMA} \) obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing \( s \) know what happened since \( d \) occurred last time?
- **Idea:** \( \mathcal{A} \) “colors” positions by \( \text{+}, \text{+}, \text{-}, \text{-} \)
  - such that:
    - If an \( s \)-position has \( \text{+} \) the next \( s \)-position has \( \text{-} \) (and \( \text{-} \rightarrow \text{+} \))
    - If an \( s \)-position has \( \text{+} \) the next \( s \)-position in the same class has \( \text{+} \)

### Proof sketch for (b) (cont.)

- Of course: **if such a coloring exists, (L6) holds:** the next \( s \)-position is never the next \( s \)-position in the same class
  
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

- **If (L6) holds such a coloring can be constructed** by applying the following rules:
  1. Assign \( \text{+} \) to the very last \( s \)
  2. If no other rule applies: assign \( \text{+} \) to the rightmost \( s \) without upper color
  3. Whenever \( \bar{x} \) is assigned to an \( s \)-position assign \( \bar{x} \) to its left \( s \)-neighbour and \( \bar{x} \) to the left \( s \)-neighbour in its class
  4. Whenever \( \bar{x} \) is assigned to an \( s \)-position assign \( \bar{x} \) to its right \( s \)-neighbour

A little bit infinite? Thomas Schwentick
Proof sketch for (b) [Björklund, S 07]

- **Strictness**: RAs can not express (L1)
- Isn’t \( \text{RegA} \subseteq \text{ClassMA} \) obvious?
- Not entirely, consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing \( s \) know what happened since \( d \) occurred last time?
- **Idea**: \( \mathcal{A} \) “colors” positions by \(+, +, -, -\) such that:
  - If an \( s \)-position has \(+\) the next \( s \)-position has \(-\) (and \(-\rightarrow +\))
  - If an \( s \)-position has \(+\) the next \( s \)-position in the same class has \(+\)

Proof sketch for (b) (cont.)

- **Of course**: if such a coloring exists, (L6) **holds**: the next \( s \)-position is never the next \( s \)-position in the same class

\[
\begin{array}{cccccccc}
S & S & S & S & S & S & S & S \\
2 & 3 & 2 & 5 & 3 & 2 & 5 & 2 \\
\end{array}
\]

- If (L6) holds such a coloring can be **constructed** by applying the following rules:
  1. Assign \(+\) to the very last \( s \)
  2. If no other rule applies: assign \(+\) to the rightmost \( s \) without upper color
  3. Whenever \( \bar{x} \) is assigned to an \( s \)-position assign \( x \) to its left \( s \)-neighbour and \( x \) to the left \( s \)-neighbour in its class
  4. Whenever \( \bar{x} \) is assigned to an \( s \)-position assign \( x \) to its right \( s \)-neighbour

A little bit infinite? Thomas Schwentick tuition Folie 30
Class Memory Automata (4/5)

Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs can not express (L1)
- **Isn't** RegA \( \subseteq \) ClassMA **obvious?**
- **Not entirely,** consider (L6): **No two successive prints by the same process**
  - The register automaton for (L6) only needs one state plus a sink state
  - How shall a ClassMA seeing \( s \) know what happened since \( d \) occurred last time?

**Idea:** \( \mathcal{A} \) “colors” positions by \( +, ++, --, -+ \) such that:

<table>
<thead>
<tr>
<th>( + )</th>
<th>( ++ )</th>
<th>( -- )</th>
<th>( -+ )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( + )</td>
<td>( ++ )</td>
<td>( -- )</td>
<td>( -+ )</td>
</tr>
<tr>
<td>( + )</td>
<td>( ++ )</td>
<td>( -- )</td>
<td>( -+ )</td>
</tr>
</tbody>
</table>

- \( \mathcal{A} \) assigns to the very last \( s \)
- If no other rule applies: assign \( + \) to the rightmost \( s \) without upper color
- Whenever \( \bar{x} \) is assigned to an \( s \)-position assign \( \bar{x} \) to its left \( s \)-neighbour and \( \bar{x} \) to the left \( s \)-neighbour in its class
- Whenever \( \bar{x} \) is assigned to an \( s \)-position assign \( \bar{x} \) to its right \( s \)-neighbour

Proof sketch for (b) (cont.)

- **Of course:** if such a coloring exists, (L6) **holds:** the next \( s \)-position is never the next \( s \)-position in the same class

<table>
<thead>
<tr>
<th>( s )</th>
<th>( s )</th>
<th>( s )</th>
<th>( s )</th>
<th>( s )</th>
<th>( s )</th>
<th>( s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( + )</td>
<td>( + )</td>
<td>( -- )</td>
<td>( + )</td>
<td>( + )</td>
<td>( + )</td>
<td>( + )</td>
</tr>
<tr>
<td>( + )</td>
<td>( + )</td>
<td>( -- )</td>
<td>( + )</td>
<td>( -- )</td>
<td>( + )</td>
<td>( + )</td>
</tr>
<tr>
<td>( + )</td>
<td>( + )</td>
<td>( -- )</td>
<td>( + )</td>
<td>( -- )</td>
<td>( + )</td>
<td>( + )</td>
</tr>
</tbody>
</table>

= A little bit infinite? Thomas Schwentick
Proof sketch for (b) [Björklund, S 07]

- **Strictness:** RAs can not express (L1)
- **Isn’t** RegA ⊆ ClassMA obvious?
- **Not entirely,** consider (L6):
  - **No two successive prints by the same process**
    - The register automaton for (L6) only needs one state plus a sink state
    - How shall a ClassMA seeing $s_d$ know what happened since $s_d$ occurred last time?

**Idea:** $A$ “colors” positions by $++, +-, -+, --$ such that:

- If an $s$-position has $+$ the next $s$-position has $-$ (and $--$)$\rightarrow$ $++$)
- If an $s$-position has $+$ the next $s$-position in the same class has $+$

Proof sketch for (b) (cont.)

- Of course: **if such a coloring exists, (L6) holds:** the next $s$-position is never the next $s$-position in the same class

```
  s_2 s_3 s_2 s_5 s_3 s_2 s_5 s_2 s_3
  + + - + + - - + + + + - - - - + +
```

- **If (L6) holds such a coloring can be constructed** by applying the following rules:
  1. Assign $+$ to the very last $s$
  2. If no other rule applies: assign $+$ to the rightmost $s$ without upper color
  3. Whenever $x$ is assigned to an $s$-position assign $\bar x$ to its left $s$-neighbour and $\bar x$ to the left $s$-neighbour in its class
  4. Whenever $\bar x$ is assigned to an $s$-position assign $\bar x$ to its right $s$-neighbour

**General proof of (b): similar coloring trick**
## Class Memory Automata (5/5)

<table>
<thead>
<tr>
<th></th>
<th>RegisterA</th>
<th>PebbleA</th>
<th>ClassMA</th>
<th>DClassMA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressiveness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(L2),(L6),(L7)</td>
<td>(L1)–(L7)</td>
<td>(L1)–(L7)</td>
<td>(L1)–(L5),(L7)</td>
</tr>
<tr>
<td><strong>Decidability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-emptiness</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Containment</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data complexity word pr.</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Closure properties</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Intersection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complement</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>
Inclusion structure of Automata Models

A little bit infinite? Thomas Schwentick
Contents

Introduction
Data Model
Automata

Logic

➢ Two-Variable Logics

Temporal Logics

Other Models

Conclusion
Logics for Data Strings/Trees

- **Automata** offer an algorithmic framework
- **Logics** offer a framework for declarative specifications
Logics for Data Strings/Trees

- **Automata** offer an algorithmic framework
- **Logics** offer a framework for declarative specifications

**We will consider:**
- Restrictions of classical first-order logic
- Extensions of temporal logics
Logics for Data Strings/Trees

- **Automata** offer an algorithmic framework
- **Logics** offer a framework for declarative specifications

**We will consider:**
- Restrictions of classical first-order logic
- Extensions of temporal logics

<table>
<thead>
<tr>
<th>Logical language...</th>
<th>... for strings</th>
<th>... for trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a(x))</td>
<td>Letter at position (x) is (a \in \Sigma)</td>
<td>(a(x)) Label of node (x) is (a \in \Sigma)</td>
</tr>
<tr>
<td>(+1)</td>
<td>successor relation on positions</td>
<td>(E\rightarrow) horizontal neighbor (&quot;next sibling&quot;)</td>
</tr>
<tr>
<td>(&lt;)</td>
<td>order relation on positions</td>
<td>(E\downarrow) parent-child</td>
</tr>
<tr>
<td>(\sim)</td>
<td>(x \sim y) if positions (x) and (y) have the same (D)-value</td>
<td>(\sim) if nodes (x) and (y) have the same (D)-value</td>
</tr>
<tr>
<td>(\pm1)</td>
<td>next position in the same class</td>
<td></td>
</tr>
</tbody>
</table>

A little bit infinite? Thomas Schwentick  
Folie 34
Logics for Data Strings/Trees

- **Automata** offer an algorithmic framework
- **Logics** offer a framework for declarative specifications
- **We will consider:**
  - Restrictions of classical first-order logic
  - Extensions of temporal logics

<table>
<thead>
<tr>
<th>Logical language...</th>
<th>... for strings</th>
<th>... for trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a(x)$</td>
<td>Letter at position $x$ is $a \in \Sigma$</td>
<td>$a(x)$ Label of node $x$ is $a \in \Sigma$</td>
</tr>
<tr>
<td>$\perp$</td>
<td>successor relation on positions</td>
<td>$E \rightarrow$ horizontal neighbor (&quot;next sibling&quot;)</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>order relation on positions</td>
<td>$E \rightarrow$ transitive closure of $E \rightarrow$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$E \downarrow$ parent-child</td>
</tr>
<tr>
<td>$\sim$</td>
<td>$x \sim y$ if positions $x$ and $y$ have the same $D$-value</td>
<td>$\sim$ if nodes $x$ and $y$ have the same $D$-value</td>
</tr>
<tr>
<td>$\perp$</td>
<td>next position in the same class</td>
<td></td>
</tr>
</tbody>
</table>

- Of course: $\sim$ **is an equivalence relation**
**Logics for Data Strings/Trees**

- **Automata** offer an algorithmic framework
- **Logics** offer a framework for declarative specifications
- **We will consider:**
  - Restrictions of classical first-order logic
  - Extensions of temporal logics

<table>
<thead>
<tr>
<th>Logical language...</th>
<th>... for strings</th>
<th>... for trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a(x)$</td>
<td>Letter at position $x$ is $a \in \Sigma$</td>
<td>$a(x)$ Label of node $x$ is $a \in \Sigma$</td>
</tr>
<tr>
<td>$+1$</td>
<td>successor relation on positions</td>
<td>$E \rightarrow$ horizontal neighbor (&quot;next sibling&quot;)</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>order relation on positions</td>
<td>$E \Rightarrow$ transitive closure of $E \rightarrow$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$E \downarrow$ transitive closure of $E \downarrow$</td>
</tr>
<tr>
<td>$\sim$</td>
<td>$x \sim y$ if positions $x$ and $y$ have the same $D$-value</td>
<td>$\sim$ if nodes $x$ and $y$ have the same $D$-value</td>
</tr>
<tr>
<td>$\pm 1$</td>
<td>next position in the same class</td>
<td></td>
</tr>
</tbody>
</table>

- Of course: $\sim$ is an equivalence relation
- No other operations on data values, in particular no arithmetic!
We know:

- First-order logic is undecidable in general
A first attempt

- We know:
  - First-order logic is undecidable in general
  - First-order logic is decidable on strings
A first attempt

- We know:
  - First-order logic is undecidable in general
  - First-order logic is decidable on strings
- What about First-order logic on data strings?
A first attempt

- We know:
  - First-order logic is undecidable in general
  - First-order logic is decidable on strings
- What about First-order logic on data strings?

**Theorem 5 [Bojańczyk et al. 06a]**

- Satisfiability of First-Order formulas on data strings is undecidable, even for formulas with 3 variables

**Proof idea**

- Reduction from PCP:
  - Given: \((u_1, v_1), \ldots, (u_k, v_k)\), pairs of strings
  - Question: is there a sequence \(i_1, \ldots, i_n\) such that \(u_{i_1} \cdots u_{i_n} = v_{i_1} \cdots v_{i_n}\)?
A first attempt

- We know:
  - First-order logic is undecidable in general
  - First-order logic is decidable on strings
- What about First-order logic on data strings?

Theorem 5 [Bojańczyk et al. 06a]
- Satisfiability of First-Order formulas on data strings is undecidable, even for formulas with 3 variables

Proof idea
- Reduction from PCP:
  - Given: \((u_1, v_1), \ldots, (u_k, v_k)\), pairs of strings
  - Question: is there a sequence \(i_1, \ldots, i_n\) such that \(u_i_1 \cdots u_i_n = v_i_1 \cdots v_i_n\)?

A bit more detail
- Encode solution candidates as data strings over \(\{a, b, \#, 1, \ldots, k\}\) of the form \(u \# v\)
- Each occurrence of a \(u_i\) is prefixed by \(i\):
  - E.g., if \(u_1 = aba\) and \(u_2 = bb\) then \(121\) induces \(1aba2bb1aba\)
- Each data value occurs exactly twice, once in \(u\) and once in \(v\)
  - corresponding positions should have the same data value (and same number/symbol)
- Crucial: check that the sequence of data values is the same on both sides for number positions and letter positions
  - Important subformula:
    \[
    x \sim y \rightarrow \exists z \left( (x + 1 = z \land \exists x \ y + 1 = x \land z \sim x) \right)
    \]
    "if \(x\) and \(y\) are equivalent then their right neighbors are also equivalent"

A little bit infinite? Thomas Schwentick
Two Variables on Data Strings: A Useful Restriction?

- A classical approach: Restriction to 2 variables
- Does this restriction give us anything useful?
Two Variables on Data Strings: A Useful Restriction?

- A classical approach: Restriction to 2 variables
- Does this restriction give us anything useful?
  1. We do not have free choice...
  2. Lot of useful properties can be expressed with only two variables
Two Variables on Data Strings: A Useful Restriction?

- **A classical approach:** Restriction to 2 variables
- **Does this restriction give us anything useful?**
  1. We do not have free choice...
  2. A lot of useful properties can be expressed with only two variables

### Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
</table>
| (L1) | No two \(a\)-positions do have the same data value  
\[
\forall x \forall y (x \sim y \land a(x) \land a(y)) \rightarrow x = y
\] |
| (L2) | There are two \(a\)-positions with the same data value  
\[
\exists x \exists y x \sim y \land a(x) \land a(y) \land x \neq y
\] |
| (L3) | For each \(a\)-position there is a \(b\)-position with the same data value  
\[
\forall x \exists y a(x) \rightarrow (b(y) \land x \sim y)
\] |
| (L4) | A print job of a user has to be printed before the next one can be requested  
\[
\forall x \forall y y = x \pm 1 \rightarrow [(r(x) \rightarrow s(s)) \land (s(x) \rightarrow t(y))]
\] |
| (L5) | Each print request of a user is eventually followed by a print  
\[
\forall x \exists y r(x) \rightarrow (s(y) \land x < y \land x \sim y)
\] |
| (L6) | Between two successive print jobs of the same user some other user’s job has to be printed  
(not expressible) |
| (L7) | After each printed job a job of some other user is eventually printed  
\[
\forall x \exists y r(x) \rightarrow (s(y) \land x < y \land x \not\sim y)
\] |
On the expressive power of $\text{FO}^2$ on data strings (1/2)

Example

- $\varphi_a$:
  - $\forall x \forall y (x \sim y \land a(x) \land a(y)) \rightarrow x = y$
  - all $a$'s are in different classes
- Similarly: $\varphi_b$
- $\psi_{a,b}$:
  - $\psi_{a,b} = \forall x \exists y (a(x) \rightarrow (b(y) \land x \sim y))$
  - each class with an $a$ also contains a $b$
- Similarly: $\psi_{b,a}$.
### Example

- **\( \varphi_a : \)**
  - \( \forall x \, \forall y \, (x \sim y \land a(x) \land a(y)) \rightarrow x = y \)
  - all \( a \)'s are in different classes
- Similarly: \( \varphi_b \)

- **\( \psi_{a,b} : \)**
  - \( \psi_{a,b} = \forall x \, \exists y \, (a(x) \rightarrow (b(y) \land x \sim y)) \)
  - each class with an \( a \) also contains a \( b \)
- Similarly: \( \psi_{b,a} \).

\( \varphi = \varphi_a \land \varphi_b \land \psi_{a,b} \land \psi_{b,a} \) implies:
- the numbers of \( a \) and \( b \)-labeled positions are equal

- In a similar fashion: number of \( a \)'s, \( b \)'s and \( c \)'s are equal
### Example

- **φ_a:**
  - \( \forall x \ \forall y \ (x \sim y \land a(x) \land a(y)) \implies x = y \)
  - all \( a \)'s are in different classes

- Similarly: **φ_b**

- **ψ_{a,b}:**
  - \( \psi_{a,b} = \forall x \ \exists y \ (a(x) \implies (b(y) \land x \sim y)) \)
  - each class with an \( a \) also contains a \( b \)

- Similarly: **ψ_{b,a}**.

\[ \implies \quad \varphi = \varphi_a \land \varphi_b \land \psi_{a,b} \land \psi_{b,a} \implies \text{the numbers of } a \text{- and } b \text{-labeled positions are equal} \]

- In a similar fashion: number of \( a \)'s, \( b \)'s and \( c \)'s are equal

\[ \implies \text{The string projection of an } \text{FO}^2 \text{-definable data language need not be context-free} \]
More example properties

- Let $\alpha$ and $\beta$ denote unary quantifier-free formulas ("types")
- $\text{FO}^2$ can express

A little bit infinite? Thomas Schwentick
### More example properties

- Let $\alpha$ and $\beta$ denote unary quantifier-free formulas ("types")
- $\text{FO}^2$ can express
  - data-blind properties, i.e., properties not using $\sim$
More example properties

- Let $\alpha$ and $\beta$ denote unary quantifier-free formulas (“types”)
- $\mathbf{FO}^2$ can express
  - data-blind properties, i.e., properties not using $\sim$
  - Each class contains at most one occurrence of $\alpha$:
    \[ \theta = \forall x \forall y \left( (\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y \right) \]
On the expressive power of \( \text{FO}^2 \) on data strings (2/2)

More example properties

- Let \( \alpha \) and \( \beta \) denote unary quantifier-free formulas ("types")
- \( \text{FO}^2 \) can express
  - data-blind properties, i.e., properties not using \( \sim \)
  - Each class contains at most one occurrence of \( \alpha \):
    \[
    \theta = \forall x \forall y \left( (\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y \right)
    \]
  - In each class, every \( \alpha \) occurs before every \( \beta \):
    \[
    \theta = \forall x \forall y \left( (\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x < y \right)
    \]
More example properties

- Let $\alpha$ and $\beta$ denote unary quantifier-free formulas (“types”)
- $\text{FO}^2$ can express
  - data-blind properties, i.e., properties not using $\sim$
  - Each class contains at most one occurrence of $\alpha$:
    \[ \theta = \forall x \forall y \left( (\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y \right) \]
  - In each class, every $\alpha$ occurs before every $\beta$:
    \[ \theta = \forall x \forall y \left( (\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x < y \right) \]
  - Each class with an $\alpha$ also has a $\beta$:
    \[ \theta = \forall x \exists y \left( \alpha(x) \rightarrow (\beta(y) \land x \sim y) \right) \]
More example properties

- Let $\alpha$ and $\beta$ denote unary quantifier-free formulas ("types")
- $\text{FO}^2$ can express
  - data-blind properties, i.e., properties not using $\sim$
  - Each class contains at most one occurrence of $\alpha$:
    $$\theta = \forall x \forall y ((\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y)$$
  - In each class, every $\alpha$ occurs before every $\beta$:
    $$\theta = \forall x \forall y ((\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x < y)$$
  - Each class with an $\alpha$ also has a $\beta$:
    $$\theta = \forall x \exists y (\alpha(x) \rightarrow (\beta(y) \land x \sim y))$$
  - If a position is in a different class than its successor it has type $\alpha$:
    $$\theta = \forall x \forall y (\neg(x \sim y) \land x + 1 = y) \rightarrow \alpha(x)$$
### More example properties

- Let $\alpha$ and $\beta$ denote unary quantifier-free formulas (“types”)
- $\mathbf{FO}^2$ can express
  - **Data-blind** properties, i.e., properties not using $\sim$
  - Each class contains at most one occurrence of $\alpha$:
    \[ \theta = \forall x \forall y ((\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y) \]
  - In each class, every $\alpha$ occurs before every $\beta$:
    \[ \theta = \forall x \forall y ((\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x < y) \]
  - Each class with an $\alpha$ also has a $\beta$:
    \[ \theta = \forall x \exists y (\alpha(x) \rightarrow (\beta(y) \land x \sim y)) \]
  - If a position is in a different class than its successor it has type $\alpha$:
    \[ \theta = \forall x \forall y (\neg (x \sim y) \land x + 1 = y) \rightarrow \alpha(x) \]

- **That’s basically all!**
More example properties

- Let $\alpha$ and $\beta$ denote unary quantifier-free formulas ("types")
- $\text{FO}^2$ can express
  - data-blind properties, i.e., properties not using $\sim$
  - Each class contains at most one occurrence of $\alpha$:
    $$\theta = \forall x \forall y \left((\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y\right)$$
  - In each class, every $\alpha$ occurs before every $\beta$:
    $$\theta = \forall x \forall y \left((\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x < y\right)$$
  - Each class with an $\alpha$ also has a $\beta$:
    $$\theta = \forall x \exists y \left(\alpha(x) \rightarrow (\beta(y) \land x \sim y)\right)$$
  - If a position is in a different class than its successor it has type $\alpha$:
    $$\theta = \forall x \forall y \left(\neg(x \sim y) \land x + 1 = y\right) \rightarrow \alpha(x)$$
- That's basically all!

Theorem 6 [Bojańczyk et al. 06a]

Satisfiability of $\text{FO}^2(\sim, <, +1, \neq 1)$ on data strings is decidable
Proof Sketch for Theorem 6 (1/2)

Scott and intermediate normal form

- We transform two-variable formulas into satisfiability equivalent formulas of **existential monadic second-order logic**
- "Scott normal form": \( \exists R_1, \ldots, R_k \ \forall x \ \forall y \ \chi \ \land \ \land_i \ \forall x \exists y \ \chi_i \)
- Intermediate normal form:
  \[ \exists R_1 \cdots R_m \ \theta_1 \land \cdots \land \theta_n \]
- \( \theta_i \):
  1. \( \forall x \forall y \ \left( \delta(x, y) \geq 2 \land \alpha(x) \land \beta(y) \land \begin{array}{c} x \sim y \\ x \not\sim y \end{array} \right) \rightarrow \begin{array}{c} x < y \\ x > y \end{array} \)
  2. \( \forall x \exists y \ \alpha(x) \rightarrow (\beta(y) \land \begin{array}{c} x + 1 < y \\ x + 1 = y \\ x = y \\ x = y + 1 \\ x > y + 1 \end{array}) \land \begin{array}{c} x \sim y \\ x \not\sim y \end{array} \)
  3. \( \forall x \forall y \ \theta \) (\( \theta \) quantifier-free, DNF, no \( \sim \))
- Both steps are straightforward
Proof Sketch for Theorem 6 (2/2)

Data normal form & Class Memory Automata

- **Data normal form:**
  
  \[ \exists R_1 \ldots R_n \theta_1 \land \ldots \land \theta_n \]

- \( \theta_i \):
  
  (a) data-blind
  (b) Each class contains at most one \( \alpha \)
  (c) In each class, every \( \alpha \) occurs before every \( \beta \)
  (d) Each class with an \( \alpha \) also has a \( \beta \)
  (e) If \( x \) is in a different class than its successor has type \( \alpha \)
Data normal form & Class Memory Automata

- **Data normal form:**
  - Disjunction of formulas: \( \exists R_1 \cdots R_n \ \theta_1 \land \cdots \land \theta_n \)
  - \( \theta_i \):
    - (a) data-blind
    - (b) Each class contains at most one \( \alpha \)
    - (c) In each class, every \( \alpha \) occurs before every \( \beta \)
    - (d) Each class with an \( \alpha \) also has a \( \beta \)
    - (e) If \( x \) is in a different class than its successor has type \( \alpha \)

- **Final Step:**
  - Each \( \theta_i \) can be recognized by a Class Memory Automaton
  - Existential monadic quantification corresponds to nondeterminism in CMAs
  - CMAs are closed under union and intersection
  - Formulas in data normal form can be effectively translated into Class Memory Automata
Proof Sketch for Theorem 6 (2/2)

Data normal form & Class Memory Automata

- **Data normal form:**
  - Disjunction of formulas: \( \exists R_1 \cdots R_n \, \theta_1 \land \cdots \land \theta_n \)
  - \( \theta_i \):
    - (a) data-blind
    - (b) Each class contains at most one \( \alpha \)
    - (c) In each class, every \( \alpha \) occurs before every \( \beta \)
    - (d) Each class with an \( \alpha \) also has a \( \beta \)
    - (e) If \( x \) is in a different class than its successor has type \( \alpha \)

- **Final Step:**
  - Each \( \theta_i \) can be recognized by a Class Memory Automaton
  - Existential monadic quantification corresponds to nondeterminism in CMAs
  - CMAs are closed under union and intersection
  - Formulas in data normal form can be effectively translated into Class Memory Automata

- Decidability of \( \text{FO}^2(\sim, <, +1, \pm 1) \) follows from decidability of Non-emptiness for Class Memory Automata
Data normal form & Class Memory Automata

- **Data normal form:**
  - Disjunction of formulas $\exists R_1 \cdots R_n \theta_1 \land \cdots \land \theta_n$
  - $\theta_i$:
    - (a) data-blind
    - (b) Each class contains at most one $\alpha$
    - (c) In each class, every $\alpha$ occurs before every $\beta$
    - (d) Each class with an $\alpha$ also has a $\beta$
    - (e) If $x$ is in a different class than its successor has type $\alpha$

- **Final Step:**
  - Each $\theta_i$ can be recognized by a Class Memory Automaton
  - Existential monadic quantification corresponds to nondeterminism in CMAs
  - CMAs are closed under union and intersection
  - Formulas in data normal form can be effectively translated into Class Memory Automata

- **Decidability of $\text{FO}^2(\sim, <, +1, \neq 1)$ follows from decidability of Non-emptiness for Class Memory Automata

- **Corollary:** $\text{ClassMA} \equiv \text{EMSO}^2(\sim, <, +1, \neq 1)$
• Complexitywise, Satisfiability of $\text{FO}^2(\sim, <, +1)$ is basically equivalent to Non-Emptiness of multicounter automata

$\Rightarrow$ Unknown complexity
\( \text{FO}^2 \) on Data Strings: Complexity

- Complexitywise, Satisfiability of \( \text{FO}^2(\sim, <, +1) \) is basically equivalent to Non-Emptiness of multicounter automata

  \[ \rightarrow \text{Unknown complexity} \]

- Restrictions:
  - \( \text{FO}^2(\sim, <) \): complete for \text{NEXPTIME} [David 04]
  - \( \text{FO}^2(\sim, +1) \): in \text{3NEXPTIME} [Bojańczyk et al. 06b]
Complexitywise, Satisfiability of $\mathbf{FO}^2(\sim, <, +1)$ is basically equivalent to Non-Emptiness of multicounter automata

$\Rightarrow$ Unknown complexity

Restrictions:

- $\mathbf{FO}^2(\sim, \prec)$: complete for $\text{NEXPTIME}$ [David 04]
- $\mathbf{FO}^2(\sim, +1)$: in $3\text{NEXPTIME}$ [Bojańczyk et al. 06b]

Extensions:

- $+2, +3, \ldots$: same results
- $\omega$-strings: same results
- Linear order on data values: undecidable
Theorem 7 [Bojańczyk et al. 06b]

For any vector addition tree automaton $A$, a formula $\varphi_A \in \text{FO}^2(\sim, <, +1)$ can be computed such that:

$L(A) \neq \emptyset$ iff $\varphi_A$ has a model.
Two-Variable Logic on Data Trees

**Theorem 7 [Bojańczyk et al. 06b]**

For any vector addition tree automaton $A$, a formula $\varphi_A \in \text{FO}^2(\sim, <, +1)$ can be computed such that:

$$L(A) \neq \emptyset \text{ iff } \varphi_A \text{ has a model}$$

- Decidability of emptiness of vector addition tree automata is an open problem.
- It is equivalent to decidability of Multiplicative Exponential Linear Logic.
- We concentrate on $\text{FO}^2(\sim, +1)$
Theorem 7 [Bojańczyk et al. 06b]

For any vector addition tree automaton \( A \), a formula \( \varphi_A \in \text{FO}^2(\sim, <, +1) \) can be computed such that:

\[ L(A) \neq \emptyset \text{ iff } \varphi_A \text{ has a model} \]

- Decidability of emptiness of vector addition tree automata is an open problem
- It is equivalent to decidability of Multiplicative Exponential Linear Logic
- We concentrate on \( \text{FO}^2(\sim, +1) \)

Theorem 8 [Bojańczyk et al. 06b]

Satisfiability of \( \text{FO}^2(\sim, +1) \) on data trees is decidable
## Two-Variable Logic on Data Trees

<table>
<thead>
<tr>
<th>Theorem 7 [Bojańczyk et al. 06b]</th>
<th>Theorem 8 [Bojańczyk et al. 06b]</th>
</tr>
</thead>
<tbody>
<tr>
<td>For any <strong>vector addition tree automaton</strong> $A$, a formula $\varphi_A \in \mathbf{FO}^2(\sim, &lt;, +1)$ can be computed such that: $L(A) \neq \emptyset$ iff $\varphi_A$ has a model</td>
<td>Satisfiability of $\mathbf{FO}^2(\sim, +1)$ on data trees is decidable</td>
</tr>
<tr>
<td>- Decidability of emptiness of vector addition tree automata is an open problem</td>
<td>- The intermediate steps of the proof are similar as for data strings</td>
</tr>
<tr>
<td>- It is equivalent to decidability of Multiplicative Exponential Linear Logic</td>
<td>- But additional techniques needed:</td>
</tr>
<tr>
<td>$\rightarrow$ We concentrate on $\mathbf{FO}^2(\sim, +1)$</td>
<td>$\rightarrow$ Model normalization by cut-and-paste arguments</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$ Canonical “small” models that can be recognized by simpler tree automata</td>
</tr>
</tbody>
</table>

A little bit infinite? Thomas Schwentick
## Two-Variable Logic on Data Trees

### Theorem 7 [Bojańczyk et al. 06b]

For any **vector addition tree automaton** $A$, a formula $\varphi_A \in \text{FO}^2(\sim, <, +1)$ can be computed such that:

$L(A) \neq \emptyset$ iff $\varphi_A$ has a model

- Decidability of emptiness of vector addition tree automata is an open problem
- It is equivalent to decidability of Multiplicative Exponential Linear Logic
- We concentrate on $\text{FO}^2(\sim, +1)$

### Theorem 8 [Bojańczyk et al. 06b]

Satisfiability of $\text{FO}^2(\sim, +1)$ on data trees is decidable

- The intermediate steps of the proof are similar as for data strings
- But additional techniques needed:
  - Model normalization by cut-and-paste arguments
  - Canonical “small” models that can be recognized by simpler tree automata

**Complexity:**
- Upper bound: $3\text{-NEXPTIME}$
- Lower bound: $\text{NEXPTIME}$
### Theorem 7 [Bojańczyk et al. 06b]

For any **vector addition tree automaton** $A$, a formula $\varphi_A \in \text{FO}^2(\sim, <, +1)$ can be computed such that:

$$L(A) \neq \emptyset \text{ iff } \varphi_A \text{ has a model}$$

- Decidability of emptiness of vector addition tree automata is an open problem
- It is equivalent to decidability of Multiplicative Exponential Linear Logic

\[ \rightarrow \text{ We concentrate on } \text{FO}^2(\sim, +1) \]

### Theorem 8 [Bojańczyk et al. 06b]

Satisfiability of $\text{FO}^2(\sim, +1)$ on data trees is decidable

- The intermediate steps of the proof are similar as for data strings
- But additional techniques needed:
  - Model normalization by cut-and-paste arguments
  - Canonical “small” models that can be recognized by simpler tree automata

**Complexity:**

- Upper bound: $\text{3-NEXPTIME}$
- Lower bound: $\text{NEXPTIME}$

- On trees of bounded depth: $\text{FO}^2$ with all axes decidable \[\text{[Björklund, Bojańczyk 07]}\]
Consequences for XML Reasoning

- **We already know:**
  - Unary key and inclusion constraints can be expressed in \( \text{FO}^2(\sim, +1, <) \) two variables
Consequences for XML Reasoning

- **We already know:**
  - Unary key and inclusion constraints can be expressed in $\mathbf{FO}^2(\sim, +1, <)$ two variables

- **Furthermore:**
  - Regular tree languages can be captured by $\mathbf{EMSO}^2(+1)$
Consequences for XML Reasoning

- **We already know:**
  - Unary key and inclusion constraints can be expressed in $\text{FO}^2(\sim, +1, <)$ two variables

- **Furthermore:**
  - Regular tree languages can be captured by $\text{EMSO}^2(+1)$
  - The core of XPath without data values corresponds exactly to $\text{FO}^2(+1, <)$ [Marx, de Rijke 05]
Consequences for XML Reasoning

- **We already know:**
  - Unary key and inclusion constraints can be expressed in $\text{FO}^2(\sim, +1, <)$ two variables

- **Furthermore:**
  - Regular tree languages can be captured by $\text{EMSO}^2(+1)$
  - The core of XPath without data values corresponds exactly to $\text{FO}^2(+1, <)$ [Marx, de Rijke 05]
  - A simple data-aware fragment of XPath (without transitive axes) can be expressed in $\text{FO}^2(\sim, +1)$
Consequences for XML Reasoning

- We already know:
  - Unary key and inclusion constraints can be expressed in $\text{FO}^2(\sim, +1, <)$ two variables
- Furthermore:
  - Regular tree languages can be captured by $\text{EMSO}^2(+1)$
  - The core of XPath without data values corresponds exactly to $\text{FO}^2(+1, <)$ [Marx, de Rijke 05]
  - A simple data-aware fragment of XPath (without transitive axes) can be expressed in $\text{FO}^2(\sim, +1)$

Query Containment for “simple data-aware XPath” relative to Schemas with integrity constraints is decidable
Consequences for XML Reasoning

- **We already know:**
  - Unary key and inclusion constraints can be expressed in $\text{FO}^2(\sim, +1, <)$ two variables

- **Furthermore:**
  - Regular tree languages can be captured by $\text{EMSO}^2(+1)$
  - The core of XPath without data values corresponds exactly to $\text{FO}^2(+1, <)$ [Marx, de Rijke 05]
  - A simple data-aware fragment of XPath (without transitive axes) can be expressed in $\text{FO}^2(\sim, +1)$

  ➨ **Query Containment for “simple data-aware XPath” relative to Schemas with integrity constraints is decidable**

- More results on reasoning about XML integrity constraints:
  [Arenas et al. 05]
Contents

Introduction
Data Model
Automata
Logic
Two-Variable Logics
▷ Temporal Logics
Other Models
Conclusion
Temporal Logics and the Freeze Quantifier

- $\mathcal{FO}^2$ is natural to consider from an XML point of view
Temporal Logics and the Freeze Quantifier

- $\text{FO}^2$ is natural to consider from an XML point of view.
- From a verification point of view it is natural to add data handling capabilities to temporal logics.
FO\(^2\) is natural to consider from an XML point of view.

From a verification point of view it is natural to add data handling capabilities to temporal logics.

Another natural idea:
- “Use registers in LTL formulas”
  [Demri, Lazić 06]
• $\text{FO}^2$ is natural to consider from an XML point of view

• From a verification point of view it is natural to add data handling capabilities to temporal logics

→ Another natural idea:
  ▶ “Use registers in LTL formulas”
  [Demri, Lazić 06]

• More precisely, add the following two constructs to LTL (or another logic):
  ▶ Unary “quantifiers” $\downarrow_i$
    (where $i$ is a natural number)
  ▶ Atomic formulas $\uparrow_i$
Temporal Logics and the Freeze Quantifier

- **$\mathbf{FO^2}$** is natural to consider from an **XML** point of view
- From a **verification** point of view it is natural to add data handling capabilities to **temporal logics**

→ Another natural idea:
  - “Use registers in LTL formulas”
    - [Demri, Lazic 06]

- More precisely, add the following two constructs to LTL (or another logic):
  - Unary “quantifiers” $\downarrow_i$ (where $i$ is a natural number)
  - Atomic formulas $\uparrow_i$

- **Informal semantics**:
  - $\downarrow_i$ stores the current data value in register $i$
  - $\uparrow_i$ is true if the current data value equals the value in register $i$
Temporal Logics and the Freeze Quantifier

- \( \text{FO}^2 \) is natural to consider from an XML point of view.
- From a verification point of view it is natural to add data handling capabilities to temporal logics.

Another natural idea:
- "Use registers in LTL formulas" [Demri, Lazić 06]

More precisely, add the following two constructs to LTL (or another logic):
- Unary "quantifiers" \( \downarrow_i \) (where \( i \) is a natural number)
- Atomic formulas \( \uparrow_i \)

Informal semantics:
- \( \downarrow_i \) stores the current data value in register \( i \)
- \( \uparrow_i \) is true if the current data value equals the value in register \( i \)

Syntax of LTL with Freeze:
\[
\varphi ::= \top \mid a \mid \uparrow_i \mid \varphi \land \varphi \mid \neg \varphi \mid \neg X \varphi \mid F \varphi \mid G \varphi \mid \varphi U \varphi \mid \downarrow_i \varphi
\]
Temporal Logics and the Freeze Quantifier

- FO² is natural to consider from an XML point of view.
- From a verification point of view it is natural to add data handling capabilities to temporal logics.

Another natural idea:
- “Use registers in LTL formulas”
  - [Demri, Lazić 06]

More precisely, add the following two constructs to LTL (or another logic):
- Unary “quantifiers” ↓ᵢ
  - (where i is a natural number)
- Atomic formulas ↑ᵢ

Informal semantics:
- ↓ᵢ stores the current data value in register i
- ↑ᵢ is true if the current data value equals the value in register i

Syntax of LTL with Freeze:
\[ \varphi ::= T \mid a \mid ↑ᵢ \mid \varphi \land \varphi \mid ¬\varphi \mid X\varphi \mid F\varphi \mid G\varphi \mid \varphi U \varphi \mid ↓ᵢ\varphi \]

Examples:
- (L5) each print request by a process is followed by a print for that user:
  \[ G(r \rightarrow ↓₁XF(↑₁ ∧ s)) \]
Temporal Logics and the Freeze Quantifier

- $\text{FO}^2$ is natural to consider from an XML point of view.
- From a verification point of view it is natural to add data handling capabilities to temporal logics.

Another natural idea:
- “Use registers in LTL formulas”
  - [Demri, Lazić 06]

More precisely, add the following two constructs to LTL (or another logic):
- Unary “quantifiers” $\downarrow^i$ (where $i$ is a natural number)
- Atomic formulas $\uparrow^i$

Informal semantics:
- $\downarrow^i$ stores the current data value in register $i$
- $\uparrow^i$ is true if the current data value equals the value in register $i$

Syntax of LTL with Freeze:
$$\varphi ::= \top | a | \uparrow^i | \varphi \land \varphi | \neg \varphi | X\varphi | F\varphi | G\varphi | \varphi U \varphi | \downarrow^i \varphi$$

Examples:
- (L5) each print request by a process is followed by a print for that user:
  $$G(r \rightarrow \downarrow^1 XF(\uparrow^1 \land s))$$
- (L6) Between two successive print jobs of the same user, some other user’s job has to be processed:
  $$G\neg(r \land \downarrow^1 X(\neg(s \land \neg \uparrow^1) U (s \land \uparrow^1)))$$

A little bit infinite? Thomas Schwentick. /lhd /rhd
Theorem 9 [Demri, Lazić 06]

(a) Finite Satisfiability for LTL with Freeze is
   (1) undecidable in general
   (2) decidable but not primitive recursive if only 1 register is used

(b) Infinite Satisfiability for LTL with Freeze is
   • undecidable even with only 1 register
# LTL with Freeze

## Theorem 9 [Demri, Lazić 06]

<table>
<thead>
<tr>
<th>(a)</th>
<th>Finite Satisfiability for LTL with Freeze is</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1) undecidable in general</td>
</tr>
<tr>
<td></td>
<td>(2) decidable but not primitive recursive if only 1 register is used</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b)</th>
<th>Infinite Satisfiability for LTL with Freeze is</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>undecidable even with only 1 register</td>
</tr>
</tbody>
</table>

### Proof idea

- More than 1 register:
  - Non-Emptiness of Minsky Counter Automata is reducible to Satisfiability of LTL with Freeze
  - Undecidability
LTL with Freeze

Theorem 9 [Demri, Lazić 06]

(a) Finite Satisfiability for LTL with Freeze is
   (1) undecidable in general
   (2) decidable but not primitive recursive if only 1 register is used

(b) Infinite Satisfiability for LTL with Freeze is
   • undecidable even with only 1 register

Proof idea

• More than 1 register:
  ▶ Non-Emptiness of Minsky Counter Automata is reducible to
    Satisfiability of LTL with Freeze
  ➔ Undecidability

• 1 register:
  ▶ Satisfiability for LTL with Freeze with 1 register is basically
    computationally equivalent to Non-Emptiness of Incrementing
    Counter Automata:
    ■ Automata with counters and zero tests,
    ■ but: counters can always be incremented non-deterministically
### Theorem 9 [Demri, Lazić 06]

(a) Finite Satisfiability for LTL with Freeze is
- (1) undecidable in general
- (2) decidable but not primitive recursive if only 1 register is used

(b) Infinite Satisfiability for LTL with Freeze is
- undecidable even with only 1 register

#### Proof idea

- More than 1 register:
  - Non-Emptiness of Minsky Counter Automata is reducible to Satisfiability of LTL with Freeze
  - Undecidability

- 1 register:
  - Satisfiability for LTL with Freeze with 1 register is basically computationally equivalent to **Non-Emptiness of Incrementing Counter Automata**:
    - Automata with counters and zero tests,
    - but: counters can always be incremented non-deterministically
  - Non-Emptiness of Incrementing Counter Automata is
    - decidable but not primitive recursive for finite strings
    - undecidable for finite strings
LTL with Freeze vs. $\text{FO}^2$

- LTL with Freeze cannot express:
  - (L3) for each $a$-position there is a $b$-position with the same data value
LTL with Freeze vs. $\mathsf{FO}^2$

- LTL with Freeze cannot express:
  - (L3) for each $a$-position there is a $b$-position with the same data value
- More generally: it cannot talk about the past
LTL with Freeze vs. $\mathbf{FO}^2$

- LTL with Freeze cannot express:
  - (L3) for each $a$-position there is a $b$-position with the same data value
- More generally: it cannot talk about the past

- $\mathbf{FO}^2$ cannot express:
  - (L6) Between two successive print jobs of the same user some other user’s job has to be printed
LTL with Freeze vs. $\text{FO}^2$

- LTL with Freeze cannot express:
  - (L3) for each $a$-position there is a $b$-position with the same data value
- More generally: it cannot talk about the past

- $\text{FO}^2$ cannot express:
  - (L6) Between two successive print jobs of the same user some other user’s job has to be printed
- More generally: it cannot talk about “betweenness” with respect to data values
LTL with Freeze vs. $\mathbf{FO^2}$

- LTL with Freeze cannot express:
  - (L3) for each $a$-position there is a $b$-position with the same data value
  - More generally: it cannot talk about the past
- $\mathbf{FO^2}$ cannot express:
  - (L6) Between two successive print jobs of the same user some other user’s job has to be printed
  - More generally: it cannot talk about “betweenness” with respect to data values

⇒ LTL with Freeze and $\mathbf{FO^2}$ are incomparable
LTL with Freeze: Extensions and Restrictions

- **LTL with Freeze and past modalities:** [Demri, Lazić 06]
  - $X^{−1}$, $G^{−1}$, $F^{−1}$, $U^{−1}$
  - Can express all $\mathbf{FO}^2$ properties
  - But: Satisfiability undecidable
  - A certain fragment exactly corresponds to $\mathbf{FO}^2$
LTL with Freeze: Extensions and Restrictions

- **LTL with Freeze and past modalities:** [Demri, Lazić 06]
  - $X^{-1}, G^{-1}, \ F^{-1}, \ U^{-1}$
  - Can express all $\text{FO}^2$ properties
  - But: Satisfiability undecidable
  - A certain fragment exactly corresponds to $\text{FO}^2$

- **Safety LTL:** [Lazić 07]
  - **Safety properties:** failure is determined by a finite bad prefix
  - Safety LTL allows $F$ and $U$ only under an odd number of nested negations
LTL with Freeze: Extensions and Restrictions

- **LTL with Freeze and past modalities:** [Demri, Lazić 06]
  - $X^{-1}, G^{-1}, F^{-1}, U^{-1}$
  - Can express all $\mathbf{FO}^2$ properties
  - But: Satisfiability undecidable
  - A certain fragment exactly corresponds to $\mathbf{FO}^2$

- **Safety LTL:** [Lazić 07]
  - Safety properties: failure is determined by a finite bad prefix
  - Safety LTL allows $F$ and $U$ only under an odd number of nested negations
  - Satisfiability for Safety LTL with one register is complete for EXPSPACE
LTL with Freeze: Extensions and Restrictions

- **LTL with Freeze and past modalities:** [Demri, Lazić 06]
  - $X^{-1}$, $G^{-1}$, $F^{-1}$, $U^{-1}$
  - Can express all $\mathcal{FO}^2$ properties
  - But: Satisfiability undecidable
  - A certain fragment exactly corresponds to $\mathcal{FO}^2$

- **Safety LTL:** [Lazić 07]
  - **Safety properties:** failure is determined by a finite bad prefix
  - Safety LTL allows $F$ and $U$ only under an odd number of nested negations
  - Satisfiability for Safety LTL with one register is complete for EXPSPACE

- **Constraint LTL$^\diamond$:** [Demri et al. 07]
  - Future and past modalities
  - More than 1 data value per position: variables
  - Two kinds of data value comparisons:
    - $x = X^k y$: variable $x$ at current position equals variable $y$ at current position $+k$
    - $x = \diamond y$: the current $x$ equals some future $y$
LTL with Freeze: Extensions and Restrictions

- **LTL with Freeze and past modalities:** [Demri, Lazić 06]
  - \(X^{-1}, G^{-1}, F^{-1}, U^{-1}\)
  - Can express all \(FO^2\) properties
  - But: Satisfiability undecidable
  - A certain fragment exactly corresponds to \(FO^2\)

- **Safety LTL:** [Lazić 07]
  - Safety properties: failure is determined by a finite bad prefix
  - Safety LTL allows \(F\) and \(U\) only under an odd number of nested negations
  - Satisfiability for Safety LTL with one register is complete for EXPSPACE

- **Constraint LTL\(\diamond\):** [Demri et al. 07]
  - Future and past modalities
  - More than 1 data value per position: variables
  - Two kinds of data value comparisons:
    - \(x = X^k y\): variable \(x\) at current position equals variable \(y\) at current position + \(k\)
    - \(x = \Diamond y\): the current \(x\) equals some future \(y\)
  - Finitary and Infinitary Satisfiability are decidable
## Automata and Logics

<table>
<thead>
<tr>
<th></th>
<th>RegisterA</th>
<th>PebbleA</th>
<th>ClassMA</th>
<th>$\text{FO}^2$</th>
<th>LTL &amp; Freeze</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressiveness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(L2),(L6),(L7)</td>
<td>(L1)--(L7)</td>
<td>(L1)--(L7)</td>
<td>(L1),(L5),(L7)</td>
<td>(L1),(L2),(L4)--(L7)</td>
</tr>
<tr>
<td><strong>Decidability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-emptiness</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Containment</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data complexity word pr.</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Closure properties</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intersection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complement</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

A little bit infinite?  Thomas Schwentick  tu.
Contents

Introduction
Data Model
Automata
Logic

▷ Other Models
Conclusion
Some Related Work on Data Strings

[Boyer et al. 03] Extension of register automata based on monoids

- Still can only remember a bounded number of data values
- Cannot express (L1), (L3)–(L5)
Some Related Work on Data Strings

[Boyer et al. 03] Extension of register automata based on monoids

- Still can only remember a bounded number of data values
- Cannot express (L1), (L3)–(L5)

[Francez, Kaminski 03] Myhill-Nerode Theorem for data strings
Some Related Work on Data Strings

[Boyer et al. 03] Extension of register automata based on monoids
  • Still can only remember a bounded number of data values
  ➞ Cannot express (L1), (L3)–(L5)

[Francez, Kaminski 03] Myhill-Nerode Theorem for data strings

[Kaminski, Tan 04] Regular expressions
  • ...corresponding to unification-based register automata
Some Related Work on Data Strings

[Boyer et al. 03] Extension of register automata based on monoids
  • Still can only remember a bounded number of data values
  ➞ Cannot express (L1), (L3)–(L5)

[Francez, Kaminski 03] Myhill-Nerode Theorem for data strings

[Kaminski, Tan 04] Regular expressions
  • ...corresponding to unification-based register automata

[Zeitlin 06] Look-ahead register automata
  • ... can guess data values
  ➞ Closed under reversal
  • Equivalent characterizations by
    ▶ Regular expressions (stronger than the above)
    ▶ Grammars
Some Related Work on Data Strings

[Boyer et al. 03] Extension of register automata based on monoids
  - Still can only remember a bounded number of data values
  ➞ Cannot express (L1), (L3)–(L5)

[Francez, Kaminski 03] Myhill-Nerode Theorem for data strings

[Kaminski, Tan 04] Regular expressions
  - …corresponding to unification-based register automata

[Zeitlin 06] Look-ahead register automata
  - … can guess data values
  ➞ Closed under reversal
  - Equivalent characterizations by
    ▶ Regular expressions (stronger than the above)
    ▶ Grammars

[Cheng, Kaminski 98] Register pushdown automata
  - Decidable Non-emptiness
Some Related Work on Data Strings

[Boyer et al. 03] Extension of register automata based on monoids
  • Still can only remember a bounded number of data values
  ➞ Cannot express (L1), (L3)–(L5)

[Francez, Kaminski 03] Myhill-Nerode Theorem for data strings

[Kaminski, Tan 04] Regular expressions
  • ...corresponding to unification-based register automata

[Zeitlin 06] Look-ahead register automata
  • ... can guess data values
  ➞ Closed under reversal
  • Equivalent characterizations by
    ▶ Regular expressions (stronger than the above)
    ▶ Grammars

[Cheng, Kaminski 98] Register pushdown automata
  • Decidable Non-emptiness

LTL on top of first-order logic
  • [Spielmann 00]: Verification of relational transducers
  • [Abdulla et al. 04]: ...even on top of MSO
  • [Deutsch et al. 04]: Verification of web services
  • In all cases: restricted comparison of data values of different states
Some Related Work on Data Trees

[Kaminski, Tan 06] Register automata for trees

[Jurdziński, Lazić 07] Alternation-free modal $\mu$-calculus

- Basically identical results as for LTL with Freeze
- In particular:
  - Computationally equivalent to Incrementing Tree Counter Automata
  - Safety fragment
A little bit infinite? Thomas Schwentick
Conclusion

- **Data strings and data trees constitute a very active research area with (potential) applications in fields like Semistructured Data and Automated Verification**

- **Data strings:**
  - Attracted most attention so far
  - No obvious analogon of regular languages (so far)
  - But "logic → automaton → analysis" possible to some extent
  - Applicability in Verification has yet to be explored:
    - Data string approach is orthogonal to Reachability-based approaches
    - Its ability to talk about data values is limited (e.g., no arithmetic)
    - Is it really useful?
    - ...for other areas? (program analysis, communicating systems,...)

- **Data trees:**
  - Clearly a good model for XML data
  - Can offer a basis for data-aware static analysis
  - Needs more work

- **In both cases we need:**
  - Models with better complexity
  - Models with richer data access
Open Problems

Technical Questions:
- Precise complexity of Satisfiability of $\mathsf{FO}^2(\sim, +1)$ on data strings
- Precise complexity of Satisfiability of $\mathsf{FO}^2(\sim, +1)$ on data trees
- Is Satisfiability of $\mathsf{FO}^2(\sim, <, +1)$ on data trees decidable?
- Upper complexity bounds for Satisfiability of $\mathsf{FO}^2(\sim, <, +1, \pm 1)$ on data strings
- Find a decidable automaton model corresponding to ClassMAs

To be explored:
- Is there a generic class of regular data (string/tree) languages?
- Find models with better complexities
- Study the trade-off between more expressive data access and complexity/decidability
- Find larger decidable fragments of data-aware XPath
Main References (for this Talk)

[Björklund, Schwentick 07] Björklund, Schwentick: On notions of regularity on words with data, FCT 2007

[Bojańczyk et al. 06a] Bojańczyk, Muscholl, Schwentick, Segoufin, David: Two-variable logic on words with data, LICS 2006

[Bojańczyk et al. 06b] Bojańczyk, David, Muscholl, Schwentick, Segoufin: Two-variable logic on data trees and XML reasoning, PODS 2006

[Demri, Lazić 06] Demri, Lazić: LTL wit freeze quantifier and register automata, LICS 2006

[Demri et al. 07] Demri, D'Souza, Gascon: A decidable temporal logic of repeating values


[Lazić 06] Lazić: Safely freezing LTL, FSTTCS 2006

[Neven et al. 01] Neven, Schwentick, Vianu: Finite state machines for strings over infinite alphabets, ACM ToCL 04 (and MFCS 01 with different title)

Surveys:

- Segoufin: Automata and logics for words and trees over an infinite alphabet, CSL 2006
- Segoufin: Static analysis of XML processing with data values, SIGMOD Record 2007
<table>
<thead>
<tr>
<th>Reference</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abdulla et al. 04</td>
<td>Abdulla, Jonsson, Nilsson, d’Orso, Mayank: Regular model checking for LTL(MSO), CAV 2004</td>
</tr>
<tr>
<td>Abdulla et al. 07</td>
<td>Abdulla, Delzanno, Rezine: Parameterized Verification of infinite-state processes with global conditions, CAV 2007</td>
</tr>
<tr>
<td>Alur, Dill 90</td>
<td>Alur, Dill: A theory of timed automata, ICALP 90, TCS 94</td>
</tr>
<tr>
<td>Arenas et al. 05</td>
<td>Arenas, Fan, Libkin: Consistency of XML specifications, Inconsistency Tolerance 2005</td>
</tr>
<tr>
<td>Arenas, Libkin 05</td>
<td>Arenas, Libkin: XML Data Exchange: Consistency and Query Answering, PODS 2005</td>
</tr>
<tr>
<td>Bouajjani et al. 00</td>
<td>Bouajjani, Jonsson, Nilsson, Touili: Regular model checking, CAV 00</td>
</tr>
<tr>
<td>Boyer et al. 03</td>
<td>Bouyer, Petit, Thérien: An algebraic approach to data languages and timed languages, Inf. Comp. 2003</td>
</tr>
<tr>
<td>Cheng, Kaminski 98</td>
<td>Cheng, Kaminski: Context-free languages over infinite alphabets, Acta Inf. 1998</td>
</tr>
<tr>
<td>David 04</td>
<td>David: Mote et données infinis, 2004</td>
</tr>
</tbody>
</table>
[Deutsch et al. 04] Deutsch, Sui, Vianu: Specification and verification of data-driven web applications, PODS 04, JCSS 06

[Francez, Kaminski 03] Francez, Kaminski: An algebraic characterization of deterministic regular languages over infinite alphabets, TCS 2003

[Henzinger 90] Henzinger: Half-order modal logic: how to prove real-time properties, PODS 90


[Marx, de Rijke 05] Marx, de Rijke: Semantic Characterizations of Navigational XPath, SIGMOD record 05


[Zeitlin 06] Zeitlin: Look-ahead finite-memory automata, 2006