Main problems of reactive system synthesis today

Scalability
Oftentimes, practical specifications are too large to handle with current approaches
→ Speedup needed

Usability
Input/Output/Specification of the system is often not Boolean
→ Expressivity improvement needed
Main problems of reactive system synthesis today

**Scalability**
Oftentimes, practical specifications are too large to handle with current approaches
→ Speedup needed

**Usability**
Input/Output/Specification of the system is often not Boolean
→ Expressivity improvement needed

→ conflict!
Main problems of reactive system synthesis today

**Scalability**

Oftentimes, practical specifications are too large to handle with current approaches

→ Speedup or specification simplification needed

**Usability**

Input/Output/Specification of the system is often not Boolean

→ Expressivity improvement needed
Simplifying a specification

How can we do that?

**Main idea:** Get rid of replication in the specification!

Examples specification parts:

- **Arbiter:** Give grants to client $i$ infinitely often
- **Packet sorter:** Forward all packets for customer $i$ to lane $i$
- **Robot waiter:** If menu item $i$ is ordered, then menu item $i$ is delivered

What do we need for that?

Allow symbolic identifiers in the specification.

→ This way, we also dive into synthesis with data
Simplifying a specification

How can we do that?

Main idea: Get rid of replication in the specification!

Examples specification parts:

- **Arbiter**: Give grants to client \(i\) infinitely often
- **Packet sorter**: Forward all packets for customer \(i\) to lane \(i\)
- **Robot waiter**: If menu item \(i\) is ordered, then menu item \(i\) is delivered

What do we need for that?

Allow symbolic identifiers in the specification.

→ This way, we also dive into *synthesis with data*
Main idea

- Allow checking the realizability of many specifications that cannot be implemented in a finite-state fashion
- Synthesize a memory-conservative implementation in case of realizability
Table

Kitchen

$Z_{\text{pickup}} = Z_6$

$Z_0$

$Z_1$

$Z_2$

$Z_3$

$Z_4$

$Z_5$
This is the one-customer variant. For the second customer, add 2 states (“surprise-order”).

There are some additional edges for specifying that a robot is never in two zones at the same time, and that the robot must not deliver to zones without a customer.
## Benchmark results

<table>
<thead>
<tr>
<th>Benchmark:</th>
<th>1-client robot waiter</th>
<th>2-client robot waiter</th>
<th>Mutex</th>
<th>Fault pattern example</th>
</tr>
</thead>
<tbody>
<tr>
<td># States:</td>
<td>17</td>
<td>19</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td># Transitions:</td>
<td>68</td>
<td>72</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Max. pattern size considered:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Computation time considered:</td>
<td>227m</td>
<td>357m</td>
<td>3.5s</td>
<td>0.96s</td>
</tr>
<tr>
<td>Number of positions in abs. game:</td>
<td>216</td>
<td>252</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>
Conclusion

Synthesis with identifiers

- The first approach for synthesis that can compute infinite-state implementations in many practically relevant cases
- A door-opener to the field of synthesis for systems with data: *universal one-weak automata & patterns*