Synthesis from Waveform Transition Graphs

Alberto Moreno\textsuperscript{1}, Danil Sokolov\textsuperscript{1}, Jordi Cortadella\textsuperscript{2}

\textsuperscript{1}School of Engineering, Newcastle University, UK
\textsuperscript{2}Department of Computer Science, Universitat Politècnica de Catalunya, Spain

ASYNC 2019
Outline

- Motivation for yet another model
- Requirements from circuit designers
- Intuition for Waveform Transition Graphs
- Conversion to Signal Transition Graphs for synthesis and verification
- Design automation in Workcraft
- Examples and evaluation
Motivation: Application domain

- "Little digital" control – an ideal case for asynchronous design [1]
  - Relatively small controllers
  - Prompt reaction is paramount
  - Interface analog world

- Modelling aspects
  - Fine-grain control at the level of individual signals
  - Graph-based representation for causality, concurrency, and conflicts

[1] D. Sokolov et al. “Automating the design of async. logic control for AMS electronics” IEEE TCAD, 2019
Motivation: Limitations of existing models

- **Signal Transition Graphs (STGs)**
  - 😊 Great expressive power and tool support
  - 😞 Underlying Petri nets are unfamiliar to engineers
  - 😞 Sophisticated modelling aspects (output persistency, input properness, non-commutativity, UCS/CSC conflicts, etc.)

- **Burst Mode (BM) and eXtended BM (XBM) automata**
  - 😊 Engineers understand the underlying state machines
  - 😞 Insufficient expressive power due to limited concurrency

- **Generalized / Extended / Symbolic STGs**
  - 😞 Even more complex than STGs
  - 😞 No mature tool support
Specification flow (industry perspective)

1. Sketch a waveform for intended circuit behaviour
2. Manually convert the waveform (or its fragment for one mode) to STG
3. Make sure that simulation of the STG resembles the sketch waveform
4. Repeat steps 2-3 for every distinctive mode of operation
5. Combine STGs for all modes in a state machine-like structure
6. Try hard to resolve all the STG implementability issues (inconsistency, irreducible encoding conflicts, non-persistency, etc.)

- How to express destabilisation/stabilisation of input signals?
- How to select the mode of operation based on signal levels?
- Can this flow be simplified and automated?
Usability requirements for a new model

- State machine to express high-level modes of operation
  - Choice is restricted to state machine level
  - Current state is represented by a single token

- Waveforms to capture partial order of signals in each mode
  - Concurrency is contained within waveforms
  - At most one waveform is active at a time

- Advanced features for input signals
  - Unstable (don’t care) and undefined (stable but unknown) states

- Flexibility in modelling of choice
  - Edge-sensitive and level-sensitive
Intuition for Waveform Transition Graphs

- **Burst Mode automaton**: state machine + input/output bursts

  - **WTG**: state machine whose arcs are waveforms

  - **Enabled waveform activation**:
    - Consume a token from the entry state
    - Execute all its events
    - Produce token at the exit state
Intuition for Waveform Transition Graphs

- Burst Mode automaton: state machine + input/output bursts

- WTG: state machine whose arcs are waveforms
Advanced features for signals

- Unstable inputs via destabilise/stabilise events
- Stabilise to low, high or unknown state

<table>
<thead>
<tr>
<th>from state</th>
<th>low</th>
<th>high</th>
<th>unstable</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>![low-to-low]</td>
<td>![low-to-high]</td>
<td>![low-to-unstable]</td>
<td>![low-to-stable]</td>
</tr>
<tr>
<td>high</td>
<td>![high-to-low]</td>
<td>![high-to-high]</td>
<td>![high-to-unstable]</td>
<td>![high-to-stable]</td>
</tr>
<tr>
<td>unstable</td>
<td>![unstable-to-low]</td>
<td>![unstable-to-high]</td>
<td>![unstable-to-unstable]</td>
<td>![unstable-to-stable]</td>
</tr>
</tbody>
</table>

Legend:
- conventional rise/fall events
- destabilise events
- stabilise events
Flexibility in modelling of choice

- **Edge-sensitive choice**

- **Level-sensitive choice**
D flip-flop example

- High-level state machine

- Possible trace waveform
WTG to STG conversion: Simple waveform

- **WTG fragment**

- **STG fragment – one-to-one mapping**
WTG to STG conversion: Simple waveform

- WTG fragment

- STG fragment – redundant arcs removed
WTG to STG conversion: Simple waveform

- WTG fragment

- STG fragment – rearranged layout
WTG to STG conversion: Stabilise at HIGH/LOW state

- WTG fragment

- STG fragment
WTG to STG conversion: Stabilise at unknown state

- WTG fragment

- STG fragment
WTG to STG conversion: Guards in level-sensitive choice

- **WTG**

- **STG**
Design automation in WORKCRAFT

- Support for capturing and simulating WTGs
- Local structural checks to ensure implementability
  - Consistency of signals between waveforms
  - Output-persistency and output-determinancy at choice states
  - See the paper for more details
- Automatic conversion to STGs as backend representation
- Reuse existing methods and tools
  - Formal verification of specification (Punf + MPSat)
  - Logic synthesis of circuit implementation (Petrify, MPSat, ATACS)
- Backtracking for communication of problems

Output-persistency: enabled output must not be disabled by another signal
Output-determinacy: if an output is enabled by a sequence of events then all executions of this trace must enable the same output
Design automation in WORKCRAFT
Instruction decoder example: Block diagram

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Opcode</th>
<th>Op0,Op1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td></td>
<td>0,0</td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td>0,1</td>
</tr>
<tr>
<td>Load</td>
<td></td>
<td>1,0</td>
</tr>
<tr>
<td>Store</td>
<td></td>
<td>1,1</td>
</tr>
</tbody>
</table>
Instruction decoder example: High-level state machine
Instruction decoder example: High-level state machine

The diagram shows a state machine with various states and transitions. The states are labeled as follows:

- **s0**: initialise
- **s1**: branch
- **s2**: arithmetic
- **s3**: complete
- **z**: no jmp
- **!z**: jmp

Transitions are labeled with conditions like `op0, !op1`, and actions like `load`, `store`, etc. The diagram illustrates the flow through the state machine based on the input conditions.
Instruction decoder example: High-level state machine
Instruction decoder example: Complete WTG
Instruction decoder example: STG and SI circuit
# Productivity: WTG vs STG

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size</th>
<th>User</th>
<th>Design time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mode</td>
<td>signal</td>
<td>STG</td>
</tr>
<tr>
<td>C-element</td>
<td>1</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>VME bus controller</td>
<td>2</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>Buck controller</td>
<td>3</td>
<td>7</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*WTG: Wire Trace Generation, STG: Spice Trace Generation, Impr: Improvement*
Productivity: WTG vs STG

Average data for 3 users with different experience: >25% productivity improvement

- Keyboard & mouse actions
- Design time (s)

Benchmark size (modes; signals)
Conclusions

- **WTGs model**
  - Based on familiar modelling abstractions
  - Explicit separation of choice and concurrency aspects
  - Simpler than STGs and more expressive than XBM automata
  - Support for unstable signals via destabilise/stabilise events
  - Edge-sensitive and level-sensitive choice

- **WTGs design automation**
  - Design flow supported in Workcraft (https://workcraft.org/)
  - 25% productivity improvement compared to STGs
  - STG translation for reuse of synthesis and verification tools