Formal models of timed systems: WCET analysis in single-core systems, and some ideas for multi-core systems

Jean-Luc Béchennec  Sébastien Faucou
CAPITAL Workshop - 4th of June 2021

Université de Nantes, CNRS, LS2N
F-44000 Nantes, France
This talk is about a work carried out in our group since a few years concerning the use of real-time model-checking to estimate the WCET of programs

- The work was initiated by Franck Cassez (now with ConsenSys Software R&D)
- Quickly joined by Jean-Luc Béchennec
- And a bit later by Mikäel Briday, Sébastien Faucou and Armel Mangean
Talk is split in 2 parts

- Review of past work concerning single-core systems
  → Sébastien Faucou

- Demo of on-going works concerning multi-core systems
  → Jean-Luc Béchennec
Real-time model-checking for WCET analysis: motivations and overview
Given a system $S$ composed of:

**A program $\mathcal{P}$**

```
00003000 <_start>:
  3000: li r1,1 ;r1 <- 1
  3004: ori r1,r1,49296 ;r1 <- r1 | 49296
  3008: bl 3010 ;call main
0000300c <loop>:
  300c: b 300c ;branch
00003010 <main>:
  3010: li r8,29 ;r8 <- 29
  3014: li r10,1 ;r10 <- 1
  3018: mtctr r8 ;ctr <- r8
  301c: li r9,1 ;r9 <- 1
  3020: b 3028 ;branch
  3024: mr r9,r3 ;r9 <- r3
  3028: add r3,r9,r10 ;r3 <- r9+r10
  302c: mr r10,r9 ;r10 <- r9
  3030: bdnz 3024 ;ctr--, ;branch if ctr!=0
  3034: blr ;return
```

**A micro-architecture $\mathcal{A}$**

Find an upper-bound on the execution time of $\mathcal{P}$ on $\mathcal{A}$
The WCET problem (cont’d)

The WCET bound does not necessarily correspond to a run of $S$: any value greater than or equal to the actual WCET is valid.
To derive a WCET bound, one needs to combine:

- **Program analysis**
  - which instructions are executed? how many times?

- **Architecture analysis**
  - how long does it take to *execute* each instruction?
Real-time model-checking = automated verification of timed models.

Timed models: discrete event formalism extended with real-valued clocks, e.g., timed automata, or time Petri nets.

Ex: monitoring of a sporadic task with a minimal inter-arrival time
Real-time model checking (cont’d)

Real-time model checkers:

- Powerful abstractions to represent and manipulate the dense-time part of the state (e.g., DBM, zone)
- But have to rely on an explicit representation of the discrete part (no BDD/ZDD, no efficient partial order)

We have experimented with 2 tools:

- **UPPAAL**¹ based on timed automata
- **Roméo**² based on time Petri nets

Both offer:

- modular models with synchronization between processes
- finite variables to model the discrete part of the state
- a C-like language to manipulate the discrete part of the state

¹ [https://uppaal.org/](https://uppaal.org/)
² [http://romeo.rts-software.org](http://romeo.rts-software.org)
Pipeline stages, cache, memory controllers, buses are concurrent components that evolve and synchronize in real-time.

WCET analysis asks to analyse their timing behavior.

At first sight, real-time model-checking is precisely done for this type of job. It seems an interesting direction to explore for WCET analysis.
With real-time model-checking, the analysis is **based on the exploration of traces**

- When an instruction is *executed*, its actual execution time is defined by the current state

→ Thanks to context-sensitive execution times, we expect to obtain accurate bounds

- In case of missing/unknown information, the trace is split to account for the different cases.

  initial cache content, input data, contention latency, ...

→ We cannot expect to support too much missing/unknown information
Open questions that we wanted to explore

• It will certainly face **scalability issues** → is it even usable?

• Since the **analysis is based on traces**, it should be ”closer” to the real system → how accurate is it ?

• **Exhaustivity** ensures correctness in the presence of so-called timing anomalies (close to non sustainability in scheduling) → is it a golden bullet?
Overview of our approach

Given $S = P \times A$

1. Compute an abstract model $\hat{P}$ of $P$ (fully automated)

2. Build an abstract model $\hat{A}$ of $A$ (not automated but needs to be done only one time)

3. Compute an abstract model $\hat{S} = \hat{P} \times \hat{A}$ of $S$ (fully automated)

4. Search for the WCET of $\hat{S}$ with a model-checker (fully automated)
Modeling a program
Basic intuition of program models

A program is a sequence of instructions + a set of memory locations

• We are only interested in **binary programs**.
• An intuitive representation is an automata/Petri net such that:
  • each instruction is associated with a location/place
  • a control flow between two instructions is denoted by an edge/transition
• This is an **untimed model**: a program is inactive
• This intuitive representation proved to be relevant for visualization and debugging
• Memory locations are represented by variables
  **warning**: explicit representation in the state
Objectives of program abstraction

Execution of an instruction is split in two parts:

- interaction with the micro-architecture, e.g.,
  - impact on the state of caches
  - traversal of pipeline
  - or memory accesses

- semantics: updates the memory locations

Observation

For WCET analysis, an update to a memory location can be discarded if it does not impact the timing behavior.

Corrolary: the content of a memory location does not need to be tracked if all its updates can be discarded.
Example

00003000 <_start>:
  3000:  li r1,1 ;r1 <- 1
  3004:  ori r1,r1,49296 ;ri <- r1 | 49296
  3008:  bl 3010 ;call main

0000300c <loop>:
  300c:  b 300c ;branch

00003010 <main>:
  3010:  li r8,29 ;r8 <- 29
  3014:  li r10,1 ;r10 <- 1
  3018:  mtctr r8 ;ctr <- r8
  301c:  li r9,1 ;r9 <- 1
  3020:  b 3028 ;branch
  3024:  mr r9,r3 ;r9 <- r3
  3028:  add r3,r9,r10 ;r3 <- r9+r10
  302c:  mr r10,r9 ;r10 <- r9
  3030:  bdnz 3024 ;ctr--,
       ;branch if ctr!=0
  3034:  blr ;return

Which registers do we need to track to compute the value of \texttt{ctr} at instruction \texttt{3030}?
Which registers do we need to track to compute the value of \texttt{ctr} at instruction \texttt{3030}?
Program slicing = techniques to compute a subprogram which is equivalent to a program wrt. a set of variables and a set of locations. For WCET analysis:

1. Find a subprogram that reaches the end node with the same control flow.
2. Build a model that tracks the content of a memory location iff it appears in this subprogram.

Interactions with the micro-architecture (incl. memory accesses) are not modified.

---

true fetch! \( \text{pipeline}(\_3000, \text{true}), \text{execute}(\_3000) \)

true fetch! \( \text{pipeline}(\_3004, \text{true}), \text{execute}(\_3004) \)

true fetch! \( \text{pipeline}(\_3008, \text{true}), \text{execute}(\_3008) \)

true fetch! \( \text{pipeline}(\_300c, \text{true}), \text{execute}(\_300c) \)

true fetch! \( \text{pipeline}(\_3010, \text{true}), \text{execute}(\_3010) \)

true fetch! \( \text{pipeline}(\_3014, \text{true}), \text{execute}(\_3014) \)

true fetch! \( \text{pipeline}(\_3018, \text{true}), \text{execute}(\_3018) \)

true fetch! \( \text{pipeline}(\_301c, \text{true}), \text{execute}(\_301c) \)

true fetch! \( \text{pipeline}(\_3020, \text{true}), \text{execute}(\_3020) \)

true fetch! \( \text{pipeline}(\_3024, \text{true}), \text{execute}(\_3024) \)

true fetch! \( \text{pipeline}(\_3028, \text{true}), \text{execute}(\_3028) \)

true fetch! \( \text{pipeline}(\_302c, \text{true}), \text{execute}(\_302c) \)

true fetch! \( \text{pipeline}(\_3030, \text{true}), \text{execute}(\_3030) \)

true fetch! \( \text{pipeline}(\_3034, \text{true}), \text{execute}(\_3034) \)
BEST: a Binary Executable Slicing Tool\textsuperscript{4}

\textsuperscript{4}Mangean et al., BEST: a Binary Executable Slicing Tool. In Int. Work. on Worst-Case Execution Time Analysis, 2016.
Efficiency of slicing for WCET analysis (cont’d)

Binary programs compiled from Mälardalen benchmarks to PowerPC with GCC and COSMIC C, sliced with BEST.

Evaluation of **registers whose contents do not need to be tracked**

<table>
<thead>
<tr>
<th>Source file</th>
<th>GCC</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>COSMIC C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-00</td>
<td>-01</td>
<td>-02</td>
<td>-03</td>
<td>-no</td>
<td>default</td>
</tr>
<tr>
<td>adpcm.c</td>
<td>11/17, 35%</td>
<td>28/32, 13%</td>
<td>26/28, 7%</td>
<td>33/36, 8%</td>
<td>22/37, 41%</td>
<td>22/37, 41%</td>
</tr>
<tr>
<td>bs.c</td>
<td>7/11, 36%</td>
<td>10/13, 23%</td>
<td>9/10, 10%</td>
<td>9/10, 10%</td>
<td>10/14, 29%</td>
<td>11/13, 15%</td>
</tr>
<tr>
<td>bsort100.c</td>
<td>9/12, 25%</td>
<td>13/18, 28%</td>
<td>11/16, 31%</td>
<td>11/16, 31%</td>
<td>13/15, 13%</td>
<td>13/15, 13%</td>
</tr>
<tr>
<td>cnt.c</td>
<td>10/15, 33%</td>
<td>13/18, 28%</td>
<td>10/16, 38%</td>
<td>10/18, 44%</td>
<td>10/37, 73%</td>
<td>10/37, 73%</td>
</tr>
<tr>
<td>compress.c</td>
<td>15/19, 21%</td>
<td>26/31, 16%</td>
<td>30/33, 9%</td>
<td>32/35, 9%</td>
<td>21/37, 43%</td>
<td>21/37, 43%</td>
</tr>
<tr>
<td>crc.c</td>
<td>8/17, 53%</td>
<td>14/23, 39%</td>
<td>10/19, 47%</td>
<td>9/19, 53%</td>
<td>18/37, 51%</td>
<td>18/37, 51%</td>
</tr>
<tr>
<td>expint.c</td>
<td>8/13, 38%</td>
<td>16/26, 38%</td>
<td>4/11, 66%</td>
<td>4/11, 63%</td>
<td>14/37, 62%</td>
<td>14/37, 62%</td>
</tr>
<tr>
<td>fdct.c</td>
<td>6/13, 54%</td>
<td>4/21, 81%</td>
<td>4/30, 87%</td>
<td>3/33, 91%</td>
<td>3/35, 91%</td>
<td>3/35, 91%</td>
</tr>
<tr>
<td>fibcall.c</td>
<td>7/11, 36%</td>
<td>7/12, 42%</td>
<td>3/7, 57%</td>
<td>3/7, 57%</td>
<td>6/12, 50%</td>
<td>6/10, 40%</td>
</tr>
<tr>
<td>fir.c</td>
<td>7/16, 56%</td>
<td>13/22, 41%</td>
<td>14/21, 33%</td>
<td>14/21, 33%</td>
<td>15/37, 59%</td>
<td>15/37, 59%</td>
</tr>
<tr>
<td>janne_complex.c</td>
<td>7/12, 42%</td>
<td>6/9, 33%</td>
<td>6/8, 25%</td>
<td>7/9, 22%</td>
<td>7/36, 81%</td>
<td>7/8, 13%</td>
</tr>
<tr>
<td>jfdctint.c</td>
<td>8/11, 27%</td>
<td>3/15, 80%</td>
<td>4/25, 84%</td>
<td>4/33, 88%</td>
<td>3/35, 91%</td>
<td>3/34, 91%</td>
</tr>
<tr>
<td>matmult.c</td>
<td>10/19, 47%</td>
<td>15/20, 25%</td>
<td>15/19, 21%</td>
<td>13/19, 32%</td>
<td>8/37, 78%</td>
<td>8/37, 78%</td>
</tr>
<tr>
<td>ndes.c</td>
<td>9/17, 47%</td>
<td>21/27, 22%</td>
<td>23/26, 12%</td>
<td>27/28, 4%</td>
<td>16/37, 57%</td>
<td>15/37, 59%</td>
</tr>
<tr>
<td>ns.c</td>
<td>9/14, 36%</td>
<td>13/17, 24%</td>
<td>13/15, 13%</td>
<td>9/12, 25%</td>
<td>14/37, 62%</td>
<td>14/36, 61%</td>
</tr>
<tr>
<td>prime.c</td>
<td>10/13, 23%</td>
<td>6/9, 33%</td>
<td>6/9, 33%</td>
<td>6/8, 25%</td>
<td>11/36, 69%</td>
<td>12/36, 67%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>38%</td>
<td>35%</td>
<td>36%</td>
<td>37%</td>
<td>59%</td>
<td>54%</td>
</tr>
</tbody>
</table>

# of registers in the slice / total # of register used in program, gain in percentage (the higher the better).
Modeling the micro-architecture
The model of the architecture:

• Should allow cycle accurate execution of instruction
  \[\Rightarrow\] it is a timed model

• But should not mimic the actual design of the micro-architecture

• In particular, semantics of an instruction should be executed independently from the interaction with the micro-architecture

Any transformation that preserves the behavior and decrease either the size of the discrete state or the number of state is welcome!
After several trials, we have adopted the following modeling style:

- the C-like language is used to define and manipulate the discrete part of the state
- the TA/TPN part is used for handling clocks and synchronizations

Our early models integrated too much timing and functional aspects. A clear separation offers more possibilities of abstraction, e.g.,

- actual content of the cache is not needed (only the tag & valid bit)
- ALU does no computation, it is just a stage to add a delay according to the class of the instruction
- speculation and rollbacks can be pre-computed offline and integrated in the model of the program
- ...
Example: UPPAAL model of memory hierarchy

Part of the model inspired from PowerPC e200z4 micro-architecture

- Instruction cache only
- On cache-hit, instruction is sent to the pipeline in 1 cycle
- On cache-miss, a burst access is required (8 4-bytes words)
- Burst is received in a FillBuffer
- Burst starts with the requested word
Example: UPPAAL model of cache update (pseudo-RR)

```c
void IMU_ICache_Update() {
    // on a miss, insert the current instruction on the instruction cache
    int addr = _INSTS[IMU.FillBuffer.index].addr;
    int[0, IMU_WAYS_MAX] way;
    int[0, IMU_SETS_MAX -1] set = (addr / 32) % IMU_SETS_MAX;
    int tag = addr / (32 * IMU_SETS_MAX);

    bool found = false;

    way = 0;
    while (!found && way < IMU_WAYS_MAX)
        if (IMU.ICache.tags[way][set] == -1)
            found = true;
        else ++way;
    if (found) {
        // free slot found
        IMU.ICache.tags[way][set] = tag;
    } else {
        // no free slot found (pseudo round-robin replacement policy)
        way = IMU.ICache.rp_way;
        IMU.ICache.tags[way][set] = tag;
        IMU.ICache.rp_way = (IMU.ICache.rp_way +1) % IMU_WAYS_MAX;
    }
}
```
Validation of the hardware model is the most difficult part

- Documentation of micro-architecture generally lacks precision
- Micro-benchmarks can be used to fill in the gaps in the documentation but it is a tedious job with no completeness guarantee
- Moreover, complexity of models is high enough to raise concerns
  - C-like language is not a 1st class citizen in UPPAAL/Roméo
- Model-checking can be used to verify some properties but we can only find the bugs that we are looking for.
WCET analysis
Both UPPAAL and Roméo provide dedicated algorithms to search for the maximum value of a clock

- UPPAAL: \texttt{sup: _clock}
- Roméo\textsuperscript{5}: \texttt{maxcost(Program::END_INST == 1)}

The result is composed of:

- a value: the WCET bound
- a trace: a run of \( \hat{P} \) on \( \hat{A} \) that yields the WCET bound
- \( \hat{A} \) can be instrumented to also embed performance counters (e.g., cache hit/miss or BTB hit/miss)

If the control flow of the program is independent from unknown input-data, the trace can be replayed on the real system.

\textsuperscript{5}In practice, we need to search the mincost of a negative value
We did once the effort to validate closely hardware models

- Model of an ARM9TDMI core by micro-benchmarking of the system
  - some hidden architectural features have been discovered using μBenchmarking but some were probably not
- Ad-hoc program slicing
- WCET analysis with UPPAAL
- Then measurements were done on the real hardware
  - for programs with input-dependant control flow, worst-case inputs were used but the measure must be considered as a lower bound of the actual WCET

---

# About the tightness of the estimation: data

<table>
<thead>
<tr>
<th>Program</th>
<th>Analysis time</th>
<th># States</th>
<th>Analysed WCET</th>
<th>Measured WCET</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single-path programs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fib-O0</td>
<td>2s</td>
<td>74,181</td>
<td>8,098</td>
<td>8,064</td>
<td>0.42%</td>
</tr>
<tr>
<td>fib-O1</td>
<td>0.6s</td>
<td>22,333</td>
<td>2,597</td>
<td>2,544</td>
<td>2.0%</td>
</tr>
<tr>
<td>fib-O2</td>
<td><strong>0.3s</strong></td>
<td>9,711</td>
<td>1,209</td>
<td>1,164</td>
<td>3.8%</td>
</tr>
<tr>
<td>jane-complex-O0</td>
<td>1.7s</td>
<td>38,038</td>
<td>4,264</td>
<td>4,164</td>
<td>2.4%</td>
</tr>
<tr>
<td>jane-complex-O1</td>
<td>0.5s</td>
<td>14,600</td>
<td>1,715</td>
<td>1,680</td>
<td>2.0%</td>
</tr>
<tr>
<td>jane-complex-O2</td>
<td>0.5s</td>
<td>13,004</td>
<td>1,557</td>
<td>1,536</td>
<td>1.3%</td>
</tr>
<tr>
<td>fdct-O1</td>
<td>21s</td>
<td>60,534</td>
<td>4,245</td>
<td>4,092</td>
<td>3.7%</td>
</tr>
<tr>
<td>fdct-O2</td>
<td>3.2s</td>
<td>55,285</td>
<td>19,231</td>
<td>18,984</td>
<td>1.3%</td>
</tr>
<tr>
<td><strong>Single-path programs w/ data dependant instr. durations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdct-O0</td>
<td>124s</td>
<td>85,008</td>
<td>11,800</td>
<td>11,448</td>
<td>3.0%</td>
</tr>
<tr>
<td>matmult-O0</td>
<td>217s</td>
<td>10,531,262</td>
<td>529,250</td>
<td>528,684</td>
<td>0.1%</td>
</tr>
<tr>
<td>matmult-O1</td>
<td>25s</td>
<td>1,112,527</td>
<td>156,367</td>
<td>153000</td>
<td>2.2%</td>
</tr>
<tr>
<td>matmult-O2</td>
<td>121s</td>
<td>6,780,931</td>
<td>148,299</td>
<td>140,664</td>
<td>5.4%</td>
</tr>
<tr>
<td>jfdcint-O0</td>
<td>92s</td>
<td>100,861</td>
<td>12,918</td>
<td>12,588</td>
<td>2.6%</td>
</tr>
<tr>
<td>jfdcint-O1</td>
<td>12s</td>
<td>35,419</td>
<td>5,072</td>
<td>4,688</td>
<td><strong>8.6%</strong></td>
</tr>
<tr>
<td>jfdcint-O2</td>
<td>5.38s</td>
<td>175,661</td>
<td>16,938</td>
<td>16,380</td>
<td>3.4%</td>
</tr>
<tr>
<td><strong>Multi-path programs (input data dependant control flow)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bs-O0</td>
<td>30s</td>
<td>1,421,274</td>
<td>1068</td>
<td>1,056</td>
<td>1.1%</td>
</tr>
<tr>
<td>bs-O1</td>
<td>23s</td>
<td>1,214,673</td>
<td>738</td>
<td>720</td>
<td>2.5%</td>
</tr>
<tr>
<td>bs-O2</td>
<td>12s</td>
<td>655,870</td>
<td>628</td>
<td>600</td>
<td>4.6%</td>
</tr>
<tr>
<td>cnt-O0</td>
<td>4s</td>
<td>77,002</td>
<td>9,027</td>
<td>8,836</td>
<td>2.1%</td>
</tr>
<tr>
<td>cnt-O1</td>
<td>1.4s</td>
<td>27,146</td>
<td>4,123</td>
<td>3,996</td>
<td>3.1%</td>
</tr>
<tr>
<td>cnt-O2</td>
<td>9s</td>
<td>11,490</td>
<td>3,067</td>
<td>2928</td>
<td>4.6%</td>
</tr>
<tr>
<td>insertsort-O0</td>
<td><strong>598.98s</strong></td>
<td>24,250,738</td>
<td>3133</td>
<td>3108</td>
<td>0.8%</td>
</tr>
<tr>
<td>insertsort-O1</td>
<td>353.80s</td>
<td>11,455,293</td>
<td>1533</td>
<td>1500</td>
<td>2.2%</td>
</tr>
<tr>
<td>insertsort-O2</td>
<td>11.68s</td>
<td>387,292</td>
<td>1326</td>
<td>1320</td>
<td>0.4%</td>
</tr>
<tr>
<td>ns-O0</td>
<td>60s</td>
<td>3,064,316</td>
<td>30,968</td>
<td>30,732</td>
<td>0.8%</td>
</tr>
<tr>
<td>ns-O1</td>
<td>8s</td>
<td>368,720</td>
<td>11,701</td>
<td>11,568</td>
<td>1.1%</td>
</tr>
<tr>
<td>ns-O2</td>
<td>55s</td>
<td>1,030,746</td>
<td>7280</td>
<td>7236</td>
<td>0.6%</td>
</tr>
</tbody>
</table>
WCET analysis of multicore systems (WIP)
Using the Roméo tool (Time Petri Net)

- Simple architecture with 2-stages pipeline (Fetch + Execute)
- Handle the Load/Store multiple words of ARM Cortex
- Small 512 bytes direct mapped instruction cache
- No data cache
Program modeling (1)

Cortex M0+ binary code of bsort compiled with -02 optimization (no slicing)

```assembly
000804c <BubbleSort>:
  804c: b570 push {r4, r5, r6, lr}
  8050: 1d06 adds r6, r0, #4
  8052: 0033 movs r3, r6
  8054: 2201 movs r2, #1
  8056: 2501 movs r5, #0
  8058: e00a b . n 8070 <BubbleSort+0x24>
  805a: 6819 ldr r1, [r3, #0]
  805c: 4281 cmp r5, #0
  805e: 2d00 cmp r5, #0
  8060: dd02 ble . n 8068 <BubbleSort+0x1c>
  8062: 2500 movs r5, #0
  8064: 6018 str r0, [r3, #0]
  8066: 6059 str r1, [r3, #4]
  8068: 3201 adds r2, #1
  806a: 3304 adds r3, #4
  806c: 2a64 cmp r2, #100 ; 0x64
  806e: d001 beq . n 8074 <BubbleSort+0x28>
  8070: 4294 cmp r4, r2
  8072: daf2 bge . n 805a <BubbleSort+0x0e>
  8074: 2d00 cmp r5, #0
  8076: d102 bne . n 807e <BubbleSort+0x32>
  8078: 3c01 subs r4, #1
  807a: 2c00 cmp r4, #0
  807c: d1e9 bne . n 8052 <BubbleSort+0x6>
  807e: bd70 pop {r4, r5, r6, pc}
```
int inst8078(core_t &core, mem_t &mem) { // 8078: subs r4, #1
    updateSR(core.regs, core.regs.r[4]);
    return cacheAccess(core.ICache, 32888);
}

int inst807a(core_t &core, mem_t &mem) { // 807a: cmp r4, #0
    uint32_t val = core.regs.r[4] - 0;
    updateSR(core.regs, val);
    return cacheAccess(core.ICache, 32890);
}

int inst807c(core_t &core, mem_t &mem) { // 807c: bne.n 8052
    return cacheAccess(core.ICache, 32892);
}

int inst807e(core_t &core, mem_t &mem) { // 807e: pop {r4, r5, r6, pc}
    core.regs.r[15] = memRead(mem, core.regs.r[13] + 0);
    core.regs.r[6] = memRead(mem, core.regs.r[13] + 4);
    core.regs.r[5] = memRead(mem, core.regs.r[13] + 8);
    core.regs.r[4] = memRead(mem, core.regs.r[13] + 12);
    return cacheAccess(core.ICache, 32894);
}
Phased opponent modeling

```
prologAccessCount > 0
prologAccessCount = prologAccessCount - 1;

lockBus == 0
lockBus = 1;
T16 [ 10; 10 ]
lockBus = 0;
T17 [ 0; 0 ]
T18 [ 500; 1000 ]
prologAccessCount == 0

lockBus == 0
lockBus = 1;
T16 [ 10; 10 ]
lockBus = 0;
T17 [ 0; 0 ]
T18 [ 500; 1000 ]
prologAccessCount == 0

T4 [ 1; 1 ]
accessCount > 0
accessCount = accessCount - 1;
T5 [ 0; 0 ]
T6 [ 0; 0 ]
prologAccessCount > 0
prologAccessCount = prologAccessCount - 1;
T7 [ 10; 10 ]
lockBus = 0;
T8 [ 1; 1 ]
lockBus = 0;
T9 [ 1; 1 ]
lockBus = 0;
T10 [ 0; 0 ]
```
Conclusions and future work
Conclusions

• Of course there are scalability problems. Naive modeling generally leads to an explosion of the state space.
• The more uncertainties there are about the behavior of a program, the greater the number of traces to explore (Timed model checkers use symbolic states for time but not for discrete variables).
• Aggressive yet accurate abstraction techniques can enable effective use of these methods.

As is often the case, going from a research prototype to industrial-grade tool will require a huge engineering effort.
Future work

• Detection of temporal anomalies using observers added to the model

• Aggressive abstractions for multi-core:
  • slicing to reduce programs to memory accesses
  • taking into account the phases in the behavior of the programs

• Ad-hoc model checking techniques taking into account the specificities of the problem
Thanks for your attention!