The Quasi-Synchronous Approach to Distributed Control Systems

Paul Caspi  Verimag Laboratory

caspi@imag.fr  http://www-verimag.imag.fr

Crisys Esprit Project

http://borneo.gmd.de/~ap/crisys/
The Quasi-Synchronous Approach to Distributed Control Systems

Paul Caspi
Verimag Laboratory

caspi@imag.fr http://www-verimag.imag.fr

Crisys Esprit Project
http://borneo.gmd.de/~ap/crisys/

- Where does it come from?
- How to simulate it?
- How to understand it?
- Fault-tolerance
Where does it come from?

From analog boards to computers

- Analog Board
- Clock
- A/D
- Computer
- D/A
- periodic clocks
- synchronous programs
Synchronous Programming

General

initialize state;

loop each input event

read other inputs;
compute outputs and state;
emit outputs

end loop

Several styles (imperative, data-flow,...)

Allow multiple simultaneous event : no performance problems
Synchronous Programming

Periodic

initialize state;

loop each clock

   read other inputs;
   compute outputs and state;
   emit outputs

end loop

Synchronous Programming

Periodic

initialize state;

loop each clock

read other inputs;
compute outputs and state;
emit outputs

end loop

most applications of synchronous programming are actually periodic ones.

hybridity: sampling differential equations require periodicity!
Where does it come from?

From networks of analog boards to local area networks

- Analog Board
- Analog Board
- Clock
- A/D
- Computer
- Serial
- D/A
- Clock
- A/D
- Computer
- Serial
- Computer
- D/A
- Clock
- A/D
- Computer
- D/A
- Bus

- independent periodic clocks
- synchronous programs
Interest

Autonomy, robustness

- Each computer is a complete one, including its own clock and even possibly its own power supply.

- Communication between computers is non-blocking, based on periodic reads and writes, akin to periodic sampling.
How to formalize it

Net View on chain - eq_chain

\[ \text{same}_\text{period}(c_1, c_2) \text{ and } \text{same}_\text{period}(c_2, c_3) \text{ and } \text{same}_\text{period}(c_3, c_1) \]

Synchronous simulation, test and verification tools apply

Efficiency issues ?
How to understand it?

- Communication Abstraction
- Continuous Systems
- Non Continuous Systems
- Mixed Systems
Communication Abstraction

Worst situation: reads occur just before writes

\[ x(t-2T) \]

\[ x'(t) \]

\[ T \]

Bounded communication delays
Uniformly Continuous Signals

\[ \forall \varepsilon > 0, \exists \eta > 0, \forall t, t', |t - t'| \leq \eta \Rightarrow |x(t) - x(t')| \leq \varepsilon \]

Bounded delays yield bounded errors
Uniformly Continuous Systems

∀ε > 0, ∃η > 0, ∀x, x', ||x - x'||_∞ ≤ η ⇒ ||f(x) - f(x')||_∞ ≤ ε

Bounded errors yield bounded errors
But …

Even very simple controllers are not uniformly continuous.

PID for instance

Bounded errors do not yield bounded errors
The closed-loop system computes uniformly continuous signals.

Bounded delays yield bounded errors.
Doubts …

This casts a doubt on two wishful thoughts:

- **composability**
  - system properties are the mere addition of sub-system ones

- **separation of concerns:**
  - automatic control people specify
  - computer science people implement

Critical control systems require a tight cooperation between both people
Non Continuous Systems

- Combinational Systems
- Robust Sequential Systems
- Sequential Systems
Uniform Bounded-Variability

There exists a minimum stable time $T_x$ associated with a signal $x$.

The analog of uniform continuity?
Sampling Tuples

A possible sampling
Sampling Tuples

Another possible sampling

Non deterministic bounded delays
But …

Delays on tuples do not yield delayed tuples

Solution: Confirmation functions
Confirmation Functions

When a component of a tuple changes, wait for some $\Delta_{\text{max}} - \Delta_{\text{min}}$ time before taking it into account.

If $x', y'$ are $(\Delta_{\text{min}}, \Delta_{\text{max}})$ bounded images of $x$ and $y$,

then $\text{confirm}(x', y')$ is a delayed image of $(x, y)$

allows to retrieve the continuous framework
Confirmation Functions

Net View on confirm - eq_confirm

\[ n_{\text{max}} = E\left(\frac{\Delta_{\text{max}} - \Delta_{\text{min}}}{T_{\text{min}}}\right) + 1 \]
Robust Sequential Systems

idea: avoid critical races

- between state variables: order insensitivity
- between inputs: confluence

Property checker
Can robustness analysis be avoided?

example: mutual exclusion

**Property:** always not \((y \text{ and } z)\)

a non robust solution:
Can robustness analysis be avoided?

**Example:** mutual exclusion

**Property:** always not \((y \text{ and } z)\)

A robust solution:

- \(z\)
- \(y\)

**Same answer as for error analysis in continuous systems**
Robust solutions are distributable

a robust solution:

\[ \text{not } y \quad (\rightarrow y \rightarrow \text{not } y)^* (\rightarrow z \rightarrow \text{not } z)^*)^* \]

\[ \text{not } z \]

no critical race!
Non Robust Sequential Systems

require either soft or hard synchronization.

Time Triggered Architecture for instance.
Non Robust Sequential Systems

A soft synchronization algorithm

requires a speed-up by 4
Implementation

Net View on SYMCH - eq_SYNCH
Implementation

State Machine View - SCHED

idle

write1

wait1

write2

wait2

execute

NUP

1 : nu and not NUP

2 : NUP

1 : true

3 : (not nu) and not NUP

2 : NUP

1 : nu and not NUP
Mixed Systems

Example: Threshold crossing

Relates errors and delays: \( \tau = \frac{2\varepsilon}{|C'(t)|} \)

This analysis too should not be skipped
Concurrency

Actual Practices (Airbus)
Concurrency

A Crisys Proposal: earliest deadline preemptive scheduling

Schedulability condition

\[ \sum_{i=1}^{n} \frac{WET_i}{T_i} < 1 \]
Concurrency

A Crisys Proposal: earliest deadline preemptive scheduling

Schedulability condition

$$\sum_{i=1,n} \frac{WET_i}{T_i} < 1$$

Generalizes the synchronous program execution condition

$$WET < T$$
Concurrency

Exact functional semantics is guaranteed as soon as

Slow processes communicate with fast processes through a slow clock unit delay

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>t</th>
<th></th>
<th>f</th>
<th>t</th>
<th></th>
<th>f</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x0</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f(x</td>
<td>f(x0)</td>
<td>f(x2)</td>
<td>f(x4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z = z0Δf(x</td>
<td>z0</td>
<td>f(x0)</td>
<td>f(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(z0,z) ↑ c</td>
<td>z0</td>
<td>z0</td>
<td>f(x0)</td>
<td>f(x0)</td>
<td>f(x2)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fault Tolerance

• Continuous Computations: Threshold Voting

  – Units differ from more than the maximum normal error
Fault Tolerance

- **Continuous Computations**: Threshold Voting
  - Units differ from more than the maximum normal error

- **Combinational**: Bounded-Delay Voting
  - Units differ from more than the maximum normal delay
Fault Tolerance

- **Continuous Computations**: Threshold Voting
  - Units differ from more than the maximum normal error

- **Combinational**: Bounded-Delay Voting
  - Units differ from more than the maximum normal delay

- **Sequential Computations**: 2/2 Bounded-Delay Voting
Bounded-Delay Voters

Net View on vote2_2 - eq_vote2_2

\[ n = E\left(\frac{\Delta_{\text{max}} - \Delta_{\text{min}}}{T_{\text{min}}}\right) + 1 \]
Sequential Computations

Idea: vote on input and on state

But Byzantine problems

2/2 votes are not sensitive to Byzantine problems:

- a bad unit is only compared with a single good one:
  - it agrees: it looks good
  - it disagrees: a fault is detected.
Sequential Computations: 2/2 Sequential Voters

Net View on SeqVote - eq_SeqVote

nx = nmax_u + nmax_x  \quad nnx = n \times nx
Proof Hints

\[ X = F(X, U) \quad X_1 = F(X, U_1) \quad X_1 = F(X_1, U_1) \]

\[ \tau_u \quad \tau_x \]

\[ X_1 = F(X, U_1) \quad X_1 = F(X_1, U_1) \]

\[ \tau_u \quad \tau_x \]
Conclusion

• Some insight on techniques used in practice.

• maybe useful for designers and certification authorities
  (Crisys Esprit Project)

• An attempt to catch the attention of the Computer Science Community on these important problems.
Questions

- When are clock synchronization methods useful and more efficient than the ones presented here?
Questions

- When are *clock synchronization* methods useful and more efficient than the ones presented here?
- How to safely encompass some *event-driven* computations within the approach?
Questions

• When are clock synchronization methods useful and more efficient than the ones presented here?

• How to safely encompass some event-driven computations within the approach?

• Are there linguistic ways to robustness (synchronous-asynchronous languages)?
Questions

- When are **clock synchronization** methods useful and more efficient than the ones presented here?
- How to safely encompass some **event-driven** computations within the approach?
- Are there **linguistic** ways to robustness (synchronous-asynchronous languages)?
- Is there a **common framework** encompassing both theories?

<table>
<thead>
<tr>
<th>continuous</th>
<th>discrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniformly continuous signals</td>
<td>uniform bounded variability</td>
</tr>
<tr>
<td>uniformly continuous functions</td>
<td>robust systems</td>
</tr>
<tr>
<td>unstable systems</td>
<td>sequential non robust systems</td>
</tr>
</tbody>
</table>