Implementation of synchronous models on asynchronous execution platforms: correctness, modularity, and performance analysis

Stavros Tripakis
UC Berkeley

Caltech Workshop on Verification and Validation, Sep 23-24, 2009
Embedded Systems

How can we build these systems both reliably and efficiently?
Embedded system tools

Automotive powertrain system modeled in Simulink

Simulink: 1 million licenses in 2004

Key concepts:
- behavior
- concurrency
- timing
- I/O
- ...

Key capabilities:
- simulation
- code generation
- verification
- ...

Dymola Multi-Engineering Modeling and Simulation
Vision

• **Modeling** languages of today will be the **programming languages** of tomorrow
  – At least in the embedded system domain

• **Programming languages** evolve with time
  – From assembly, to structured, to object-oriented programming, ...
  – **High-level** concepts: master complexity => increase productivity

• **Key concepts for embedded systems:**
  – Behavior, Time, Concurrency, I/O protocols, ...
  – Not well-supported by standard programming languages: C++, Java, ...
  – Better supported by: Simulink, Labview, Modelica, UML, SystemC, ...
What is our job?

• **Modeling/programming:**
  – Invent the right languages: with the right abstractions

• **Analysis:**
  – Check correctness, measure performance, ...

• **Implementation:**
  – Build executable systems: reliable and inexpensive
Model-based design: semantics-preserving implementation

- Design in a high-level language
  - Verify correctness, check performance, etc.
    - As much as possible at the high level!
  - Synthesize automatically correct-by-construction implementations
    - “What you verify is what you execute”
    - Preserve the semantics!
- Reduce the need for testing => reduce cost
Our work (2002 – present)*

* With colleagues from Verimag, UC Berkeley and Cadence
Plan of talk

• Synchronous block diagrams

• Two problems:
  – Semantics-preserving distribution
  – Modular code generation

• Conclusions and perspectives
Plan of talk

• Synchronous block diagrams

• Two problems:
  – Semantics-preserving distribution
  – Modular code generation

• Conclusions and perspectives
Synchronous block diagrams

- **Synchronous semantics:**

```
+----------------+    +----------------+    +----------------+    +----------------+
| Throttle       | →  | Engine          | →  | Torque Converter| →  | Gearset and Shift Mechanism| →  | Vehicle Dynamics |
|                |    |                 |    |                 |    | Transmission              |    |                  |
|                |    |                 |    |                 |    | Transmission Control Unit |    | brake            |
|                |    |                 |    |                 |    |                             |    |                  |
+----------------+    +----------------+    +----------------+    +----------------+
```

Time

inputs | inputs | ... | outputs | outputs

Copyright The Mathworks
Why synchronous block diagrams?

• Basis of de-facto standards: Simulink (automotive), SCADE (avionics), ...

• Fundamental model of concurrency
  – C.f. synchronous circuits

• Deterministic semantics
  – Results do not depend on block interleaving (as long as dependencies are respected)
    • Contrast this with threads
  – Easier to understand
  – Easier to verify (less state explosion that asynchronous models)
Example: synchronous block diagram

A → B → C

A, B, C | A, C, B | ...  rounds
Example: synchronous block diagram

deterministic concurrency
Plan of talk

• Synchronous block diagrams

• Two problems:
  – Semantics-preserving distribution
  – Modular code generation

• Conclusions and perspectives
From synchronous models to asynchronous distributed implementations

Joint work with
Claudio Pinello, Cadence
Alberto Sangiovanni-Vincentelli, UC Berkeley
Albert Benveniste, IRISA
Paul Caspi, VERIMAG
Marco di Natale, SSSA
Implementation on asynchronous distributed platforms

Asynchronous distributed platform:
- Many computers, each with a local clock
  - No clock synchronization
- Computers communicate using some network/protocol
  - Don’t care which network, as long as finite FIFO queues (TCP) can be implemented on top
Implementation on asynchronous distributed platforms

**synchronous model**

Intermediate layer: asynchronous processes communicating with finite FIFO queues

asynchronous platform with some communication network
Implementation on asynchronous distributed platforms

**synchronous model**

This is like Kahn Process Networks with blocking write() when FIFO is full.

Intermediate layer: asynchronous processes communicating with finite FIFO queues

FIFOs must be large enough to avoid deadlocks.

=> semantical (stream) preservation
Semantical preservation: proof

• Use old theories [1970s]:

• Marked graphs
  – Subclass of Petri Nets
  – Used to show FFP liveness (no deadlock)

• Kahn Process Networks
  – Used Kahn’s fundamental result: determinism
  – Streams do not depend on process interleaving
Performance analysis: worst-case logical-time throughput

WCLTT = 1/2

WCLTT = 1

Computing worst-case logical-time throughput

Theorem 9: Let $\mathcal{F}$ be an SFFP. Let $\Delta$ be any positive real number. Let $c$ be a vector of clocks such that $\forall i, \forall n, c_i(n+1) - c_i(n) \leq \Delta$. Then, for any process $P_i$ of $\mathcal{F}$:

$$\lambda^*(\mathcal{F}, P_i, c) \geq \frac{\lambda^*(\mathcal{F}, P_i)}{\Delta}$$
Performance analysis: worst-case logical-time latency

Worst-case logical-time latency: computation based on reachability

3 firings of P2

Relating real-time and worst-case logical-time latency

**Theorem 17** Let $F$ be an SFFP. Let $\Delta$ be any positive real number. Let $c$ be a vector of clocks such that $\forall i, \forall n, c_i(n + 1) - c_i(n) \leq \Delta$. Then, for any path $\pi$ of $F$:

$$\mu^{rt}(F, \pi, c) < \Delta \cdot (\mu^+(F, \pi) + 1)$$
Plan of talk

• Synchronous block diagrams

• Two problems:
  – Semantics-preserving distribution
  – Modular code generation

• Conclusions and perspectives
Modular code generation from synchronous block diagrams

Joint work with
Roberto Lublinerman (Penn State)
Christian Szegedy (Cadence)
Hierarchy in synchronous block diagrams (and Simulink, SCADE, Ptolemy, ...)

![Diagram of hierarchical block diagram with blocks A and B connected in series]
Hierarchy in synchronous block diagrams (and Simulink, SCADE, Ptolemy, ...)

Fundamental modularization mechanism: hide details, master complexity
Separate compilation

We want to do the same for synchronous block diagrams
Modular code generation

• Goal: generate code for a given block P

• Code should be independent from context:

  Will P be connected like this?

  – Enables component-based design (c.f., AUTOSAR)

  ...or like that?
Problem with standard approach: “monolithic” code

False I/O dependencies => code not usable in some contexts

P.step(x1, x2) returns (y1, y2)
{
    y1 := A.step( x1 );
    y2 := B.step( x2 );
    return (y1, y2);
}
Our solution

- Generate for each block an INTERFACE
- Interface may contain MANY functions

```plaintext
P.step1( in1 ) returns out1 {
    return A.step( in1 );
}

P.step2( in2 ) returns out2 {
    return B.step( in2 );
}
```
Our solution

P.step1( in1 ) returns out1 {
    return A.step( in1 );
}

P.step2( in2 ) returns out2 {
    return B.step( in2 );
}
Our solution

The function call order depends on the usage of the block

A Mealy machine with multiple output functions
How it’s done

• An **interface synthesis** problem
  – Given interfaces for children blocks, synthesize an interface for the parent

• Interface extraction = **automatic abstraction**:
  – Difficult problem for general SW
  – Easier for these domain-specific languages: also provably **optimal**
Trade-offs

• **Modularity vs. Reusability**
  – The smaller the interface, the more modular
  – If too small, some information is lost => less reusable (false I/O dependencies)

• Modularity vs. Code size

• **Bounds:**
  – At most N+1 interface functions to achieve maximal reusability

• **Complexity**
  – Polynomial vs. NP-complete
Extensions to timed diagrams

Firing Time Automata

\[ P = A \cup B \]

FTA division
Plan of talk

• Synchronous block diagrams

• Two problems:
  – Semantics-preserving distribution
  – Modular code generation

• Conclusions and perspectives
Conclusions and perspectives

• Model-based design for embedded systems:
  – a non-trivial problem

What properties should be preserved?

Super-duper Compiler!

Richer languages

more complex execution platforms

more powerful analyses

new challenges for CS + EE
Thank you

• Questions?