MIMOS a Framework for Design and Update
of Real-Time Embedded Systems

Susanne Graf
Verimag - Grenoble University &
Uppsala

Wang Yi, Bengt Jonsson, Philipp
Rümmer, Morteza Mohaqeqi …
Uppsala University
MIMOS: A Framework for Design and Update of Real-Time Embedded Systems

Susanne Graf
Verimag - Grenoble University & Uppsala
Wang Yi, Bengt Jonsson, Philipp Rümmer, Morteza Mohaqueqi, ...

2 supporting projects:

CUSTOMER: Customizable Embedded Real-Time Systems: Challenges & Key Techniques (Wang Yi's ERC)
UPDATE: Designed for Update of Next-Generation Embedded Systems (Knut & Alice Wallenberg Foundation)
both until end of 2026
Model-Based Design of Real-Time Systems

Function Architecture

Software Architecture (task layer)

Hardware Architecture

Physical Architecture

Functional Modelling & Verification

Mapping

Scheduling

Code running on HW: generated from the model and deployed
Update in this picture?

Function update:
change/upgrade functions, add
new functions, eliminate, reconnect ...

Approach: contract-based

Design challenge:
• composability
• Support for impact analysis
• independence of time and function
Update in this picture?

“Software” update:

Can we guarantee that

- small changes at function layer lead to small changes at software layer?
- Small increase of workload leads to small resource usage increase
- Efficient deployment of updates (container based)
- ...

**Challenge:** Resource-Efficiency

- Keeping track of workload and available resources
- Dynamic scheduling and schedulability analysis
- Architecture challenges: multicores, distributed ...
1. Motivations: on Design and Update of Real-time systems

2. System Design in MIMOS
   - Requirements on the design language
   - Our design language

3. A Type system for MIMOS
   - Boundedness as type correctness

4. Contracts for MIMOS
   - Some reflections on property specification and verification at the function layer
Function Design

F1: periode p1

F2: periode p2

F3: periode p3

Functions: Streams (& Memory) → Streams
Hierarchically defined

Requirements:

- **Determinism** is fundamental
- Separation of Concern: Abstraction
  - Independence of timing and functionality
- Updatability/Composability
  - Avoidance of interference & Resilience
  - Asynchronous Communication (non-blocking)
Current Approaches

Function Design and Verification:

**Synchronous** (Scade, Lustre, Synchronous Data-Flow, ...)

Advantages:

- Deterministic
- Time and function reasonably independent
- (Mostly) easy to design
- Easy to simulate and verify

Software level: “**Virtually Synchronous**” : semantic preserving mapping to a task set run asynchronously on OS / middleware

- **TTA** (Timed Triggered Architecture) – single rate [HK&al 90ies]
- **PALS** (Physically Asynchronous Logically Synchronous) [JM&al 09]
- Recent LF (**Lingua Franca**) Reactor Model [EL&al 2019]
Why do we need something different?

There are problems: synchronous is

- Very good for single rate (computation within period, communication between periods)
- Can be adapted to multi-rate (some loss of simplicity)
- Problematic for **deadline > period** ... can be done but delay changes function ...

\[ f(r_k, s_{k+1}) = F(e_k) \]

Period \( p \)

\( \delta = 2p \)

\( \delta = 3p \)

Insert a delay

No!

\( f(r_k, s_{k+1}) \)
Kahn Process Networks (KPN)


**Semantics** of KPN: stream transformation as a fix point

**Operational model** for KPN

A network of processes
- Communicating through (potentially unbounded) FIFO buffers
- Read (and compute a step): when all required data is in the FIFOs (blocking, similar to PetriNets)
- Write: non-blocking (asynchronous)
Properties of KPN

- **Determinism**: a KPN defines a function from input streams to output streams
  - independent of the execution orders/scheduling
  - Independent of computation/communication delays

- **Boundedness of FIFOs**: undecidable in the general case (expressiveness)

  - MIMOS: **Typed** KPN will make it “tractable”

**Observation**: a synchronous program is a KPN with (very) strict constraints on execution order which guarantees: when a node is executed, FIFOs contain exactly the input to be read (→ no need for FIFOs)
Timed KPN

**KPN extended with timing constraints** [YMG-Coordination-22]

- **Nodes**: real-time tasks with a *period* and a *deadline* (or possibly other recurring task release strategy).
- **Execution rule**:
  - Read input at release times (if present)
  - Write output at deadline (or relaxed “upto” deadline)
- **Extensions (optimizations and resilience)**:
  - Registers: keep only the most recent value of a FIFO (synchronous)
  - “timed read” (efficient implementation of “sporadic” tasks, resilience)

**Theorem**: TKPN are deterministic functions from timed input to timed output streams

**Observation**: in many cases, no need for time stamps
(Period p)

KPN solve the problem (by waiting for input)

Initial elements in buffers: e.g. in \( s \): shorten (minimal) delay from \( x \) to \( o \) – by using “older” values of \( s \) (synchronous solution)

\[ \text{Tradeoff between delay and precision} \]

**Deterministic:** function does not depend on the actual delays
Model-based Design with MIMOS

Typed KPN & contracts
Mapping
Timed KPN Node=RT task
Scheduling
Run-Time
1. Motivations: on Design and Update of Real-time systems

2. System Design in MIMOS
   - Requirements on our design languages
   - The design language and semantics

3. A Type system for MIMOS
   - Boundedness and deadlock freedom by type correctness

4. Contracts for MIMOS
   - Some reflections on property specification and verification at the function layer
A type system for KPN

Why a type system?

Remember: general form of a KPN “step function”:
Any program whose effect is (a) to read a finite number
of elements from the FIFOs and (b) to write at most a
finite number elements to its output and terminates

Problem:
- How many elements does the program read and write?
- Does it terminate?

To be able to give guarantees:
we must impose restrictions on such programs
A type system for KPN

A node: a typed function

Types:
- Basic Type (BT): Bool, Natural, Real, Int, Tuple/Product, List ...
- (bounded) Segment (Sgt): BT<sup>k</sup>, BT<sup>&le;k</sup>
- Interface Type (IT): tuple(Sgt)
- Step function (ft): IT → IT → the function to be implemented
- Node function (FT): ft<sup>ω</sup> = IT<sup>ω</sup> → IT<sup>ω</sup>

Abstract Types:
- Abstract Segment (ASgt): k (≤k)
- Abstract Interface (AIT): tuple(ASgt)
- Abstract step function (Aft): AIT → AIT
Bounded Memory Property

A Connector C from node N to N’

ASgt  
\[ k \]  ASgt’  
\[ k' \]

The FIFO associated with C is bounded if:

#produced tokens = #consumed tokens (on the long run)

If we know the periods of N and N’: \( p_N, p_{N'} \)

Then, the FIFO associated with C is bounded iff:

\[
\text{ASgt} / p_N = \text{ASgt} / p_{N'}
\]

That is: production rate = consumption rate

Fact: A KPN is bounded if all its connectors are bounded
Examples

Bounded KPN

\[ \frac{1}{10} = \frac{1}{10} \]

\[ \frac{1}{5} = \frac{2}{10} \]

\[ \frac{1}{10} = \frac{1}{10} \]

not Bounded KPN

\[ \frac{2}{5} > \frac{2}{10} \]
Do we need the periods?

A live and bounded cycle: all have the same period (no node can consume more than its predecessor produces)

A deadlock cycle: for any period assignments different 0, the cycle asks to consume more than it produces
Idea: “Close” the system with an abstract interface representing the input rates for some period $p$:

- We can calculate the “effective periods” of all nodes (if a solution exist).

Fact: we can efficiently compute the interface type of any composition.
Can we go further?

**Fact**: this “simple case” covers the “clock correctness” analysis of synchronous programs (Lustre)

Consider Connector C of type $\leq k \leq k$

A *deterministic protocol* allows dynamically adapting read-write strategies without impacting other components/connections
Fact: A type correct KPN is bounded memory and deadlock free. ... and if the execution time of all the programs implementing the step functions can be bounded, the e2e-delay of all system functions is also be bounded.

Important lesson

General form of the step function

1. Read strategy (deterministic)
2. Local computation
3. Write strategy (deterministic)

Type

Any other abstraction of read/write strategies can be used for type analysis ... as long as it can be handled by some tool.
OUTLINE

1. Motivations: on Design and Update of Real-time systems

2. System Design in MIMOS
   - Requirements on our design languages
   - The design language and semantics

3. A Type system for MIMOS
   - Boundedness and deadlock freedom by type correctness

4. Contracts for MIMOS
   - Some reflections on property specification and verification at the function layer