Agenda

1. Background
2. State-of-the-art
3. Conceptual deficiencies
4. The link to synchronous languages
5. Requirements of embedded systems
6. Outlook: A new language?
I. Background

Our domain

Many Bosch products are driven by embedded software

- Rapid increase in:
  - number of products,
  - their functionality,
  - complexity

- Little progress in:
  - methodology and tools

- We need to advance:
  - Analysis
  - Architecture
  - Implementation
  - Verification
II. State-of-the-art

Programming Frontend

- Assembler → C → Simulink, ASCET
II. State-of-the-art

Deployment Backend

- Runnables: void-void C function
  - no inputs, no outputs, operates on global variables

- Runnables are ordered in sequence to form a task
  - Sequence $\text{T } f() \text{ g() h()}$
  - Tasks may be ordered by priority
  - Tasks $\approx$ clock rates (e.g. 1ms, 5ms, 10ms, ...)

- Stack of active tasks
  - A running task may be preempted by tasks with higher priority

For more details see: "Real world automotive benchmark for free" by Simon Kramer, Dirk Ziegenbein and Arne Hamann, WATERS 2015
III. Conceptual deficiencies

▶ Handling concurrency:
  ▶ Who is writing what variable and when?
  ▶ Ordering of runnables and tasks determined by a separate task list in XML
    – Implicitly introduces prev and current accesses without ever being documented

▶ State machine behaviour is either implicit or formulated in a separate monolithic model

▶ Nondeterminism:
  ▶ Above ordering has no formal criteria
  ▶ Communication between concurrent threads is non-deterministic
III. Conceptual deficiencies

Effects of deficiencies

- Adding new software is hard
- Most effort is spent in:
  - Reverse engineering to find out who is writing what variable and when
  - Composition of software components, requiring lots of meta data about side effects, timing constraints, ...
- Lack of software qualities such as:
  - Determinism & testability
  - Readability
  - Flexibility (refactoring!)
  - Modularity
IV. The link to synchronous languages

Why we are here

- We believe we can benefit greatly from the synchronous programming approach:
  - Behaviour over time
  - Preemptions and mode switches
  - Structured programming of state machines
  - Causality of concurrent functions
- We hope your research may benefit from industrial challenge
- So where is the challenge?
V. Requirements

Clear focus

- Software: Not hardware
- Embedded: Not “IT”-level software
- Reactive: Trigger-response execution
- Real-time: Time is functional, not a performance measure
- Resource-constrained hardware: No heap allocation, garbage collection
- Scale to software with millions lines of code: Not “wrist watch”
V. Requirements

Domain orientation

- Control intensive systems
- Intertwined functionality
- Computations and switching behaviour
- Preemptions
- Causality

Relaxed notion of causality is sufficient for software
= concurrent processes + shared variables + barriers!

Synchronous programming = Unique writer and
write before read between each pair of barriers.
V. Requirements

Compatible with the past and future

- Integration of legacy code
- Integration in legacy code
- Support separate compilation
- Address deployment on multi-core platforms
- Program across threads, cores maintaining guarantees such as causality
V. Requirements

Deployment

- Efficient code generation
  No definitive consensus yet?
- Safe code generation
  Runtime errors shall be impossible on a final system
- Integrate synchronous “execution shell” with existing real-time OS environments
- Low level mapping to cores and tasks
V. Requirements
Developer orientation

- Readable programs
  Programs are mostly read, sometimes adapted and almost never written from scratch

- Crystal clear semantics
  Make it hard to write nonsense, make it obvious what any piece of code does

- Express stateflow in control flow

- Provide structured data types
  These cannot be disintegrated into primitives (arrays, structs, enums)

- Enable structuring, information hiding
  Structures cannot be just macros that are instantiated and inlined

- Provide a safe and modern type system
  Physical units, sum types
V. Requirements

Testing and verification

- Easy testing
  - Write tests in the same language and compose concurrently with production code

- Integration with existing simulation frameworks
  - E.g.: Simulink, Functional Mockup Interface

- Generate verification conditions for abstract interpreters
  - Lots of assertions (no 0-division, no out-of-bounds access, ...) are never specified by the programmer but are trivial to generate and significantly help to find bugs
V. Not requirements

By focusing we gain a few degrees of freedom

- What we do not need
  - Hardware related issues
  - Single value per tick
  - Reordering of commands
  - Fine grained causality based on logical constructiveness or dynamic analyses
  - Full range of preemption expressions

- And hence no
  - Schizophrenia
  - Fix point computations
  - Intricate surface/depth compilation
VI. Outlook

Do we need something new?

- No off-the-shelf solution available which meets above requirements
- Theoretically most requirements are straight forward
- Some however are not
  - True parallelism
  - Deployment
  - OO, references vs. causality

- We have a vision that all requirements together lead to a new language with a new compiler and IDE that support (most of) the above
- And we believe this will significantly improve the implementation methodology of embedded systems
- And there is the first practical evidence that a paradigm shift may be of interest to real-life developers