Expressing Parallelism and Timing in Embedded Real-Time Applications

Sören Braunstein*, Jochen Härdtlein*, Michael Philippsen**

* *Robert Bosch GmbH, Robert-Bosch-Straße 2, 71701 Schwieberdingen, Germany*

** *Universität Erlangen-Nürnberg, Informatik 2, Martenstraße 3, 91058 Erlangen, Germany*

ABSTRACT

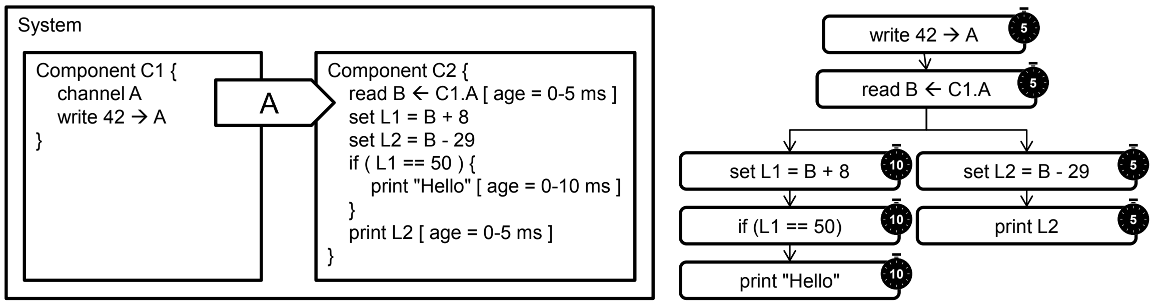
We present a domain specific language for hard real-time applications. It uses the concept of single variable assignment and allows the definition of timing constraints. The language has been designed to express the parallelism in real-time control system applications. The included editor runs static analysis and shows the degree of parallelism on the fly. The proposed language is part of an execution framework that is used to evaluate patterns for parallel programming.

1 Introduction

Current embedded real-time systems are equipped with multi-core processors. In future, the number of cores will grow massively. Systems with thousands of cores are imaginable [15]. There is still a gap between these hardware platforms and concepts for parallelism in available software. Expressing parallelism in software is a commonly occurring problem, and therefore many different software design patterns have been developed to address this issue [9, 12]. While these patterns are designed for grid, distributed, or personal computing, they are not targeted towards being used in software for real-time control systems. The existing patterns for parallel software modeling have to be re-evaluated and new patterns have to be found for this application domain.

In software development for real-time control systems, parallelism is not directly addressed by used modeling tools and programming languages such as C, ASCET [4] or Simulink [14]. This forces developers to write sequential software. The current support for writing parallel software is inappropriate. Sequential software usually cannot utilize the advantages of currently available and future parallel hardware platforms.

Not only do existing parallel patterns not match the requirements of embedded real-time [10, 2] and especially control systems, but also lack currently used programming languages for the embedded real-time domain any support for parallelism. We therefore introduce a language that can express parallelism and parallel patterns in an early state of the software development process. The language is designed to help to avoid common coding pitfalls by means of static analysis and to explore how much inherent parallelism is in the developed control system.



```
System
Component C1 {
   channel A
   write 42 → A
}
Component C2 {
   read B ← C1.A [ age = 0-5 ms ]
   set L1 = B + 8
   set L2 = B - 29
   if ( L1 == 50 ) {
      print "Hello" [ age = 0-10 ms ]
   }
   print L2 [ age = 0-5 ms ]
}
```

Figure 1: Two components of a system block (left): Component ′C1′ provides a channel ′A′ that is read by component ′C2′ and used in in the variables ′L1′ and ′L2′. The generated data-flow graph (right) has each instruction as a sequential element according the "communication sequential elements" pattern. The stop watch icons indicate the calculated and derived execution frequencies.

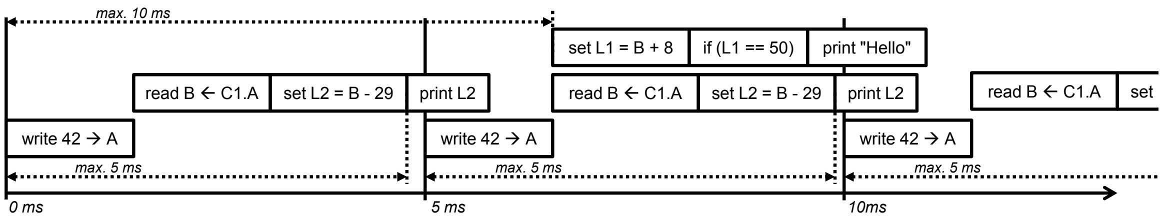


Figure 2: Possible schedule were the instructions are executed according the calculated frequencies. It is ensured that the given age constraints are met.

2 Central Language Ideas

The presented language is based on the "communicating sequential elements" pattern and the concept of single variable assignment. The concept of single variable assignment reduces the data dependencies to true data dependencies (flow dependencies). As anti or output dependencies are avoided, concurrency can be exposed easily [8, 5]. Other data-flow-oriented languages are Lustre [6] and VHDL [11]. Related modeling technologies are ASCET and Simulink. In a control system, sensors read the environmental variables. Afterwards, this data "flows" via one or several controllers to the system's actuators [10]. Therefore, a data-flow-oriented language perfectly suits the nature of control systems. As all data dependencies are known the source code can be transformed into a data-flow graph for further analysis. Nodes of this graph are single instructions like variable instantiations, arithmetic operations, sequential instructions, if-else statements, or loops. A "component" in the code is a structural element that groups source code according to its functional coherence and that has multiple "channels", a kind of fixed-size message queue. The defining component can write immutable values to its channels. Other components can read these values from the provided channels. This concept is based on the before-mentioned communicating sequential elements pattern. Hence component hierarchies can be resolved easily during the generation of the data-flow graph as shown in Figure 1.

Correct timing is crucial in control systems. The frequency of a controller within a control system is an important variable of the overall control algorithm. In most control systems, the frequencies are static to simplify calculations. Therefore, the timing description is another important aspect of the proposed language. Available timing definition languages, e.g. TDL

[13] and TADL [1], or general modeling languages e.g. ASCET [3] and AADL [7] provide structures to define timing. Such a timing definitions can be the last point in time when a component must write a value to an actuator.

Each consumer of a channel defines an age constraint, i.e. a minimal and/or a maximal age for the values it wants to read from the channel. The defined timing information is added for further analysis to the data-flow graph and is used to calculate the minimal execution frequency of each instruction by traversing the graph backwards. During execution, it is guaranteed that the consumed values adhere the given constraints. Figure 2 illustrates this behavior. This ensures the developer intended behavior of the control system.

3 Static Code Analysis

The single assignment semantics and the data-flow principle do not only reduce data dependencies but also allow static analysis within the editor. The goal of the static analysis is to equip the programmer with tools and information about the degree of parallelism in his software. Following features are provided in the editor:

Unintended Oversampling. Oversampling happens if a consumer runs less frequently than its provider. This can happen if two consumers read from the same channel having two different time demands. Because of the data-flow principle and the timing definitions, the static analyzer can easily detect oversampling. There are two kinds of oversampling recognizable so far. Oversampling on a read instruction is caused by another read instructions with a higher timing demand, or a write instruction that writes to the same channel which is read more frequently. Oversampling does not have to be a bug. Sometimes oversampling is once needed, for example to filter a signal. The decision is to the programmer if oversampling is functional necessary or not. Nevertheless, reducing unintended oversampling decreases the system utilization and increases the overall system efficiency and performance.

Coverage Checks. The consumer-driven approach helps to detect if components produce values that are not consumed, i.e. that does not effect any actuators. Static analysis can detect and point out such "dead-ends".

Bottlenecks. Another aspect of the provided language is to give the software developer hints about the degree of parallelism in his software. We plan to automatically detect bottlenecks or long paths of sequential data-flows. All in all, the given feedback can aid the developer to produce less sequential source code.

4 Future Work

The requirements for an embedded real-time systems are advanced timing constraints such as predictability. This includes very short response times (e.g. in automotive systems less than 1 millisecond) and very precise timing. A sufficient level of determinism is important to ensure a provable level of quality, which is very important for safety critical applications.

In the future, we will implement several control systems with the proposed design language to verify the concepts and their applicability. Furthermore, we will evaluate parallel programming patterns with respect to real-time constraints in embedded environments.

References

[1] Blom, H., Johansson, R., and Lönn, H. Annotation with Timing Constraints in the Context of EAST-ADL2 and AUTOSAR - The Timing Augmented Description Language. Workshop on the Definition, evaluation, and exploitation of modelling and computing standards for Real-Time Embedded Systems, STANDRTS'09.

[2] Ebert, C., and Jones, C. Embedded Software: Facts, Figures, and Future. *Computer 42*, 4 (2009), 42–52.

[3] Eppinger, A. ASCET - Computer Aided Simulation of Engine Functions. IEE Colloquium on Computer Aided Engineering of Automotive Electronics, pp. 511–513.

[4] ETAS Group. ASCET V6.1 AUTOSAR, XML, MISRA-C and more. White paper published on ETAS website. http://www.etas.com/en/downloadcenter/15284.php, January 2011.

[5] Giloi, W. *Rechnerarchitektur*. Springer-Lehrbuch. Springer, 1993.

[6] Halbwachs, N., Lagnier, F., and Ratel, C. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering 18*, 9 (September 1992), 785 –793.

[7] Hudak, J., and Feiler, P. *Developing AADL Models for Control Systems: A Practitioner's Guide*. Software Engineering Institute Carnegie Mellon, 2007. Published on SEI CMU website. http://www.sei.cmu.edu/library/abstracts/reports/07tr014.cfm.

[8] Kelly, W. Finding Inherent Parallelism. Slides presented at the LCA2011 Multicoreand Parallel Computing Miniconference. http://multicorelca.wordpress.com/.

[9] Keutzer, K., and Mattson, T. A Design Pattern Language for Engineering (Parallel) Software. Tech. rep., OPL Working Group, 2009. Published online at Intel Technology Journal. http://www.intel.com/technology/itj/2009/v13i4/ITJ9.4.2_DesignPatternLanguage.htm.

[10] Lee, E., and Sehsia, S. *Introduction to Embedded Systems - A Cyber-Physical System Approach*. UC Berkeley, 2011.

[11] Lis, J., and Gajski, D. Synthesis from VHDL. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '88* (Rye Brook, NY (USA), October 3, 1988), pp. 378 –381.

[12] Ortega-Arjona, J. *Patterns for Parallel Software Design*. Wiley Software Patterns Series. John Wiley & Sons, 2010.

[13] Pree, W., and Templ, J. Modeling with the Timing Definition Language (TDL). Slides presented at Center for Hybrid and Embedded Software Systems. http://chess.eecs.berkeley.edu/pubs/157.html, September 26, 2008.

[14] The MathWorks, Inc. Simulink 7 simulation and model-based design. White paper published on MathWorks website. http://www.mathworks.com/tagteam/43815_9320v06_Simulink7_v7.pdf?s_cid=SL2012_bb_datasheet, September 2007.

[15] Uhrig, S. Risks and Chances of Many-core Processors. International Conference on High Performance Computing Simulation, HPCS '09.

Expressing Parallelism and Timing in Embedded Real-time Applications

Sören Braunstein (Bosch/FAU), Jochen Härdtlein (Bosch), Michael Philippsen (FAU)

We introduce a language that can express parallelism and parallel patterns in an early state of the software development process. The language is designed to help to avoid common coding pitfalls by means of static analysis and to explore how much inherent parallelism is in the developed control system.

Problem Statement:

- Increase of parallelism in embedded real-time systems
- Current parallel patterns not designed for embedded real-time

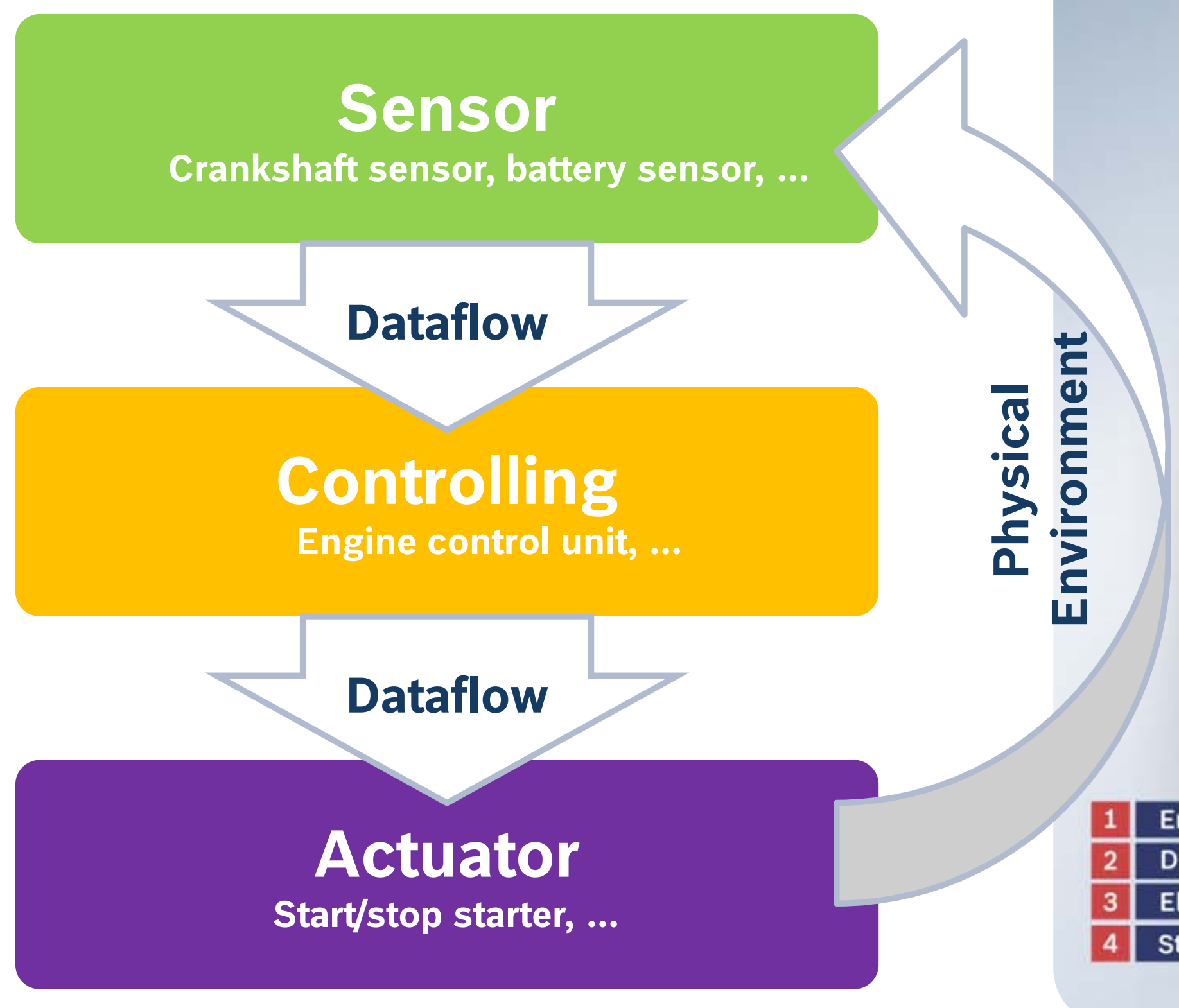
Central Ideas :

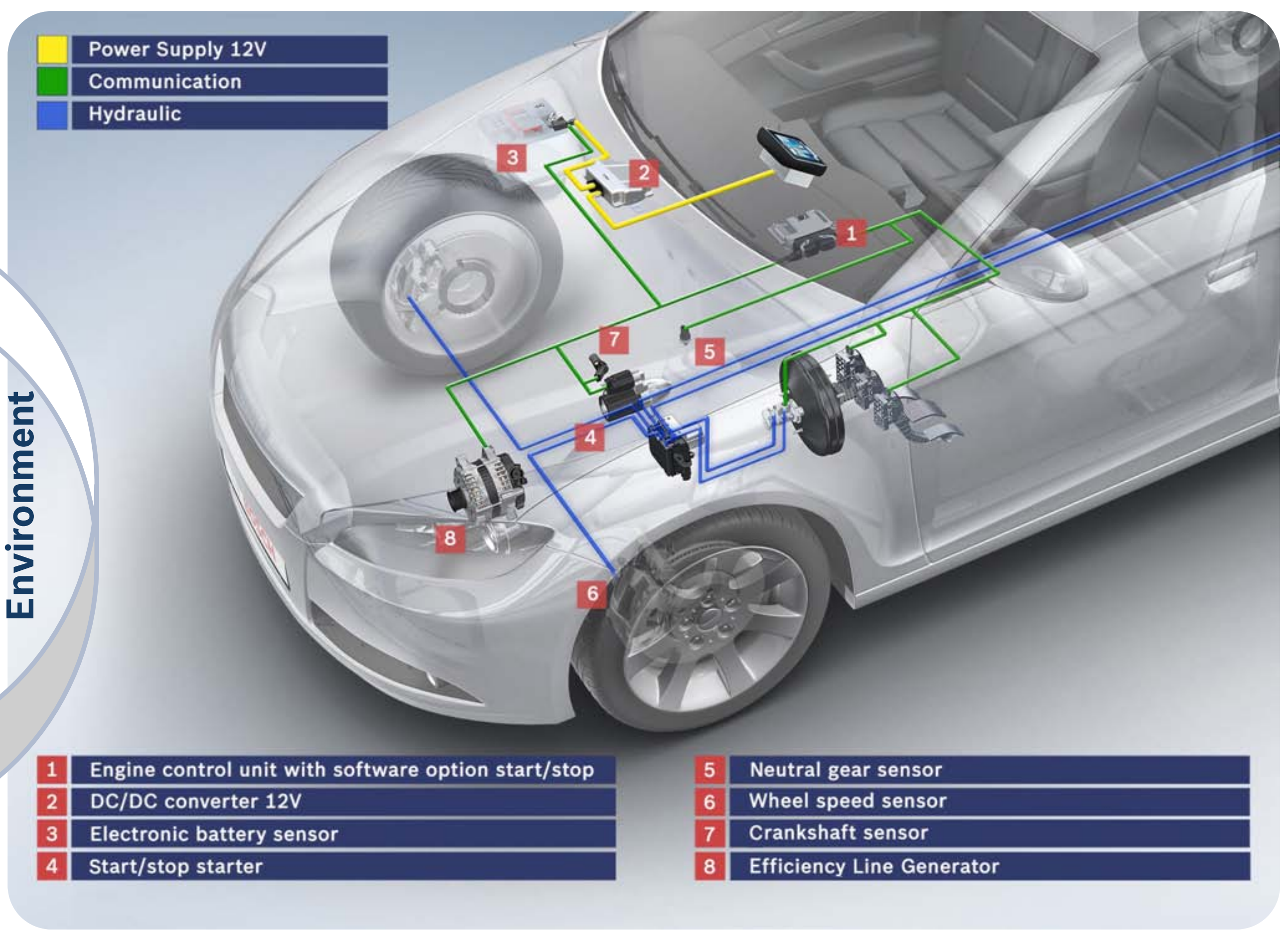
- Reduction of data dependencies by single variable assignment
- Use of “communicating sequential element” patterns for easy parallelization
- Expressing timing with minimum and maximum allowed value ages
- Consumer-driven execution

Overall Goals :

- Expressing and maximizing parallelism in control system applications
- Retaining real-time requirements

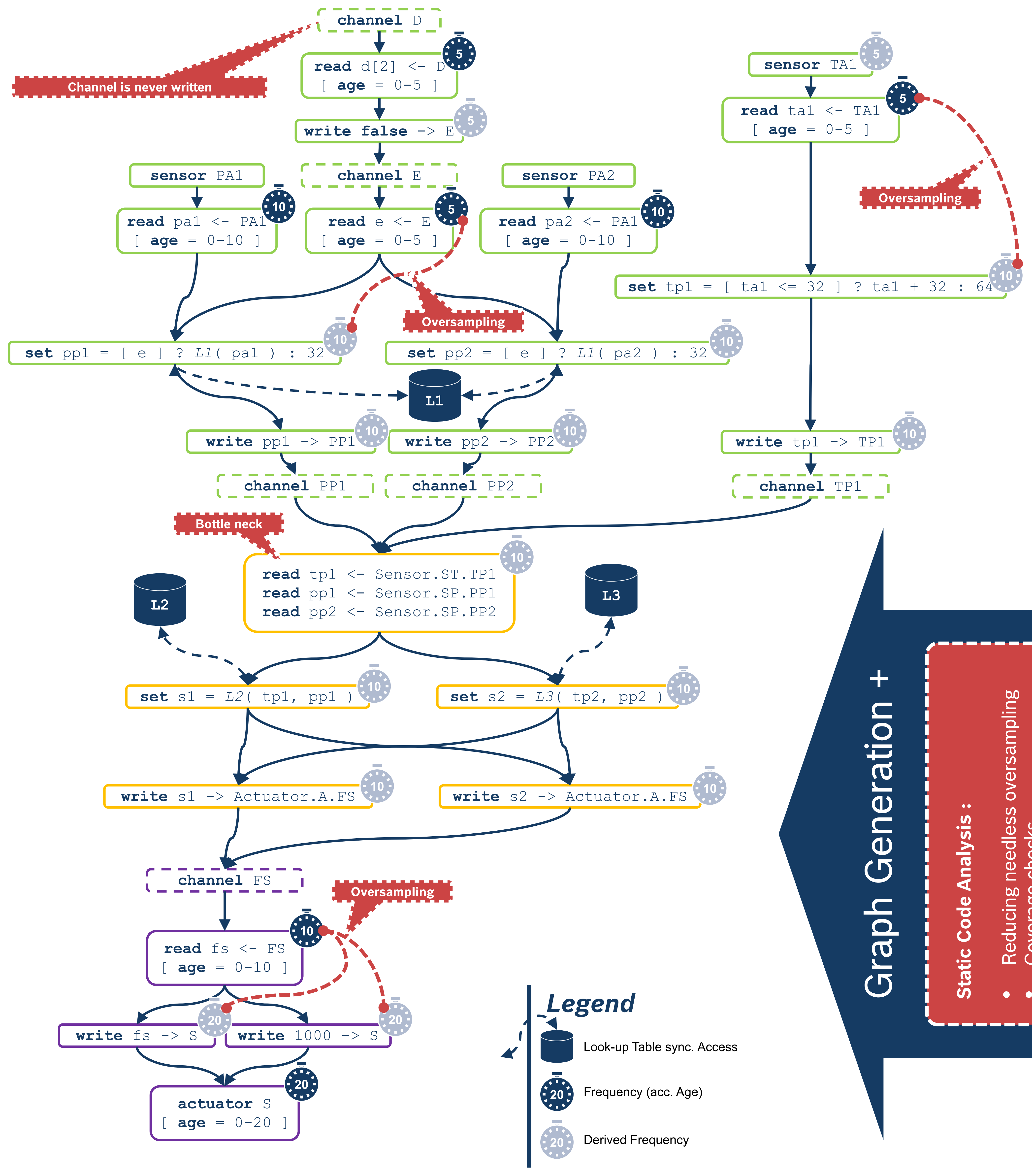
Typical control flow, here in a start/stop system :





Source: bosch-presse.de

Generated dataflow graph :


channel D
read d[2] <- D [age = 0-5]
Channel is never written
write false -> E
sensor TA1
read ta1 <- TA1 [age = 0-5]
Oversampling
sensor PA1
read pa1 <- PA1 [age = 0-10]
channel E
read e <- E [age = 0-5]
sensor PA2
read pa2 <- PA1 [age = 0-10]
set tp1 = [ta1 <= 32] ? ta1 + 32 : 64
Oversampling
set pp1 = [e] ? L1(pa1) : 32
set pp2 = [e] ? L1(pa2) : 32
L1
write pp1 -> PP1
write pp2 -> PP2
write tp1 -> TP1
channel PP1
channel PP2
channel TP1
Bottle neck
read tp1 <- Sensor.ST.TP1
read pp1 <- Sensor.SP.PP1
read pp2 <- Sensor.SP.PP2
L2
L3
set s1 = L2(tp1, pp1)
set s2 = L3(tp2, pp2)
write s1 -> Actuator.A.FS
write s2 -> Actuator.A.FS
channel FS
Oversampling
read fs <- FS [age = 0-10]
write fs -> S
write 1000 -> S
actuator S [age = 0-20]
Legend
Look-up Table sync. Access
Frequency (acc. Age)
Derived Frequency

Graph Generation +

Static Code Analysis :

- Reducing needless oversampling
- Coverage checks
- Detection of bottle necks

Code example :

```
system Sensor {
   sensor TA1;
   sensor PA1;
   sensor PA2;
   component ST {
      channel TP1;
      read ta1 <- TA1 [ age = 0-5 ];
      controller T {
         set tp1 = [ ta1 <= 32 ] ? ta1 + 32 : 64;
         write tp1 -> TP1;
      }
   }
   component SP {
      channel PP1;
      channel PP2;
      channel E;
      channel D;
      controller A {
         read pa1 <- PA1 [ age = 0-10 ];
         read pa2 <- PA1 [ age = 0-10 ];
         read d[2] <- D [ age = 0-5 ];
         read e <- E [ age = 0-5 ];
         set pp1 = [ e ] ? L1( pa1 ) : 32;
         set pp2 = [ e ] ? L1( pa2 ) : 32;
         if ( d[0] = 0 and d[1] = 1 ) {
             write false -> E;
         }
         write pp1 -> PP1;
         write pp2 -> PP2;
      }
   }
}
```

```
system Controlling {
   component CG {
      synced [ age = 0-20 ] {
         read tp1 <- Sensor.ST.TP1;
         read pp1 <- Sensor.SP.PP1;
         read pp2 <- Sensor.SP.PP2;
      }
      set s1 = L2( tp1, pp1 );
      set s2 = L3( tp1, pp2 );
      if ( s1 < s2 ) {
         write s1 -> Actuator.A.FS;
      } else {
         write s2 -> Actuator.A.FS;
      }
   }
}
```

```
system Actuator {
   actuator S;
   component A {
   channel FS;
   read fs <- FS [ age = 0-10 ];
   if ( fs < 1000 )
      write 1000 -> S;
   } else {
      write fs -> S;
   }
}
```

Note: The shown code and graph example is not taken from a real start/stop system. It is a made up application with no direct relation to any real product.

Contact :

Sören Braunstein (CR/AEA1)
Soeren.Braunstein@de.bosch.com

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

BOSCH