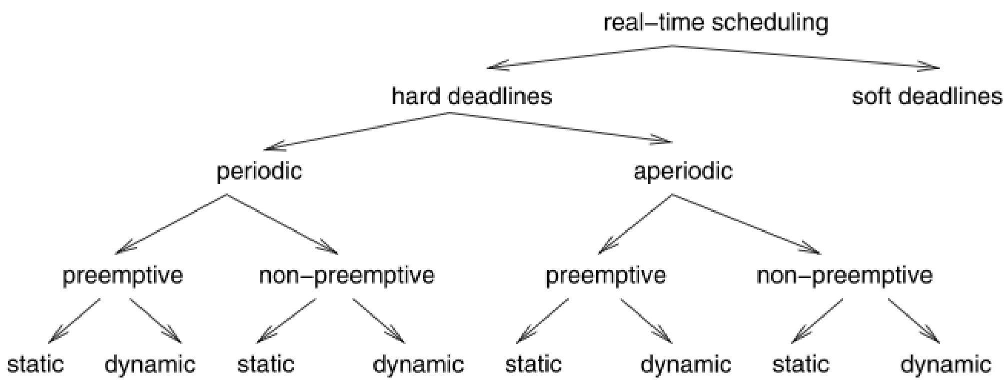
Functional Safety of Embedded Systems - Part 5 - Application Mapping

- Mapping of Application onto execution platform.
- Certain applications will be used together (e.g. call, blue-tooth, PIM) => still need to meet deadlines.
- **hardware-software-co-design** => find combination of hardware and software s.t. design is most efficient
    - => Platform-based design
- *homogeneous multiprocessor systems*: all processors provide same functionality (PC-like) => code compatibility between processors => reallocate processors at run-time (fault tolerance)
- *heterogeneous multiprocessor systems*: processors of different types => special processors for special tasks
- *definition of mapping problem*:
    - *given*: set of applications, use cases, set of candidate architectures
    - *find*: mapping of applications to processors, appropriate scheduling technique, target architecture
    - *objectives*: keep deadlines, maximize performance, minimize cost/ energy, ...
- *design space exploration* (DSE): exploration of possible architectural options

Scheduling in Real-Time Systems

- One of the key issues to be solved.
- Already roughly considered in Specification.

Classification of Scheduling Algorithms



- classes used to classify schedulers:
    - *soft and hard deadlines*:
        - soft => extension to OS-default
        - hard => periodic and aperiodic
    - *periodic and aperiodic tasks*:
        - periodic: Task must be executed once every p units of time, p is the period, each execution is a *job*
        - aperiodic: all other, if minimum time between two calls => *sporadic*
    - *preemptive and non-preemptive*:
        - preemptive: tasks may be interrupted
        - non-preemptive: tasks run until finished
    - *static and dynamic*:
        - static: decision about scheduling at compile/ design time (can respect resource requirements, dependences between tasks)
        - => *entirely time triggered* (TT systems): totally controlled by a timer, *task descriptor list* (TDL) contains schedule
        - dynamic: decision about scheduling at run-time (overhead run-time)
    - *independent and dependent tasks*: dependencies between tasks in embSys rule
    - *mono- and multiprocessor*: single processor more simple, multi => distinguish between hetero- and homogeneous systems
    - *centralized and distributed*: scheduling algorithm can either be on one processor or distributed among many
    - *type and complexity of schedulability test*: exact prediction is often NP-hard => sufficient and necessary tests => can show, that no schedule exists, but schedule can still not exists, even if successful
    - *cost functions*: different algorithms aim at different minimizations
        - *maximum lateness*: difference between deadline and completion, maximized over all tasks (negative => all tasks before deadline)

Aperiodic scheduling without precedence constrains

- c_i execution time, d_i deadline interval => $l_i = d_i - c_i$ **laxity** or **slack** => denotes time that a process can still wait

Earliest Due Date (EDD)

- *static scheduling* deadlines known in advance (=> preemption not used)
- sort tasks by deadline => $\mathcal{O}(n\log(n))$
- EDD is an optimal scheduling algorithm

Earliest Deadline First (EDF)

- is optimal with respect to maximum lateness
- *dynamic scheduling*: every time a job arrives, it is inserted into a list sorted by deadlines
- *preemption*: when task is inserted at head of list
- sorted lists => $\mathcal{O}(n^2)$

Least Laxity (LL), Least lack Time First (LST), Minimum Laxity First (MLF)

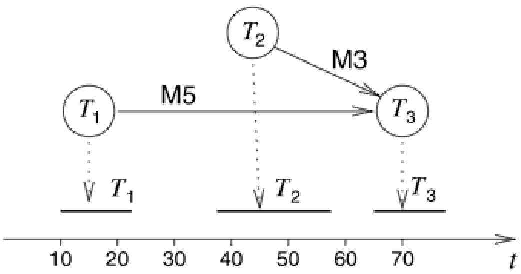
- task priorities are a monotonically decreasing function of laxity (less laxity => higher priority)
- negative laxities => deadline will be missed (early warning)
- *preemptive* (not only when new tasks become available)
- optimal for mono-processor systems => it will find a schedule if one exists

Scheduling without preemption

- processor may be forced to be idle sometimes for optimal schedule (tasks with early deadlines arrive late)
- needs knowledge about future (when will task with early deadline arrive)
- no knowledge about future? => EDF most optimal of all
- if arrival times known => NP-hard

Aperiodic scheduling with precedence constraints

Latest Deadline First (LDF)



- **a priory?**
- put task with no successor in queue, execution is opposite
- non-preemptive optimal for mono-processors
- asynchronous arrival => modified EDF

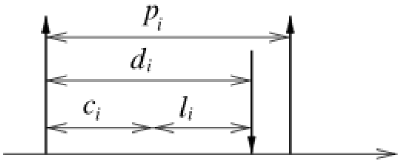
As-Soon-As-Possible (ASAP), As-Late-As-Possible (ALAP), Force-Directed (FDS), List (LS)

- popular in high-level synthesis community (HLS)
  - dependencies between tasks
  - multi-processor scheduling
  - simplified resources (processor) model
  - use heuristics (optimality not guaranteed)
  - are typically fast
  - almost never exploit global information about periodicity
- ASAP and ALAP => no resource or time constraints
- LS => resources constraints
- FDS => global time constraint
- ASAP: start tasks as soon as all inputs are available
- ALAP: start with tasks that no other tasks depend on, build task list in negative time domain => run-time: shift all values to positive
- LS: requires some priority function
- FDS: probability (likelihood, that task is scheduled), define forces (move tasks away from "high pressure" areas)

Periodic scheduling without precedence

- *Optimal Schedule*: For periodic scheduling, a scheduler is **optimal**, iff it will find a schedule, if one exists.
- pi: period, ci: execution time, di: deadline, li=di-ci: laxity

- µ *utilization* (accumulated (execution time/period)) $\mu = \sum_{i=1}^{n} \frac{c_i}{p_i}$



Rate Monotonic Scheduling (RM)

- Most well known for periodic processes.

- *RM assumptions* (when true: *the priority of tasks is a monotonically decreasing function of their period*):

  1. All tasks that have hard deadlines are periodic.
  2. All tasks are independent
  3. di=pi for all tasks
  4. ci is constant and is known for all tasks
  5. The time required for context switch is negligible.

  6. For a single processor and *n* tasks, the following equation holds for µ :

     $\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{\frac{1}{n}-1})$ The right side is about 0.7

- preemptive scheduling with fixed priorities

- schedulability not guaranteed (especially when µ is high)

Earliest Deadline First (EDF)

- can also be applied to periodic task sets
- *hyper period*: least common multiple of the periods of the individual tasks
- => only need to solve scheduling for one hyper period
- EDF can also solve for di!=pi

Periodic Scheduling With Precedence Constraints

- scheduling for dependent tasks more difficult (NP-complete)
- possible: add additional resources => scheduling becomes easier
- possible: partition scheduling into static and dynamic parts (make as many decisions at design time as possible)

Sporadic Events

- connect to sporadic events via interrupt, execute the immediately if priority is highest
- => very unpredictable behavior for all other tasks
- **sporadic task servers** => tasks, that are periodic and check for ready sporadic events

Hardware/ Software partitioning

- What must be implemented in hardware, what in software?
- For each node in task graph => need information about effort and benefits of hardware/ software implementation

COOL (CO-design toOL)

- partition on multi-processor infrastructure
- COOL input:
  - *target technology*: available hardware platform components (only homogeneous processors)
  - *design constraints* required throughput, latency, maximum memory size, maximum area for application specific hardware, ...
  - *behavior* hierarchical task graphs (communication & timing edges) in VHDL
- partitioning:
  1. Translation into internal graph model.
  2. Translation of behavior from VHDL to C
  3. Compilation of C programs for selected processor type
  4. Synthesis of hardware components: for each leaf node

5. Flatten the hierarchy: extract flat task graph => no merging/ splitting of nodes, cost from compilation and hardware synthesis are added
6. Generate and solve mathematical model of optimization problem: integer linear programming (ILP) to solve
7. Iterative improvements: merge nodes on the same hardware (better communication estimates)
8. Interface Synthesis: glue logic required for interfacing processors, application-specific hardware and memory is created.

- **0/1-ILP**-model:
  - sets: V -> graph nodes, L -> node types, M -> component types, hardware can have multiple instances j, KP -> processors
  - decision variables: X_vm node-v-to-hardware-m, Y_vk node-v-to-processor-k, NY_lk node-of-type-l-to-processor-k, Type V->L from task graph node to type
  - cost function: C = processor cost + memory cost + cost of application specific hardware
  - implementation either in hardware, or in software
  - *resource constraints* => ensure, that "not too many" nodes on one component
  - *precedence constraint*: ensure that schedule is consistent with task graph
  - *design constraints*: put limit on cost of hardware components
  - *timing constraints*: input into COOL => convert to ILP

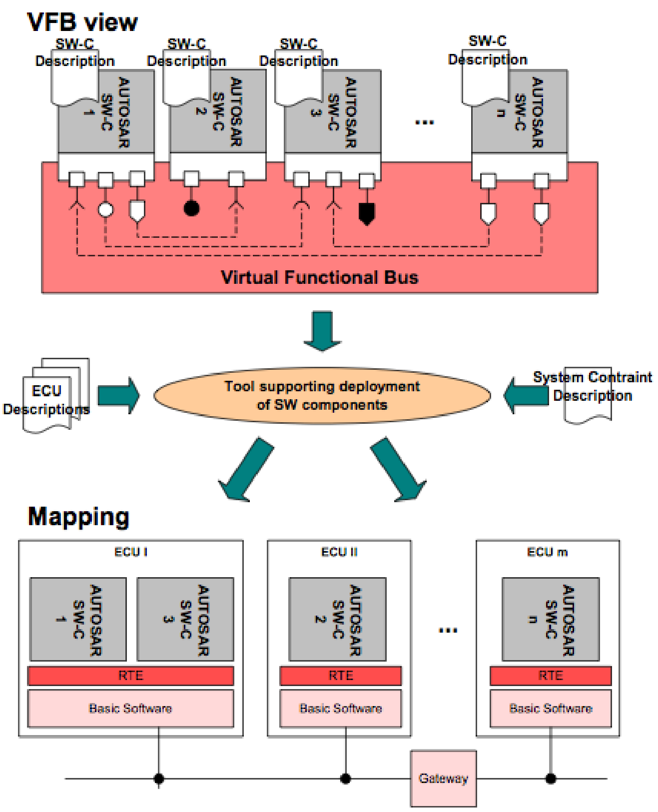
Mapping to heterogeneous multi-processors

- still research topic (**2010**)
- DOL tool from ETH
  - *automatic selection of computation templates*: processor types can be completely heterogeneous (processors, micro-controllers, DSP, FPGA, etc)
  - *automatic selection of communication techniques*: interconnection schemes like central buses, hierarchical buses, rings, etc
  - *automatic selection of scheduling and arbitration*: DOL design space exploration tools choose rate monotonic, EDF, TDMA or priority based

Architecture fixed/ Auto parallelizing	Fixed Architecture	Architecture to be designed
Starting from a given model	HOPES, mapping to CELL proc., Q, Xu, T. Simunic	COOL, DOL SystemCodesigner
Auto-parallelizing	Mnemee, O'Boyle and Franke	Daedalus
	MAPS	

- input of DOL consists of set of tasks together with use cases
- output of DOL describes the execution platform, the mapping of tasks to processors together with task schedules
- => specification graph
  - *allocation* □: subset of architecture graph, representing hardware components
  - *binding* □: selected subset of edges between specifications and architecture denoting a relation => *bindings*
  - *schedule* □: assigns start times to node

Autosar (AUTomotive Open System ARchitecture)



- *AUTOSAR SW-C*: software components encapsulate an application which runs on AUTOSAR infrastructure => *SW-C Description*
- *Virtual Functional Bus* (VFB): sum of all communication mechanisms
- *System Constraint and ECU Description*: to integrate AUTOSAR into a network of ECUs
- *Mapping on ECUs*
- *Runtime Envinronment* (RTE): from the view of autosar software component => implements VFB functionality.

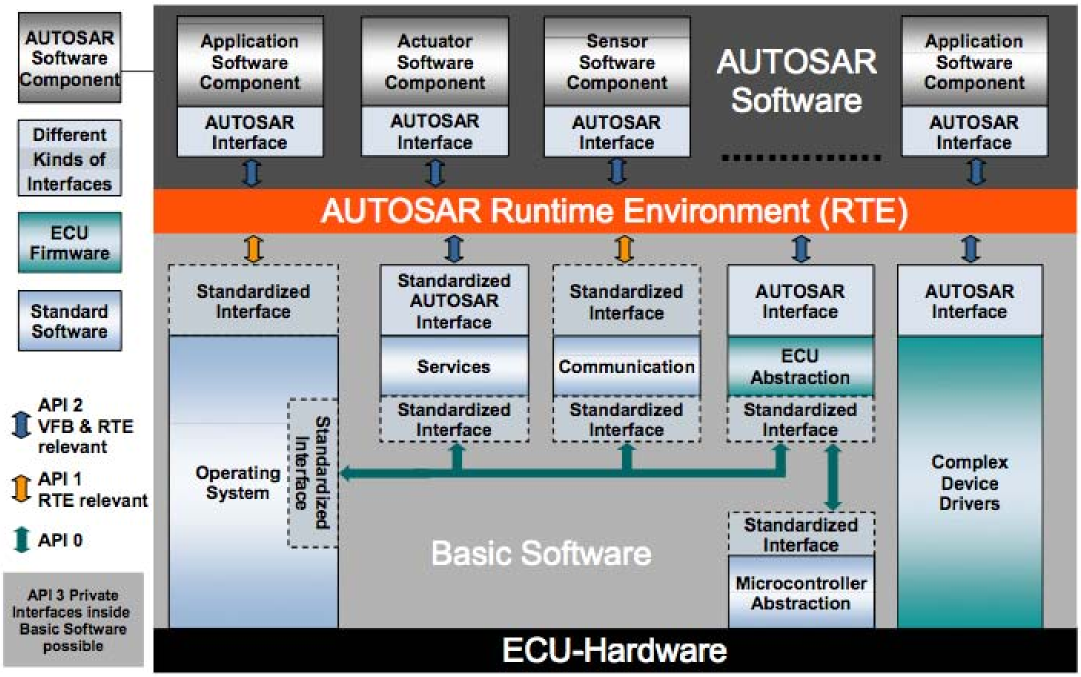
Components

- separation between application and infrastructure
- autosar software component encapsulates the functionality of the application
- autosar software component is an atomic component => cannot be distributed over several ECU
- autosar software: complete formal description of software components
- *autosar software component description*
  - operations and data elements the component provides
  - the requirements of the component for the infrastructure
  - the resources needed by component (CPU, memory, etc)
  - information on specific implementation of component
  - => software component template
- software component implementation is independent from
  - type of micro-controller of ECU
  - type of ECU
  - location of other autosar software components
  - number of times a software component is instantiated
- Sensor/ Actuator Software Components
  - encapsulate dependency on actuators, sensors
- generic "Component"
  - logical interconnection between components may be modeled as component => composition

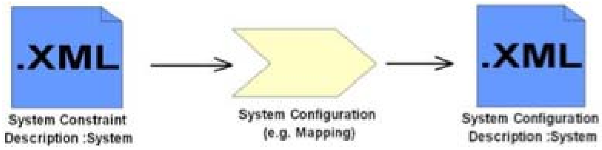
VFB

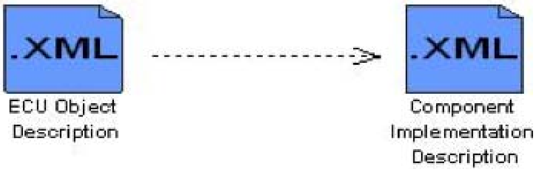
- components implemented independently from underlying hardware
- VFB is abstraction of global communication in vehicle
- *autosar interface*: client/ server or sender/ receiver
- pport (provides), rport (requires)
- *client/ server*: client initializes, + request a service, server responds
  - client can be blocked or not blocked
- *sender/ receiver*: solution for asynchronous distribution of information
  - sender is not blocked
  - sender does not know the identity of receivers

AUTOSAR ECU Software Architecture



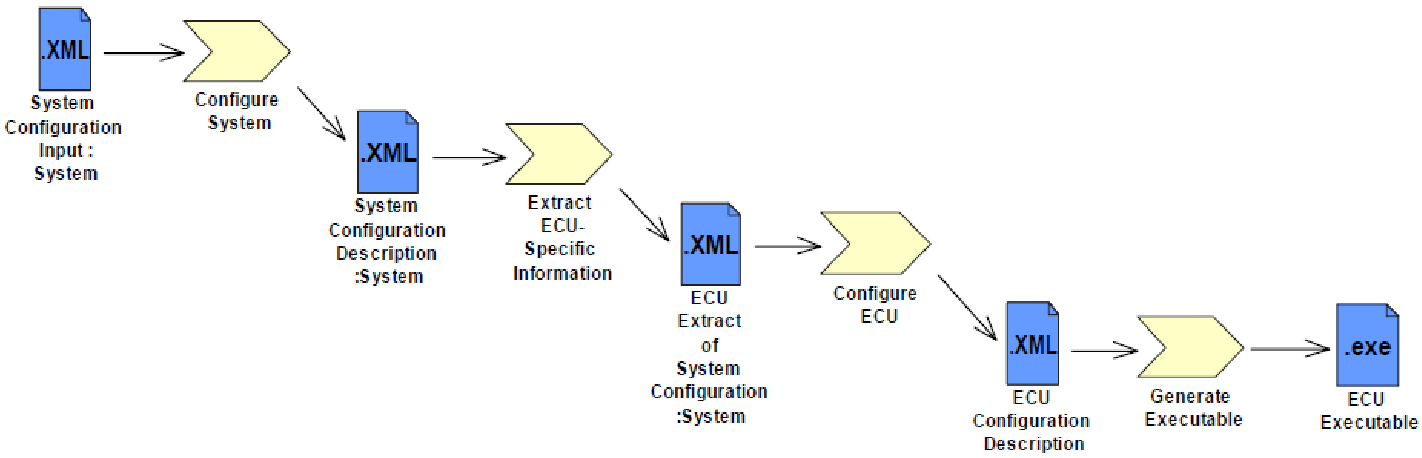
AUTOSAR methodology

- uses the *Software Process Engineering Meta-model* (SPEM) by OMG
- *Work-Product*: Piece of information or physical entity produced by an activity
- *Activity*: describes a piece of work performed by one or a group of persons
- *Guidance*: associated with activities and represent additional information or tools
- *Flow of Work-Products*: 

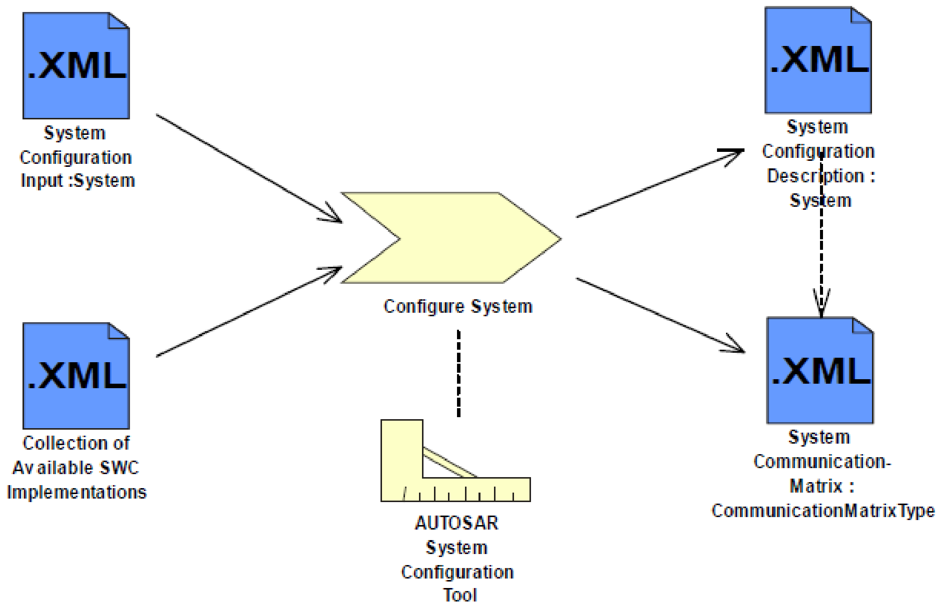
from source to destination, states in-/output
- *Dependency*: 

direction clarifies, who depends onf whom

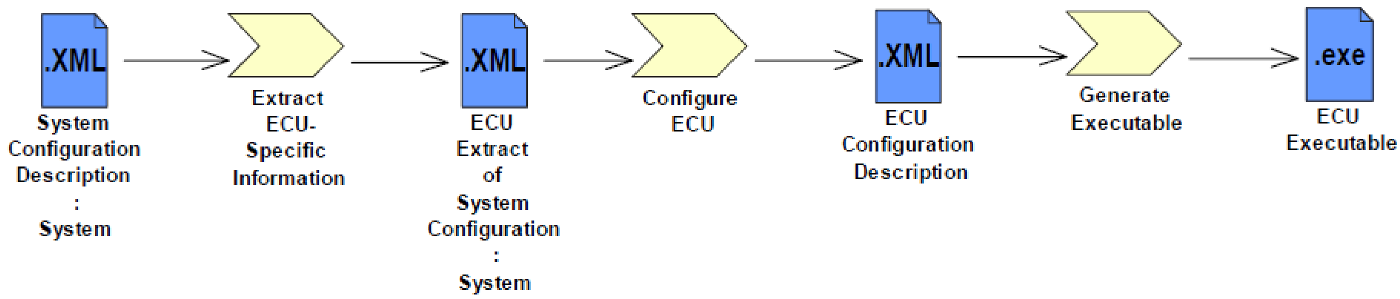
Methodology Overview



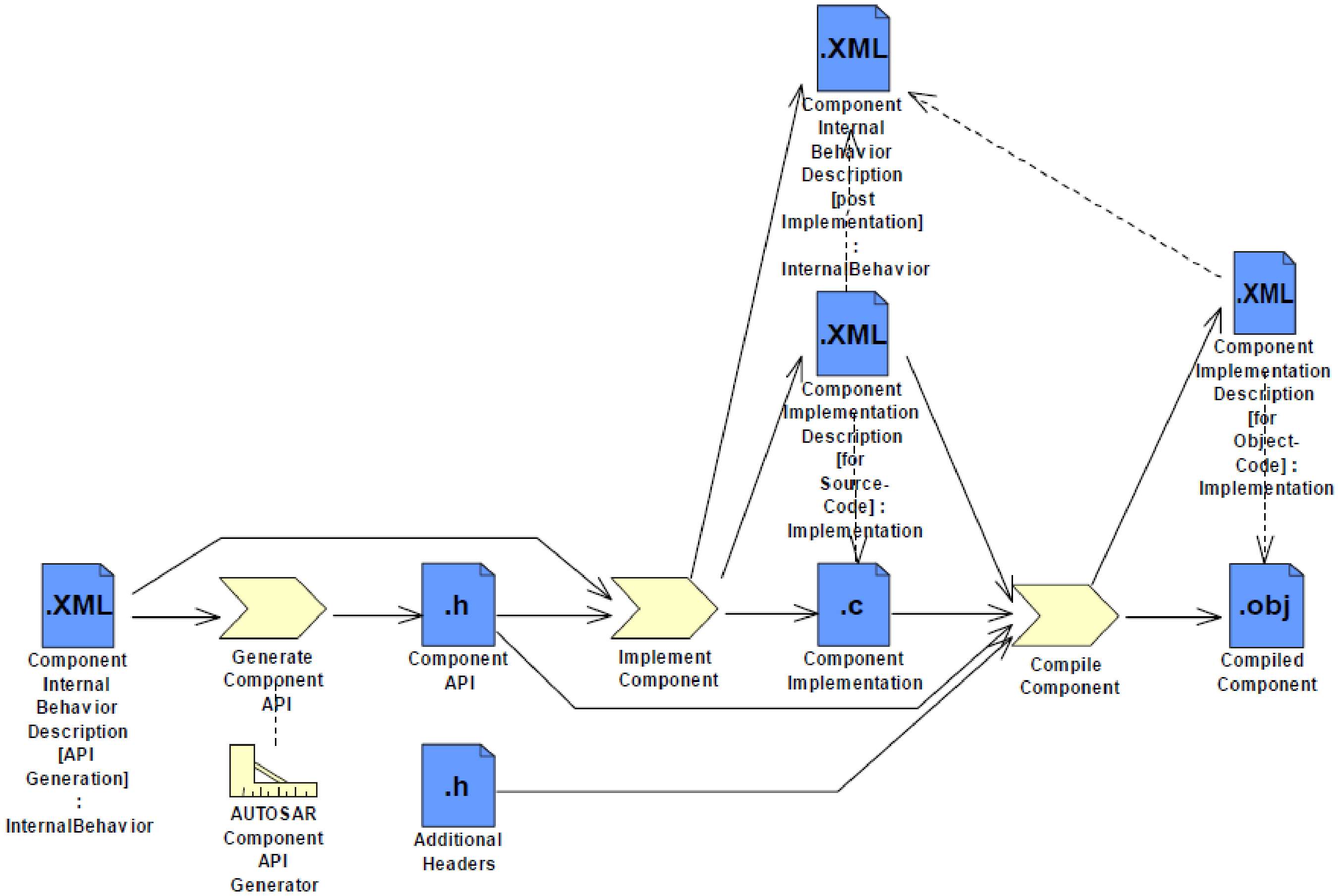
System Configuration



ECU Design and Configuration Methodology

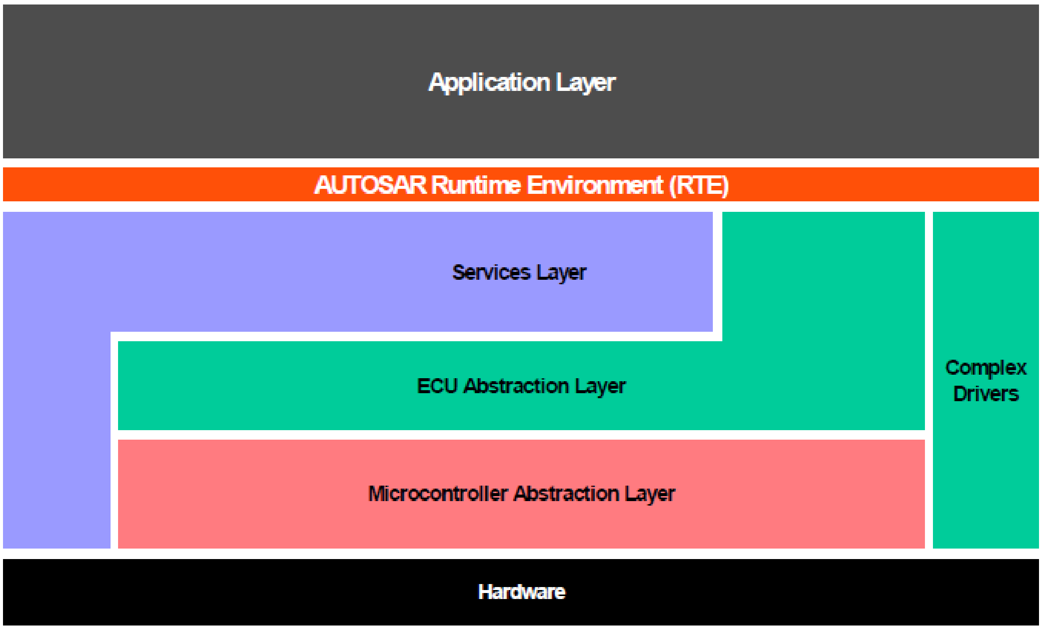


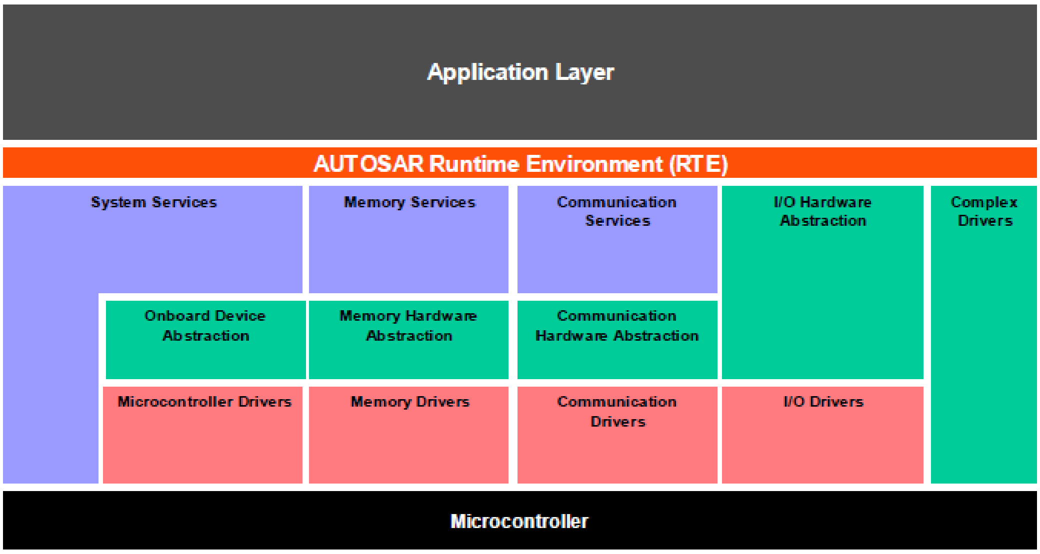
Component Implementation



Detailed ECU Architecture

Layered Architecture





The Runtime Environment

- provide uniform environment, make implementation of software components independent from communication mechanisms and channels

Complex Device Driver

- loosely coupled container, where specific software implementations can be placed