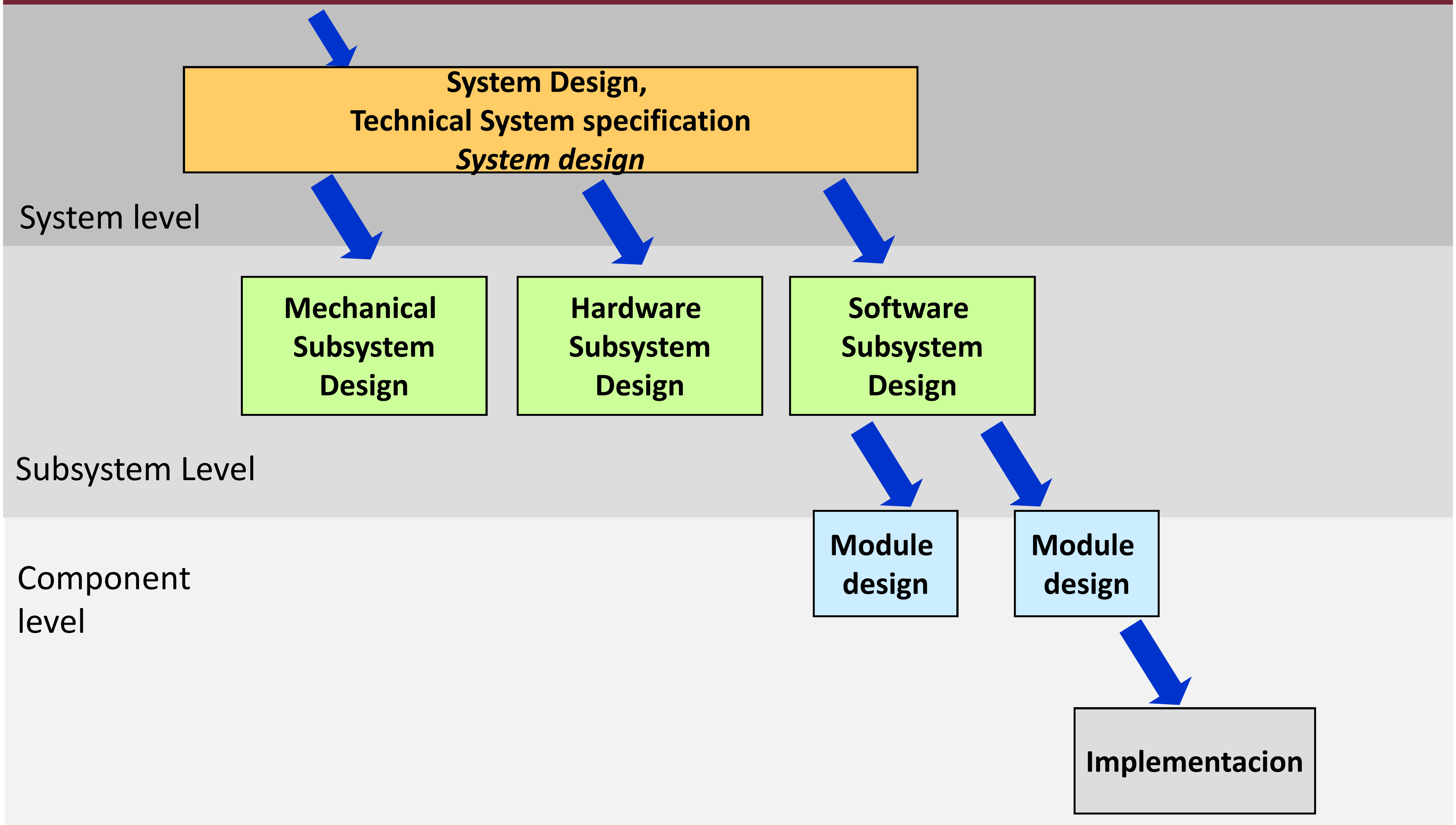
Software Design Path

VIMIMA11 Design and integration of embedded systems

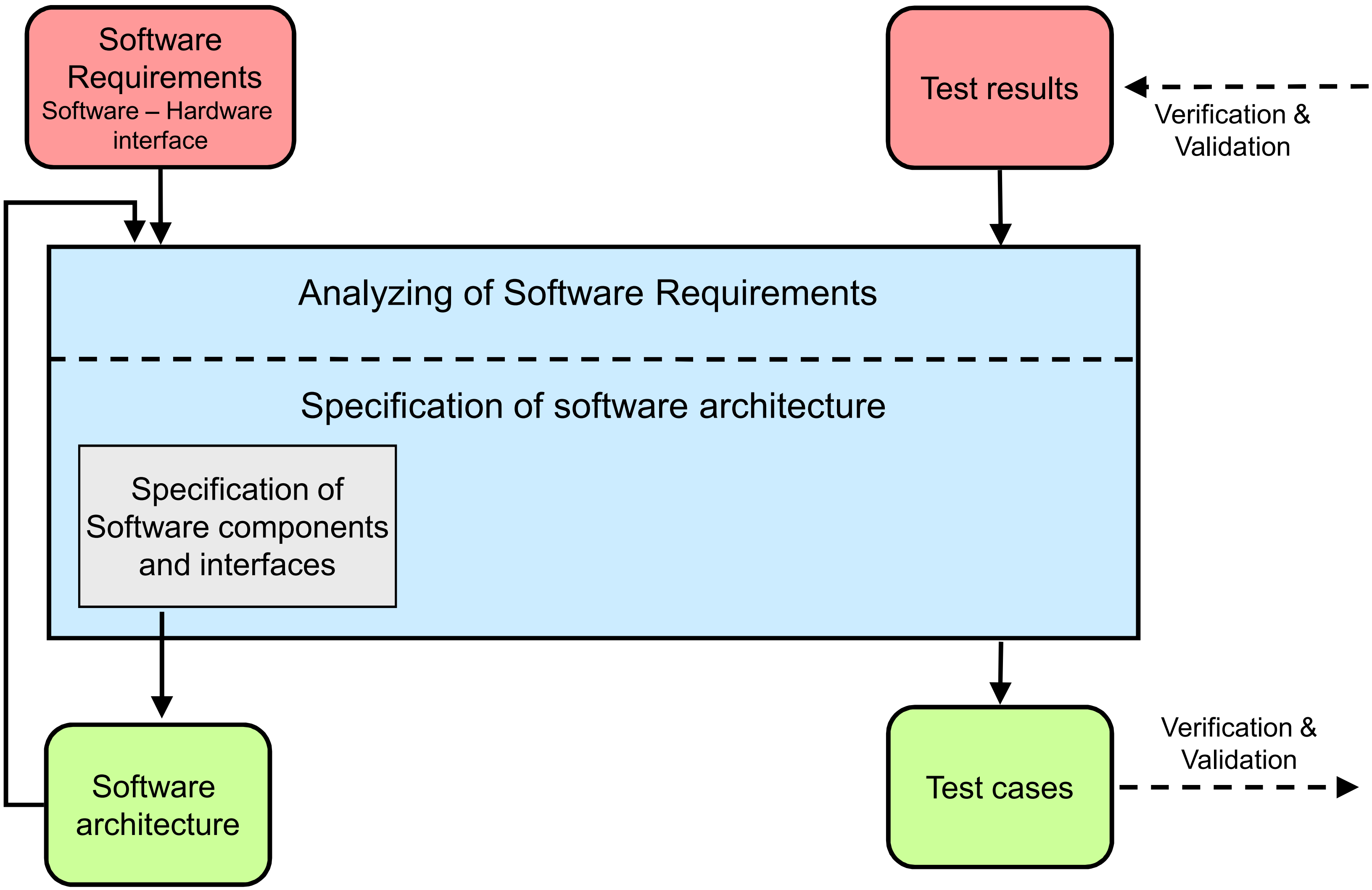
Balázs Scherer

Branching to subsystem paths – Electronics *(Hardware – Software)* path

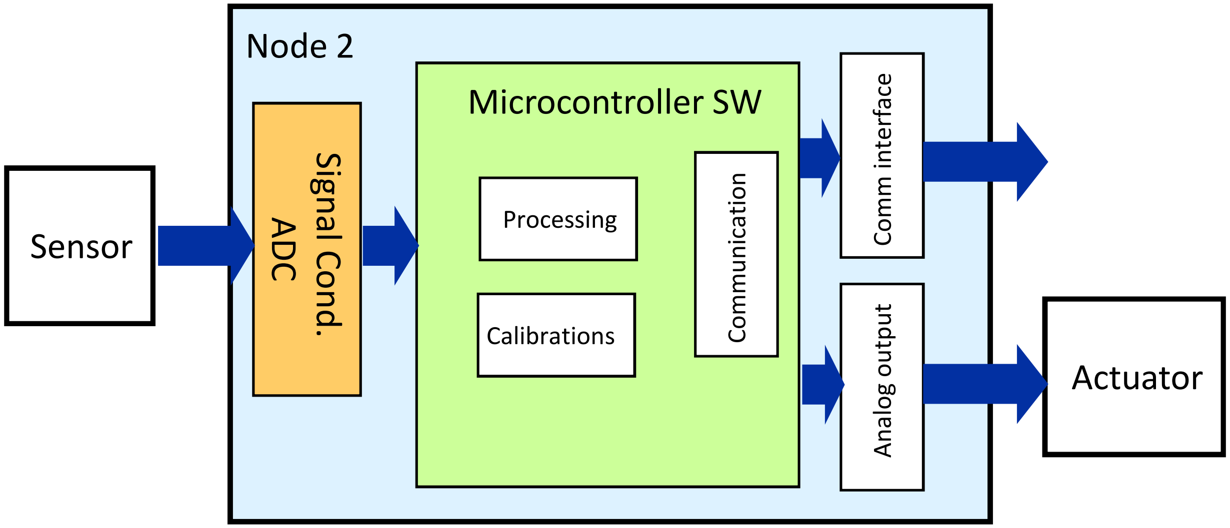


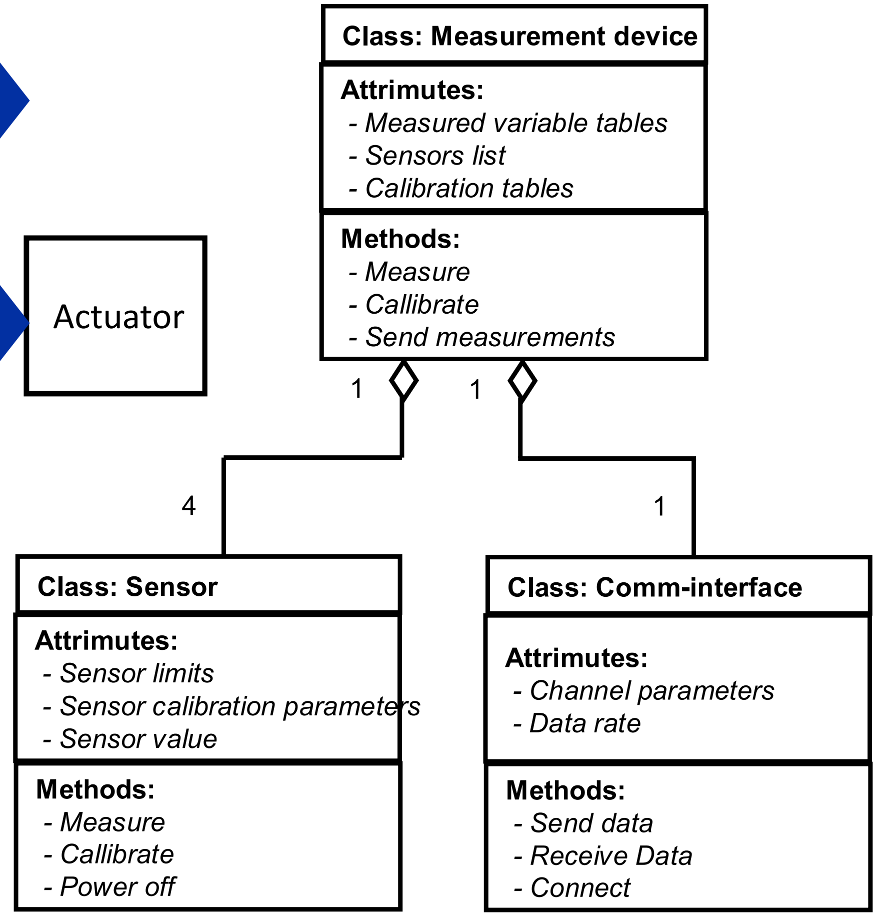
MŰEGYETEM 1782

Software architecture

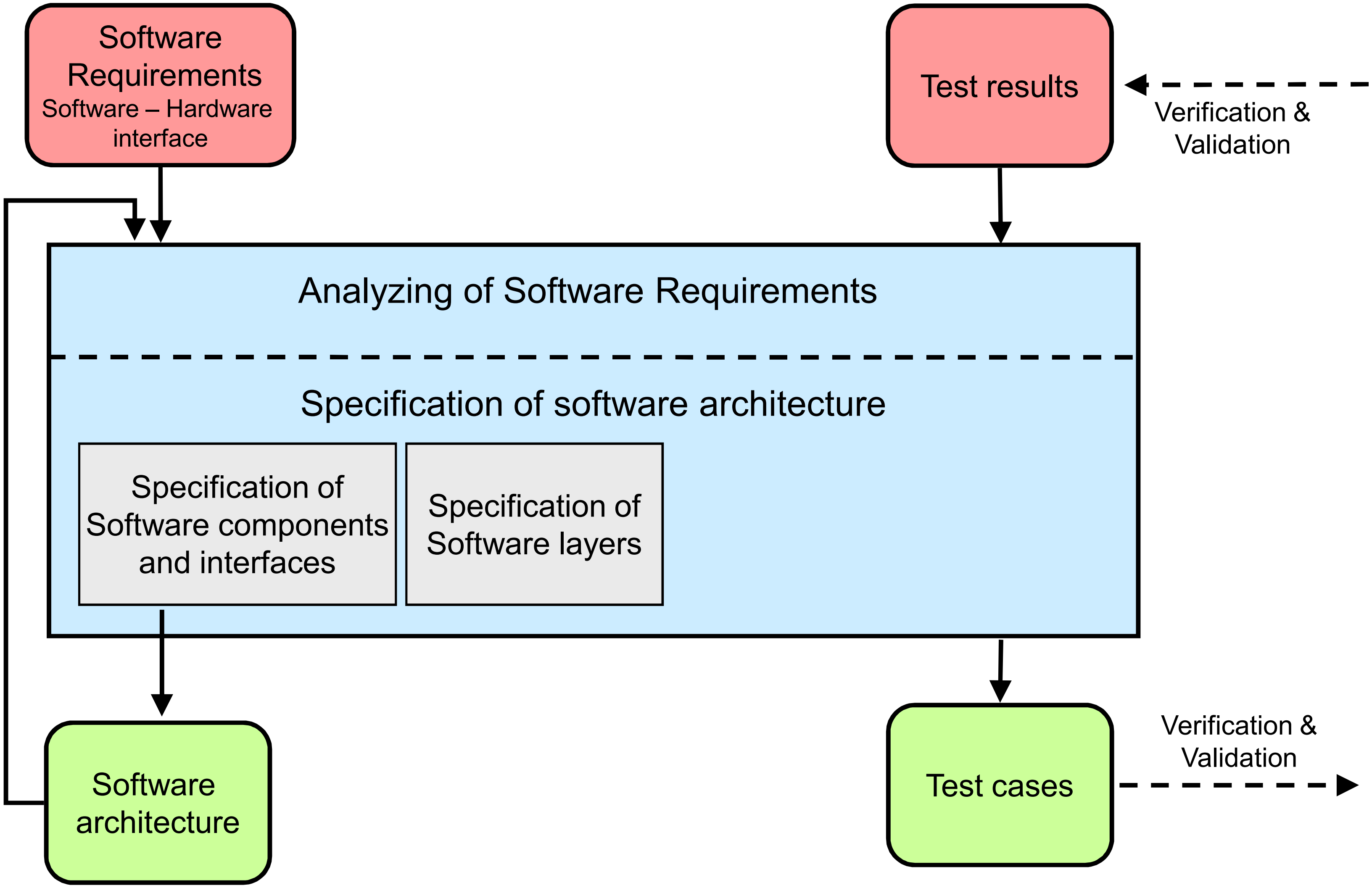


Determining software components and interfaces

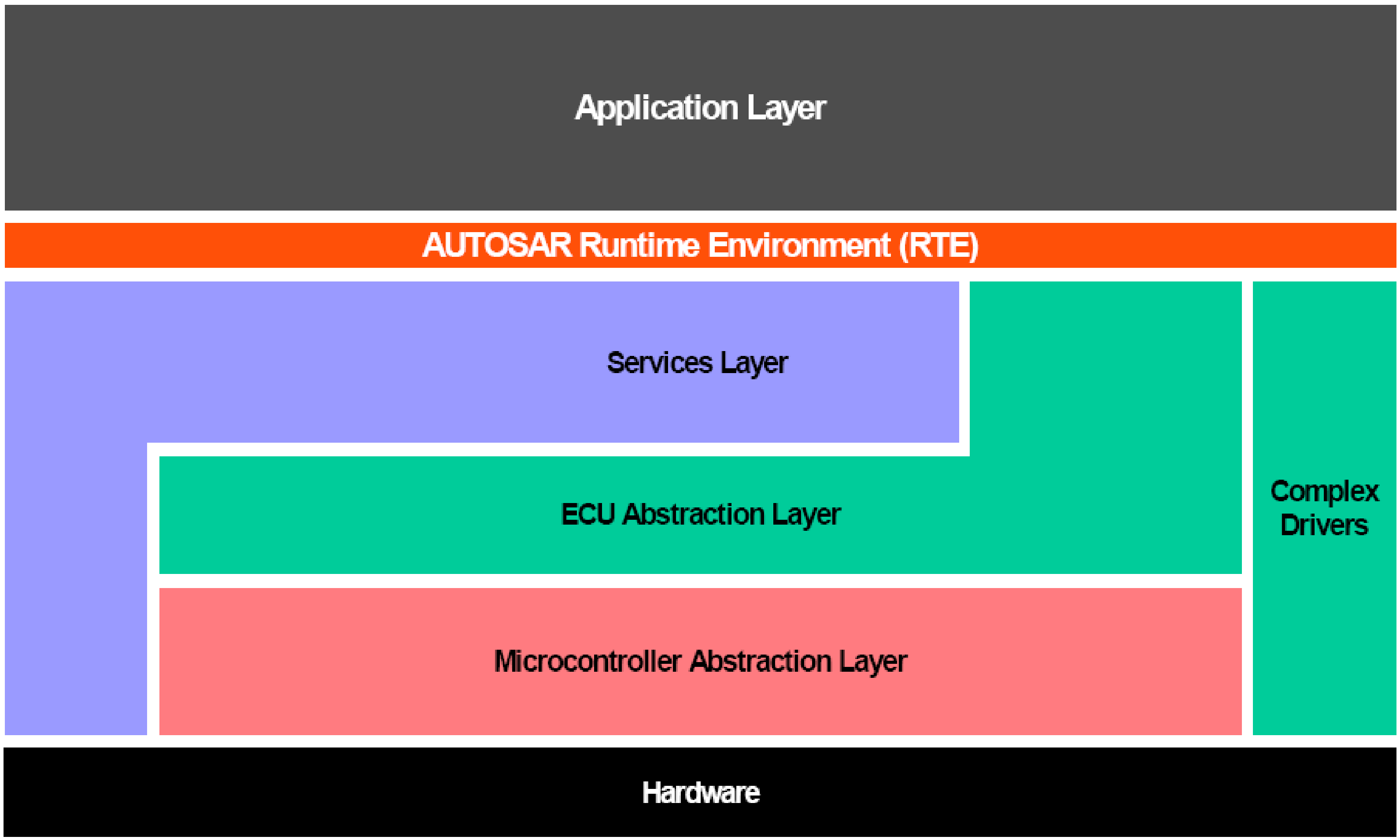




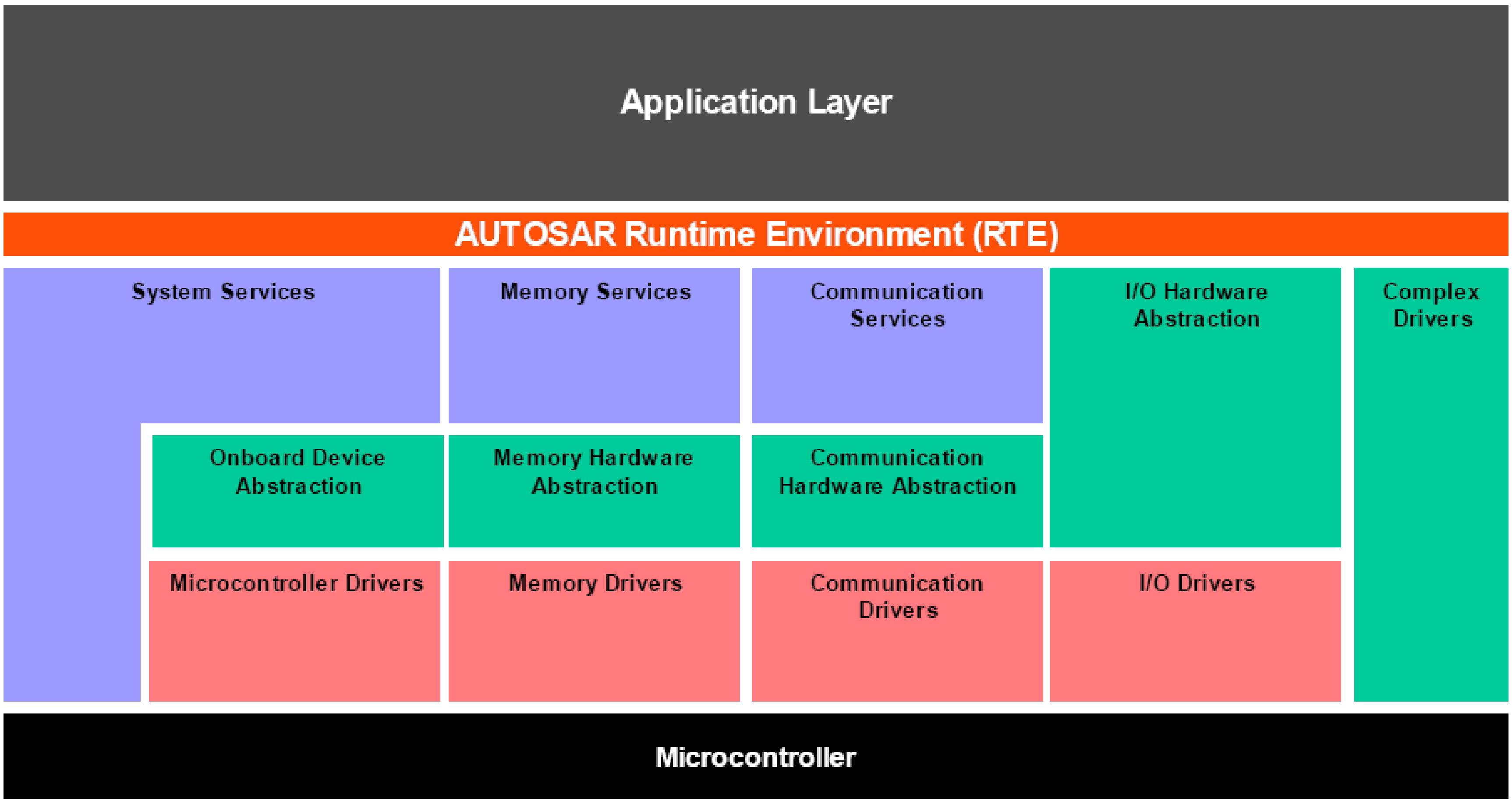
Software architecture



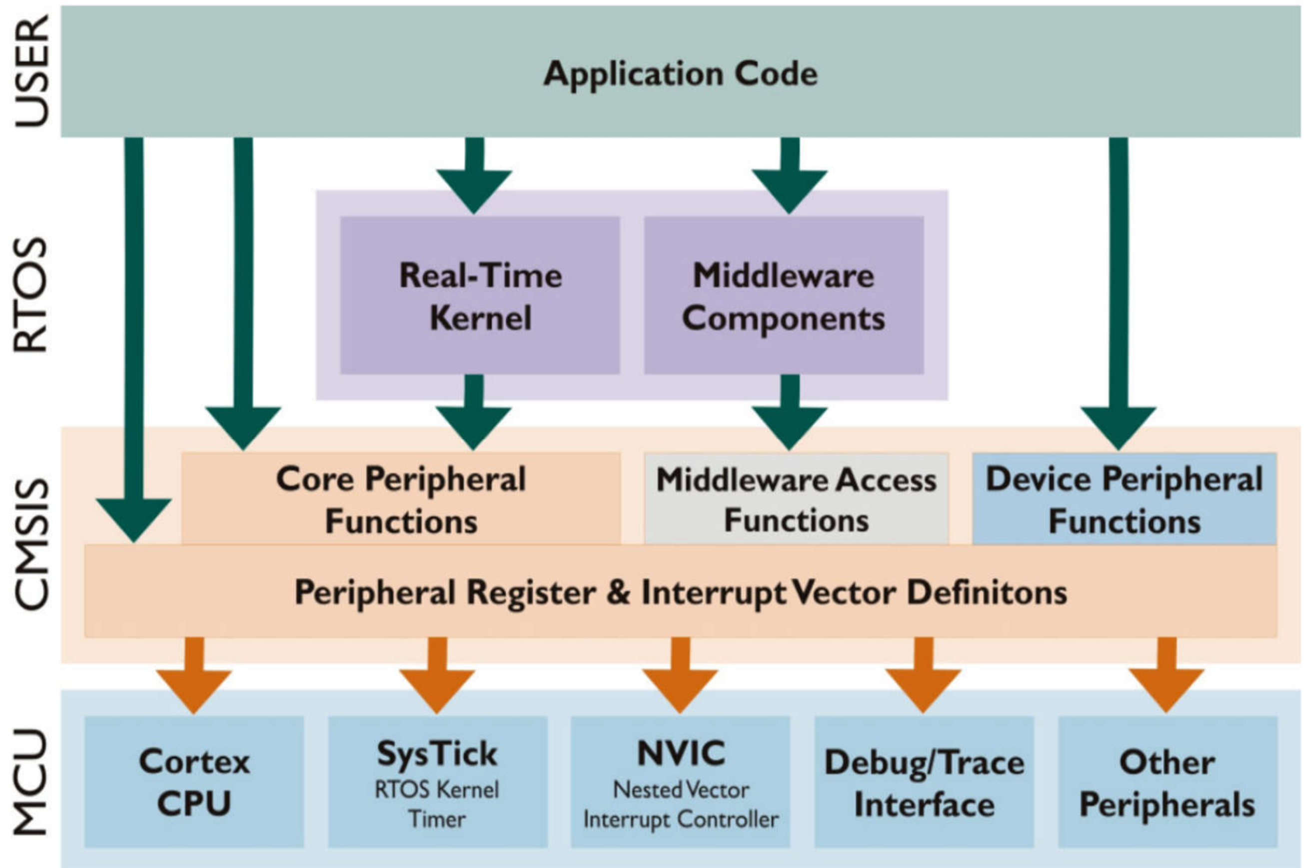
Layered software architecture example AUTOSAR



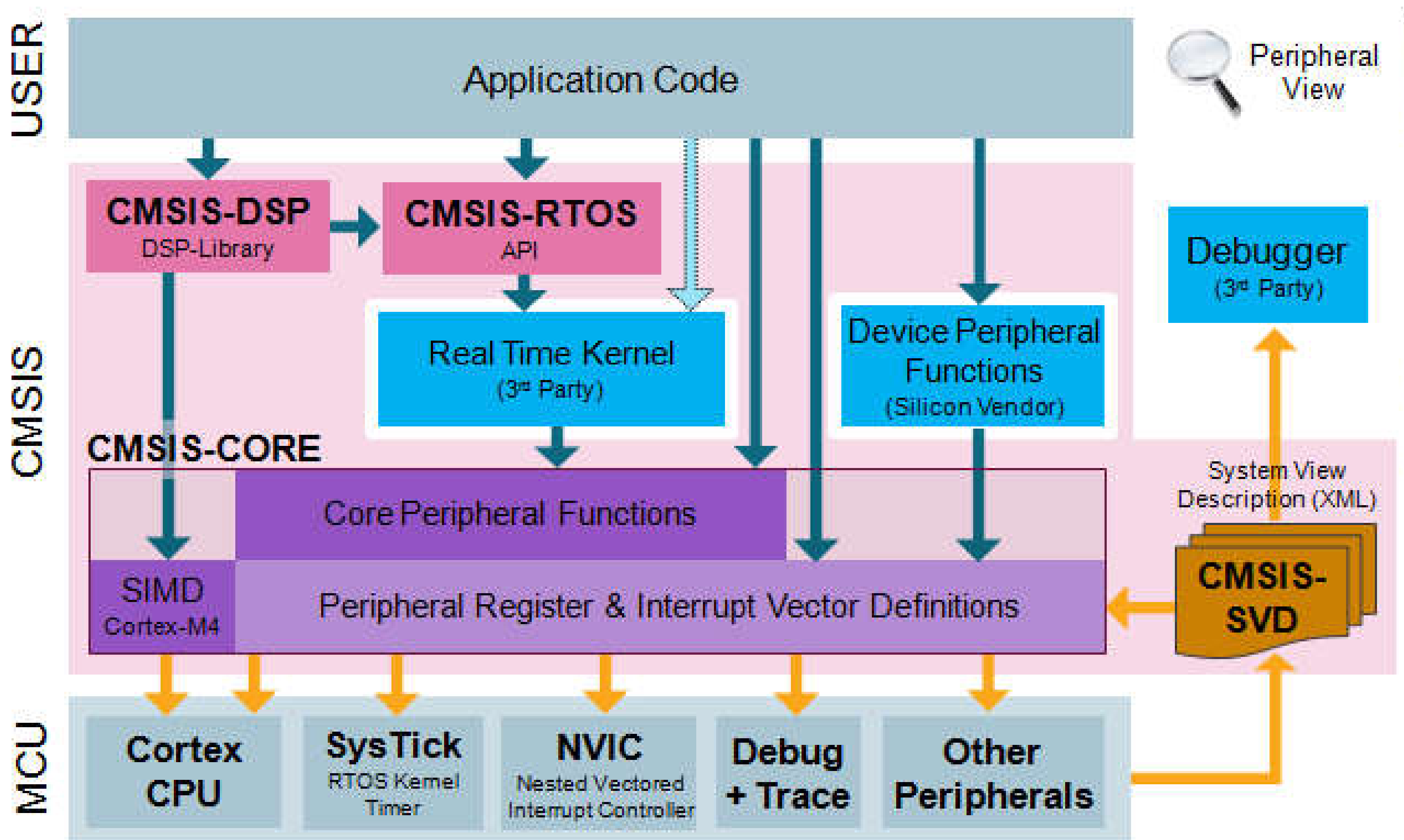
Layered software architecture example AUTOSAR



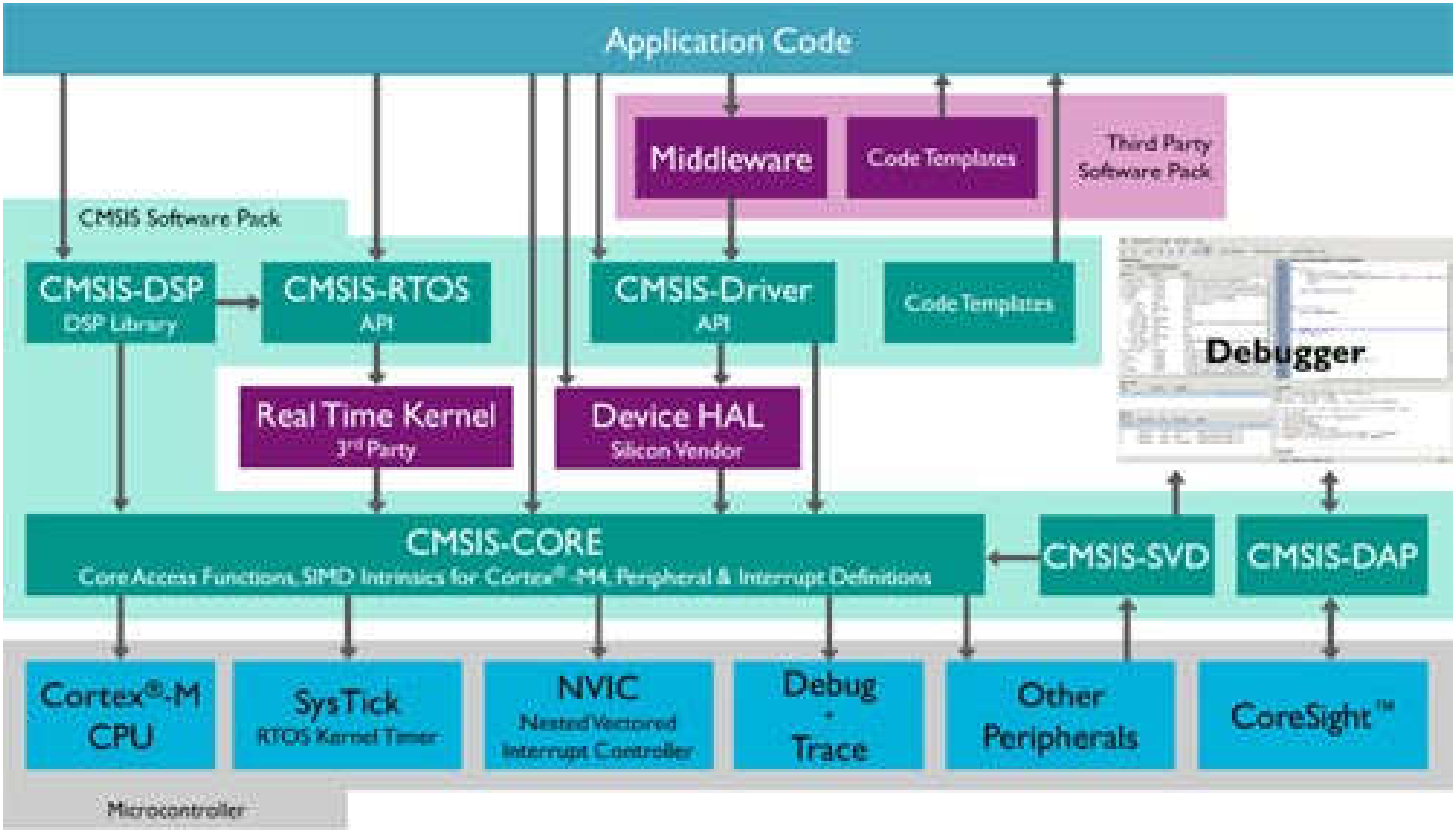
Layered software architecture example CMSIS (v1.3)



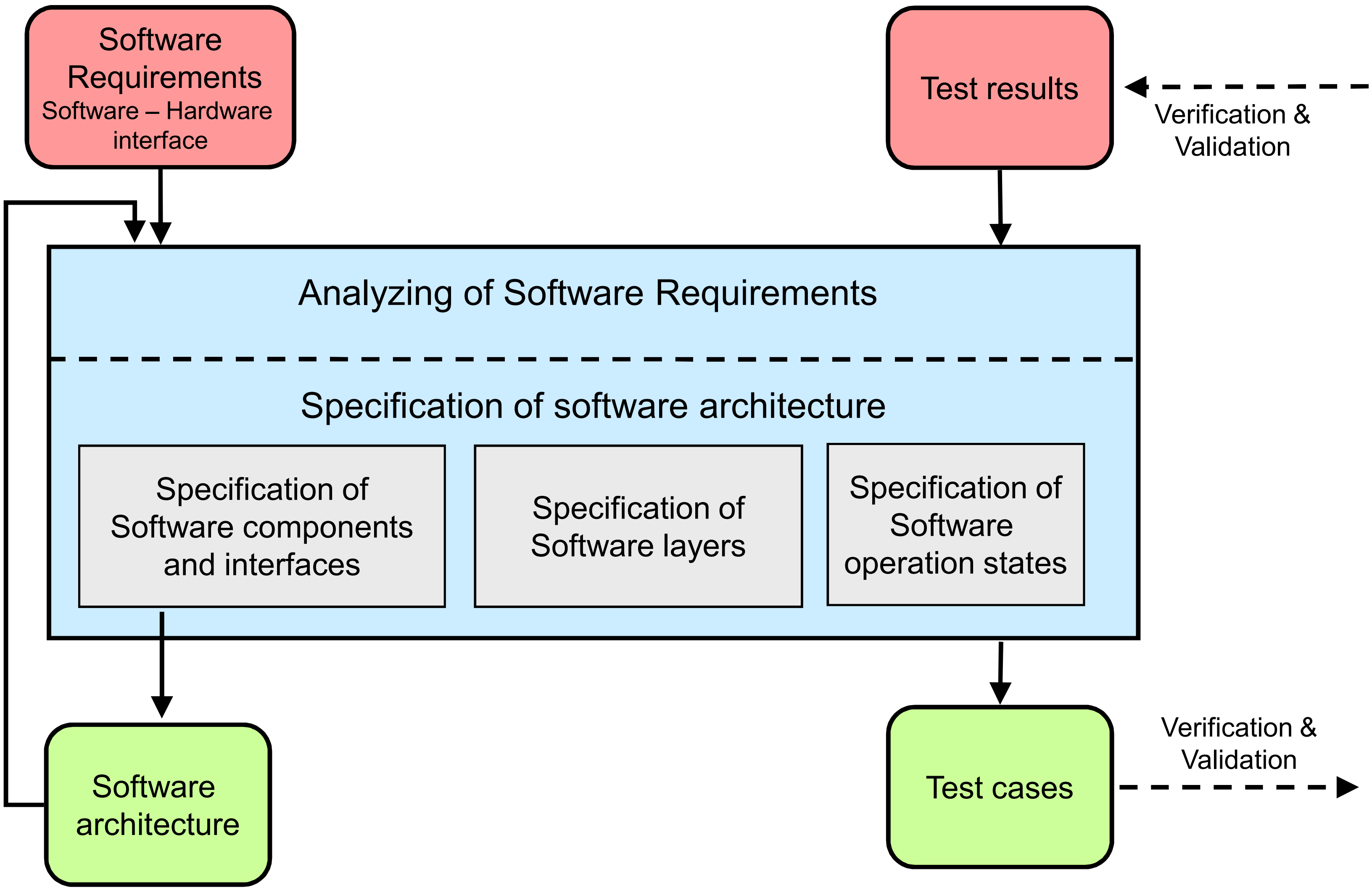
Layered software architecture example CMSIS (v3)



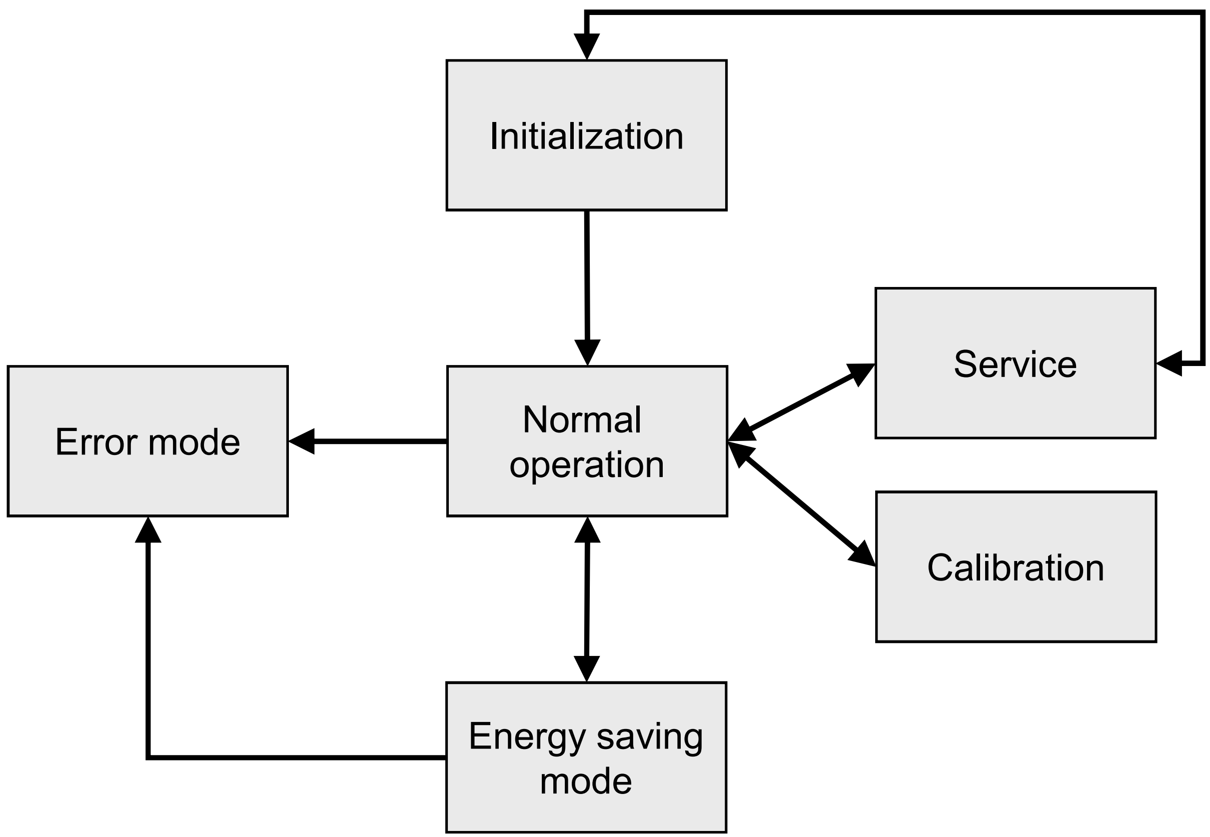
Layered software architecture example CMSIS (v4)



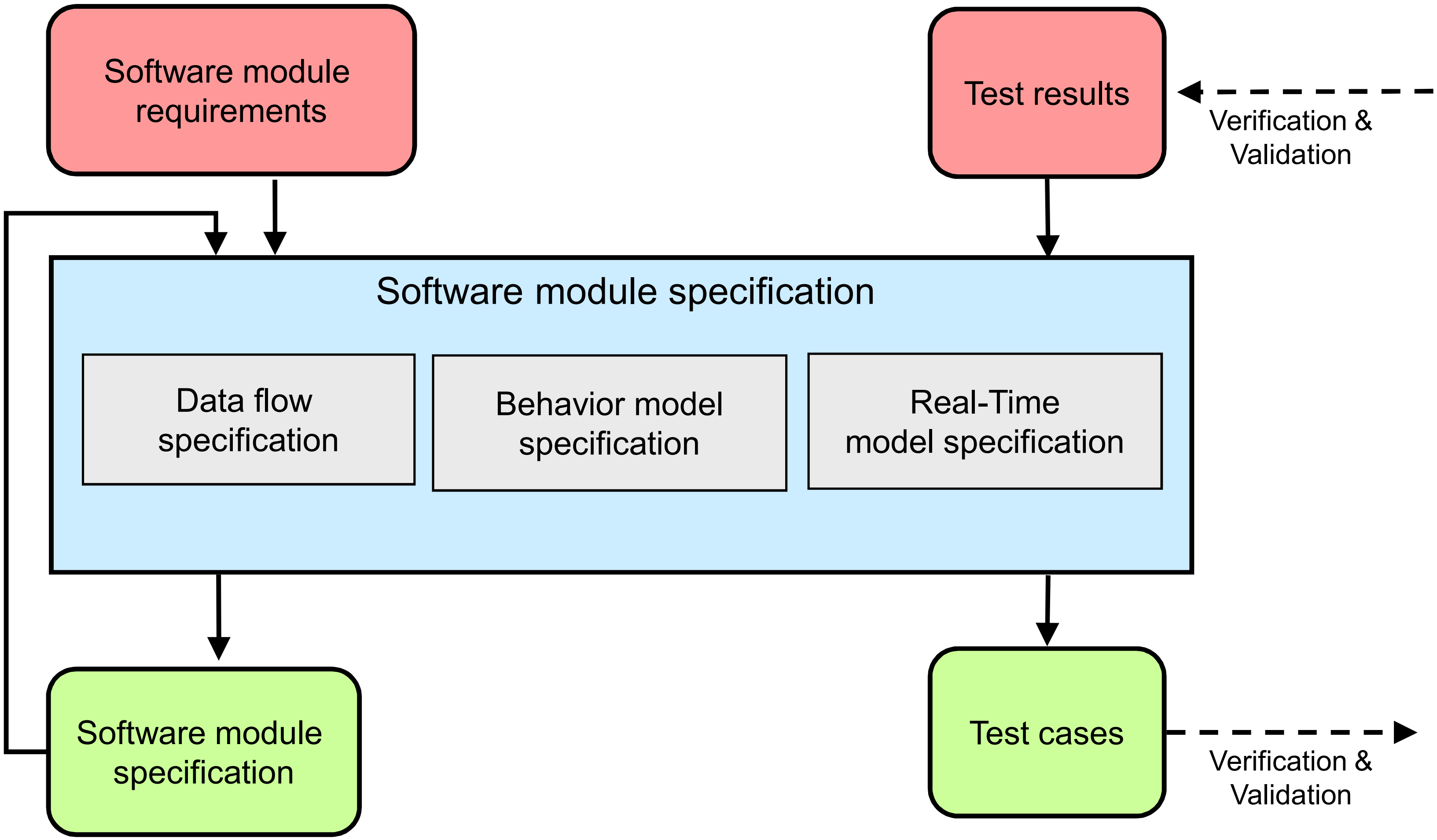
Software architecture



Software operation states

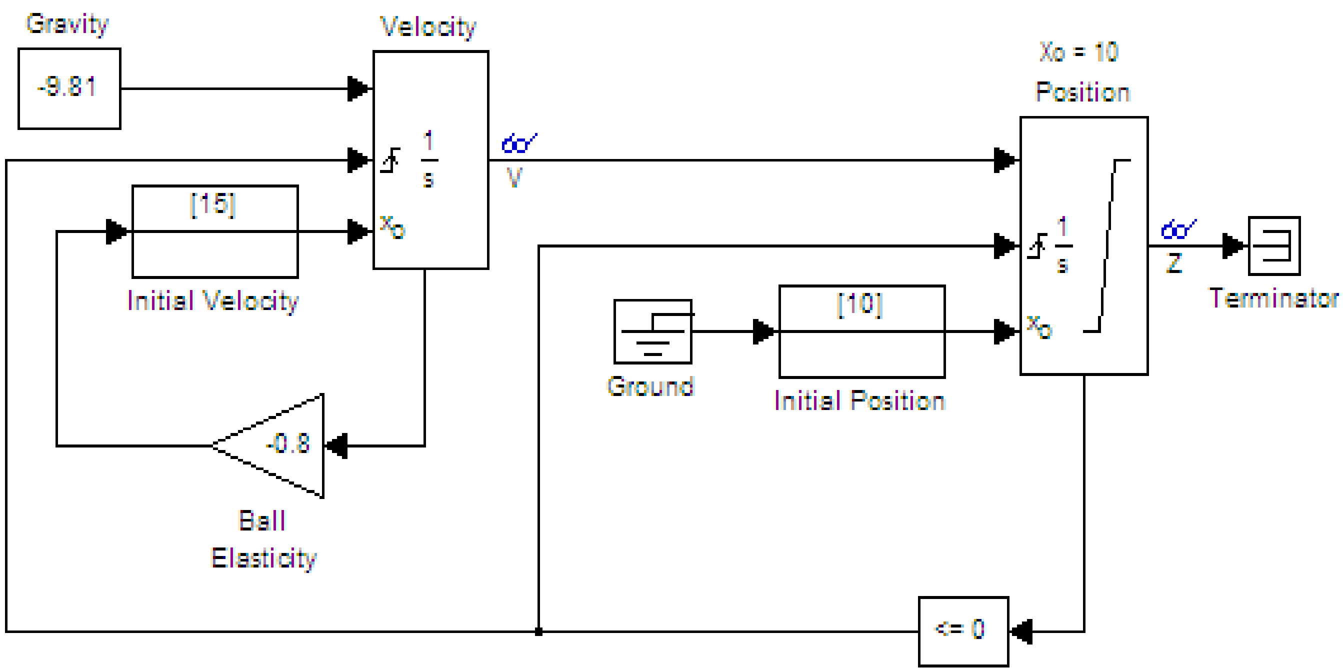


Software module design



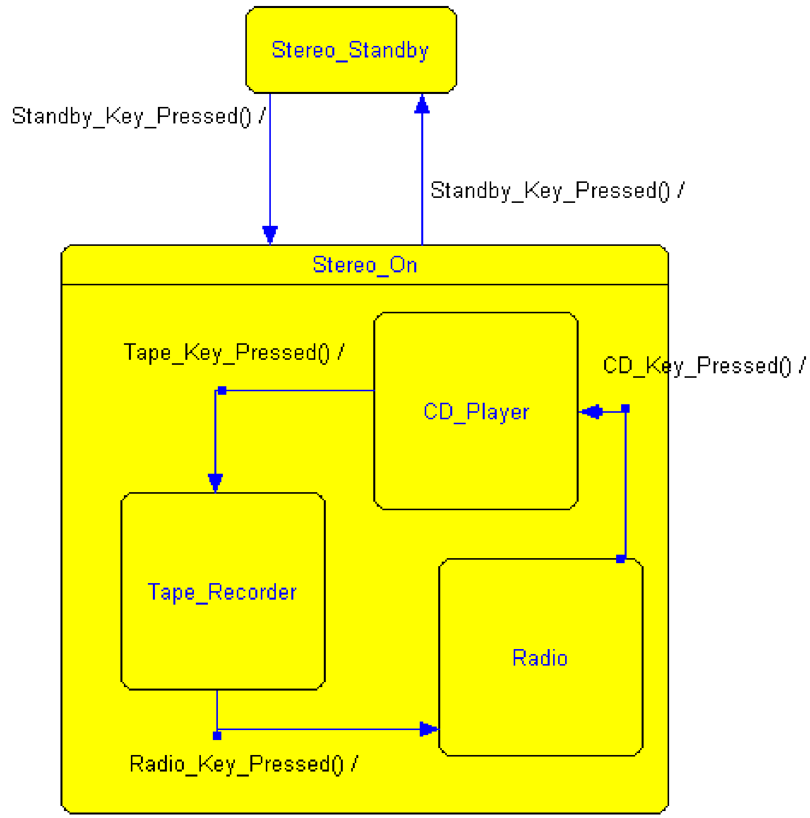
Specification of the data flow, and data model

- Most of the cases there are domain specific language for this
  - Simulink
  - *ASCET*



Behavior model specification

- Most of the cases some State machine description

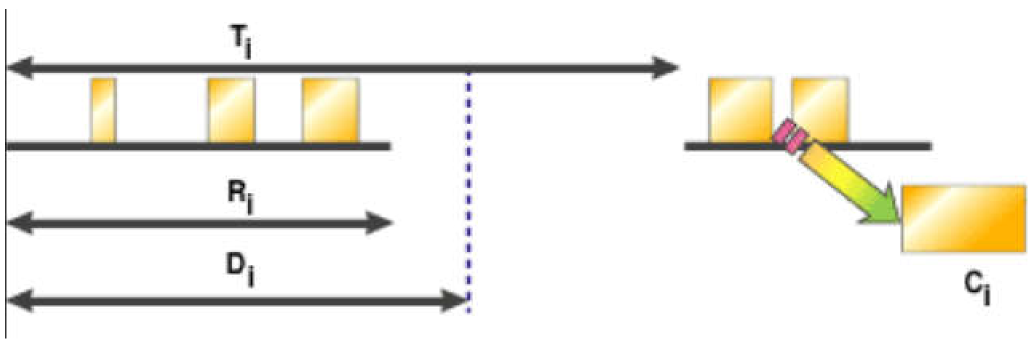


Specification of real time model

- Typically tasks with fix period time: 2.5ms, 5ms, 10ms …

Specification of the Real-Time behavior

- Typically tasks with fix period time: 2.5ms, 5ms, 10ms …
- **DMA:** Deadline Monotonic analysis



$$R_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Table 1 DMA example

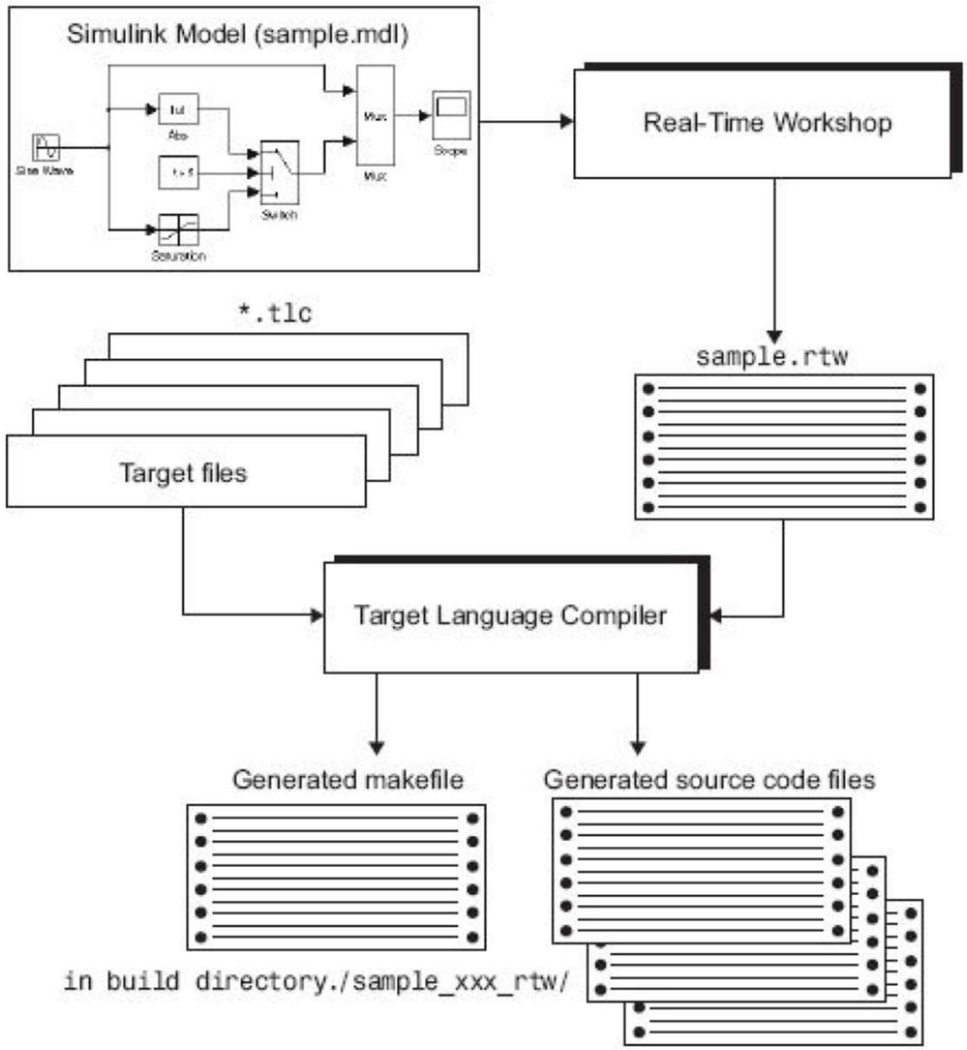
Task	T	C	D
1	250ms	5ms	10ms
2	10ms	2ms	10ms
3	330ms	25ms	50ms
4	1000ms	29ms	1000ms

Table 2 Calculation of worst-case Task 3 response time

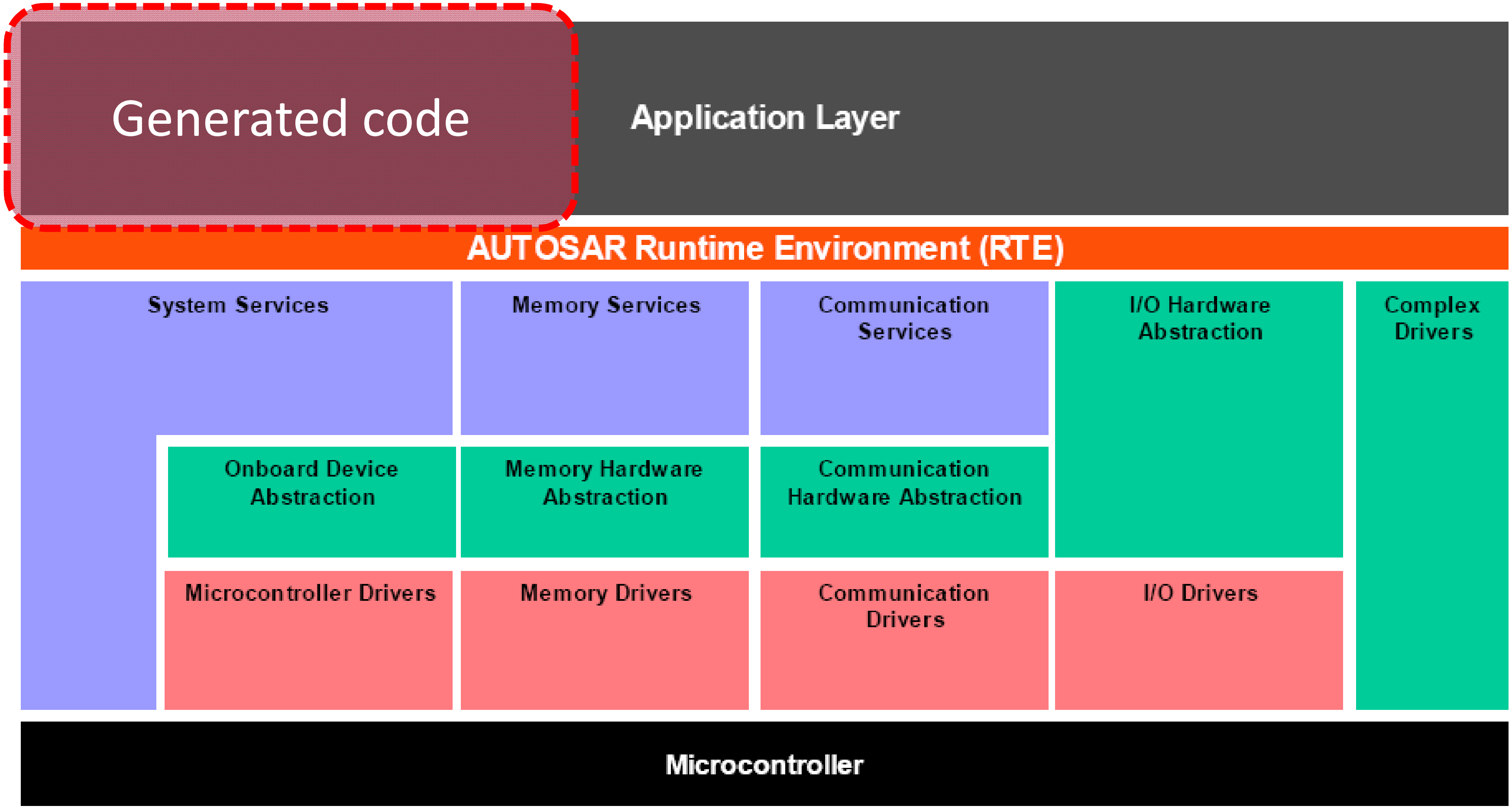
Step	R^n	I	R^{n+1}
1	0	0	25
2	25	5+3x2=11	36
3	36	5+4x2=13	38
4	38	5+4x2=13	38

Model based code generation

Simulink Real-Time Workshop



Application area of generated code



Implementing

Some issues related to implementation

- Problems caused by hardware limitations:
  - Fix, or floating point representation
  - Online calculation or lookup table
  - Problems arising due to floating point calculations

- Considering hardware architecture
  - Special hardware dependent peripherals: DMA
  - Cache an its behavior
  - Tightly-coupled memory
  - Internal or external RAM
  - Power safe modes: Sleep levels.

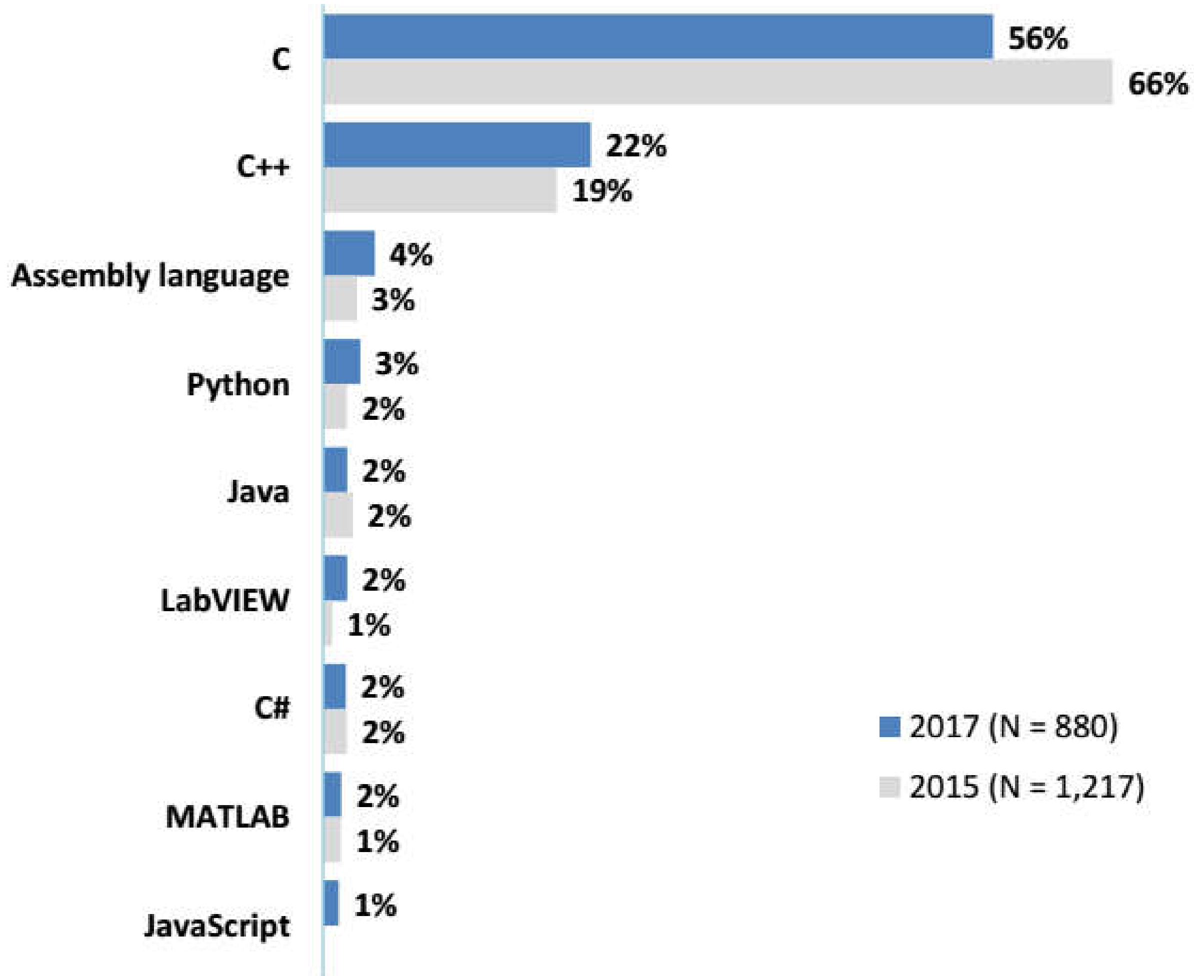
General rules of software implementation

- Many of them are independent of SIL or ASIL level

Topics		ASIL A	ASIL B	ASIL C	ASIL D
1a	Enforcement of low complexity	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++
1c	Enforcement of strong typing[c]	++	++	++	++
1d	Use of defensive implementation techniques	o	+	++	++
1e	Use of established design principles	+	+	+	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++

Program language used for implementation

- Statistics of Embedded Market Study



MISRA-C language subset rules

(Motor Industry Software Reliability Association)

- MISRA-C 1998: The first version. Its goal is to improve the quality of automotive software in the UK (United Kingdom)
- A MISRA-C 1998 version getting widespread and used not only for automotive software
- A MISRA-C 2004: Also approved in the USA (SAE J2632) and Japan
  - Upgrade and clarification of MISRA-1998
  - 121 Mandatory and 20 Advisory rules for C language
- MISRA-C 2012: introduced in 2013 based on C99 standard

Purpose of MISRA-C

- C is a very free language. Programmer can use it a very flexible way
  - Programmers can write syntactically good, or semantically wrong code. Add rules to avoid these situations.
  - Prohibit the use of non unambiguous variable type usage
  - Controlling precedens usage
  - Prohibit the use of non structural programing

Trivial rules

- Comments can not contain code lines
  - Can cause problem, because of embedded comments and the programmers wont know why these lines are uncommented
- Do not modify a cycle variable inside a cycle

```
flag = 1;
for (i = 0; (i<5) && (flag == 1); i++)
{
	flag = 0; /* Can be used to terminate the cycle */
	i = i + 3; /* Can not be used */
}
```

- Using *goto* is prohibited!
- Using *continue* is prohibitied!
- It is prohibited to use bitmanipulation for *signed,* or *floating* types (>>, <<, ~, &, ^)

Non trivial rules

- There are compiler depended behavior. For example divinding to integer number is not unambiguous
    - (-5/3) can be -1 where the remainder is -2
    - (-5/3) can be -2 where the remainder is +1

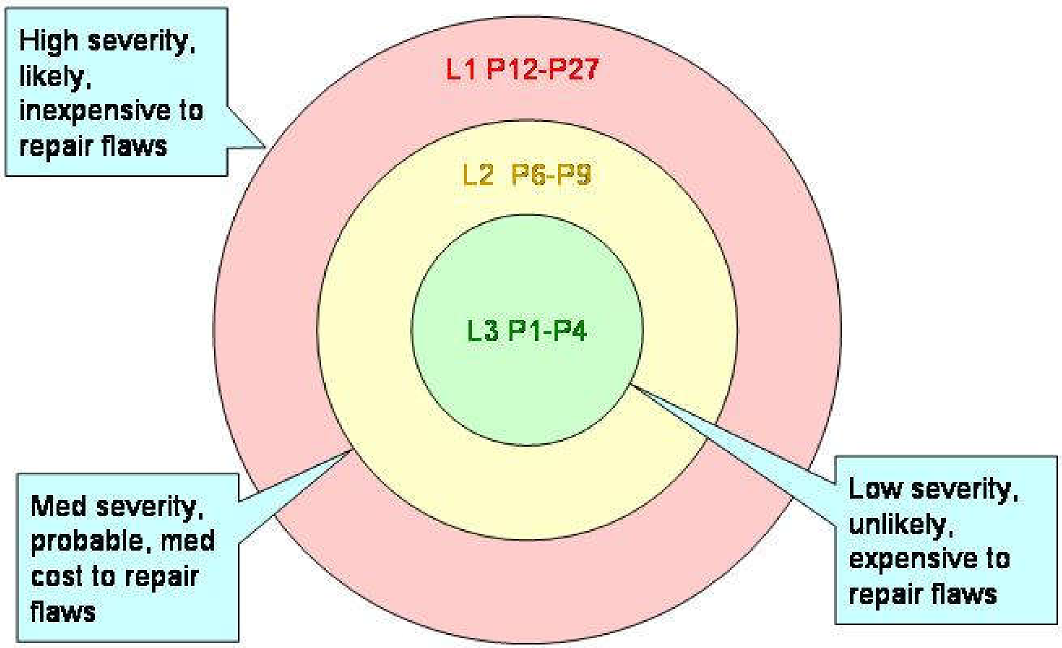
- Type conversions can lead to problems:

```
uint16_t u16a = 40000;
uint16_t u16b = 30000;
uint32_t u32x;

U32x = u16a + u16b; /* u32x = 70000 or 4464? */
```

SEI CERT Coding Standards

- Similar to MISRA-C, but not so embedded specific
- C, C++, JAVA, Perl, Android
- Levels: Severity, Likehood, Remediation cost



Style guides

- The goal of Style guides is to give an uniform view to the software code
  - o Non uniform code make the teamwork harder

- There is no an international standard for this

- Companies has internal coding standards
  - o Structure of C and header files
  - o Variable naming conversions
  - o Control flow styling
  - o Comment styling

Typical C file structure

1. Comment about the name of the file, its purpose, the main author, version, and history. (Some version control systems can handle this headers automaticaly)
2. Header file includes
3. Definitions: Typedefs, Defines, Contants, macros,
4. Global variables: extern, non static, static global
5. Functions: usually in order of usage

Structure of a typical Header file

1. Comment about the name of the file, its purpose, the main author, version, and history. A name of the file cannot be a same as a standard c include name like “math.h”.

2. Header file starting structure

```
#ifndef EXAMPLE_H
#define EXAMPLE_H
...     /* body of example.h file */
#endif /* EXAMPLE_H *
```

3, Do not define variable in header file

Naming notation

- *Hungarian Notatinon* is one of the most widespread
- The system comes from the Hungarian naming logic, the where the family name precede the given name
- This logic is used for the variables. First there is a type or application notation used after that the name
- There are two variant System and Application
  - *System* uses the type of the variable as forename
  - *Application* using the application area or goal as forename
- This notation usually extended with the visibility notation

Examples for Hungarian notation

bBusy: boolean
cApples: count of items
dwLightYears: double word (system)
fBusy: boolean (flag)
nSize: integer (system) vagy count (application)
iSize: integer (system) vagy index (application)

g_nWheels: member of a global namespace, integer
m_nWheels: member of a structure/class, integer
s_wheels: static member of a class
_wheels: local variable

Structure of control flow

brace placement	styles

`while (x == y) {`
`    something();`
`    somethingelse();`
`}`
	K&R and variants:
	1TBS, Stroustrup, Linux kernel, BSD KNF

`while (x == y)`
`{`
`    something();`
`    somethingelse();`
`}`
	Allman

`while (x == y)`
`  {`
`    something();`
`    somethingelse();`
`  }`
	GNU

`while (x == y) {`
`    something();`
`    somethingelse();`
`    }`
	Ratliff

`while (x == y) {`
`    something();`
`    somethingelse(); }`
	Lisp

Automatic formatting tools

- Artistic Style: free to download

--style=allman / --style=bsd / --style=break / -A1

Allman style uses broken brackets.

```
int Foo(bool isBar)
{
    if (isBar)
    {
        bar();
        return 1;
    }
    else
        return 0;
}
```

--style=pico / -A11

```
int Foo(bool isBar)
{   if (isBar)
    {   bar();
        return 1; }
    else
        return 0; }
```

Usual problems about documentation

1. We write the code
2. We make comment for it
3. We make the documentation

4. We modify the code
5. Maybe the comment is modified
6. There is a high probability that the documentation won't be modified

Inconsistent state: code – comment - documentation

Automated documentation generation form comment: Doxygen

- First version 1997
- Intended to solve the comment – documentation inconsistency problem
- Two type of comenting style is suported
- JAVA doc style

```
/**
... text ...
*/
```

- C stlye

```
/*!
... text ...
*/
```

Example for Doxygen commenting

```
/*! \fn void UART1_Init(unsigned long baud_rate, void (*handler)(void));
 *  \brief An inicialisation function to redirect STDIO to UART1
 *  \param baud_rate: Baudrate in bit/sec
 *  \param handler: Callback function for receiving UART characters with IT
 *  \return nothing
*/
```

```
/** @fn void UART1_Init(unsigned long baud_rate, void (*handler)(void));
 *  @brief An inicialisation function to redirect STDIO to UART1
 *  @param baud_rate: Baudrate in bit/sec
 *  @param handler: Callback function for receiving UART characters with IT
 *  @return nothing
*/
```

Creating groups

- Doxygen creates file based documentation. To organize the documentation to function or module style the grouping of these modules are needed.

- Used by the firmware libraries

```
/** @addtogroup my group
@{
*/

......
/**
@}
*/
```

Examples for using Doxygen

```
/** @brief  I2C Init structure definition  */
typedef struct
{
  uint32_t I2C_ClockSpeed;         /*!< Specifies the clock frequency */
  uint16_t I2C_Mode;               /*!< Specifies the I2C mode.
This parameter can be a value of @ref I2C_mode */
} I2C_InitTypeDef;
```

```
/** @defgroup I2C_mode

@{

*/
#define I2C_Mode_MASTER                        1
#define I2C_Mode_SLAVE                         0
/**
@}
*/
```