Introduction to safety-related software engineering: Multitasking timing application design

Frédéric Thomas - @fthomas_fr - frederic.thomas@krono-safe.com - Master

Summary

Chapter 1: Engineering and processus overview

Which are the engineering issues and constraints ?

Chapter 2: Design approches and methodologies

On which criteria should I choose an approach to design my application ?

Chapter 3: Concurrent & Timing Design illustration with the Psy Approach

Could you detail one approach to be more concrete ?

Chapter 4: Implementation key concepts

Do we have specific implementation issues and constraints ?

Practical works

Disclaimers

This support attempts to give an overview of issues, concepts and pocesses to design a safety critical software. It doesn't focus on requirement engineerings, safety analyses and tests which are also important parts of a safety critical development process.

Engineering and processus overview

Which are the engineering issues and constraints ?

Summary

Issue: Development complexity

Characteristics of a simple task	**Characteristics of a complex task, e.g. a multitasking timing application development**
Static: The properties of the task do not change	Dynamic: The properties of the task are time dependant
Discrete: System variable can only take values from discrete sets	Continuous: System variable domain is continuous
Separable: Independant task	Non-separable: Highly interactive tasks that can not be isolated
Sequential: System behavior can be understood by a sequential step by step analysis	Simultaneous: Concurrent processes interact in generating visible behavior. Step by step is difficult.
Homogenous: components, principles and rules are alike	Heterogeneous: different components, principles and rules
Linear Functional relationship	Non-Linear Functional relationship
Context independance explanatory	Conditinal and dependent context explanatory
Regularity of principles and rules	Irregular context dependent rules
Surface: Important principles and rules are apparent by looking at observable properties	Deep: Important principles and rules are covered and abstract and not detectable

Adapted from "Keeping it simple: How the reductive tendency affects cognitive engineering (2004) by P J Feltovich, R R Hoffman, D D Woods, A Roesler"

Issue: Robust and reliable software development

- Robust software development is critical and risk is increased when:
  1. Designers/Developers shall manage hardware complexity
  2. Specifiers/Designers/Developers shall manage application complexity (end to end constraints, memory partition)
  3. Specifiers/Designers/Developers shall ensure spatial and timing partitionning
  4. Specifiers/Designers/Developers shall manage configure/implement safe mechanisms
  5. Testers shall check/validate such systems
  6. Specifiers/Designers/Developers/Testers shall work with time-to-market constraints
  7. Specifiers/Designers/Developers shall developt the right system
  8. Specifiers/Designers/Developers shall developt a safety system

Issue: Confidence (1/2)

- This confidence is built on:
  1. Determinism for the **features** and for the **tests**
     1. Real-time system behavior is always unique and invariant
     2. Different implementations have identical results (except for the sizing)
     3. Test scenarios are predictable and their results reproducible
     4. A Causal chain between cause and the consequence effect can be established without a doubt (timing partitionning)
     5. The consequence of failures shall be deterministic
     6. Faults are confined, detected and their impacts are mastered
     7. New fault detection and confining deterministic mechanisms

Issue: Confidence (2/2)

- This confidence is built on:
  1. Spatial and timing partitionning
  2. Fault tolerant approach: Assist to built correct & safe application
  3. Mastering execution: Safe & efficient resource sharing
  4. Diversification
  5. Relevant verification & validation: Ensure behavioral determinism
  6. In-depth defense
  7. Requirements, tests and codes are traced
  8. Development methods are defined and applied

Issue: Confidence

- This confidence is built on processus reliability and processus management: quality assurance (Certification)

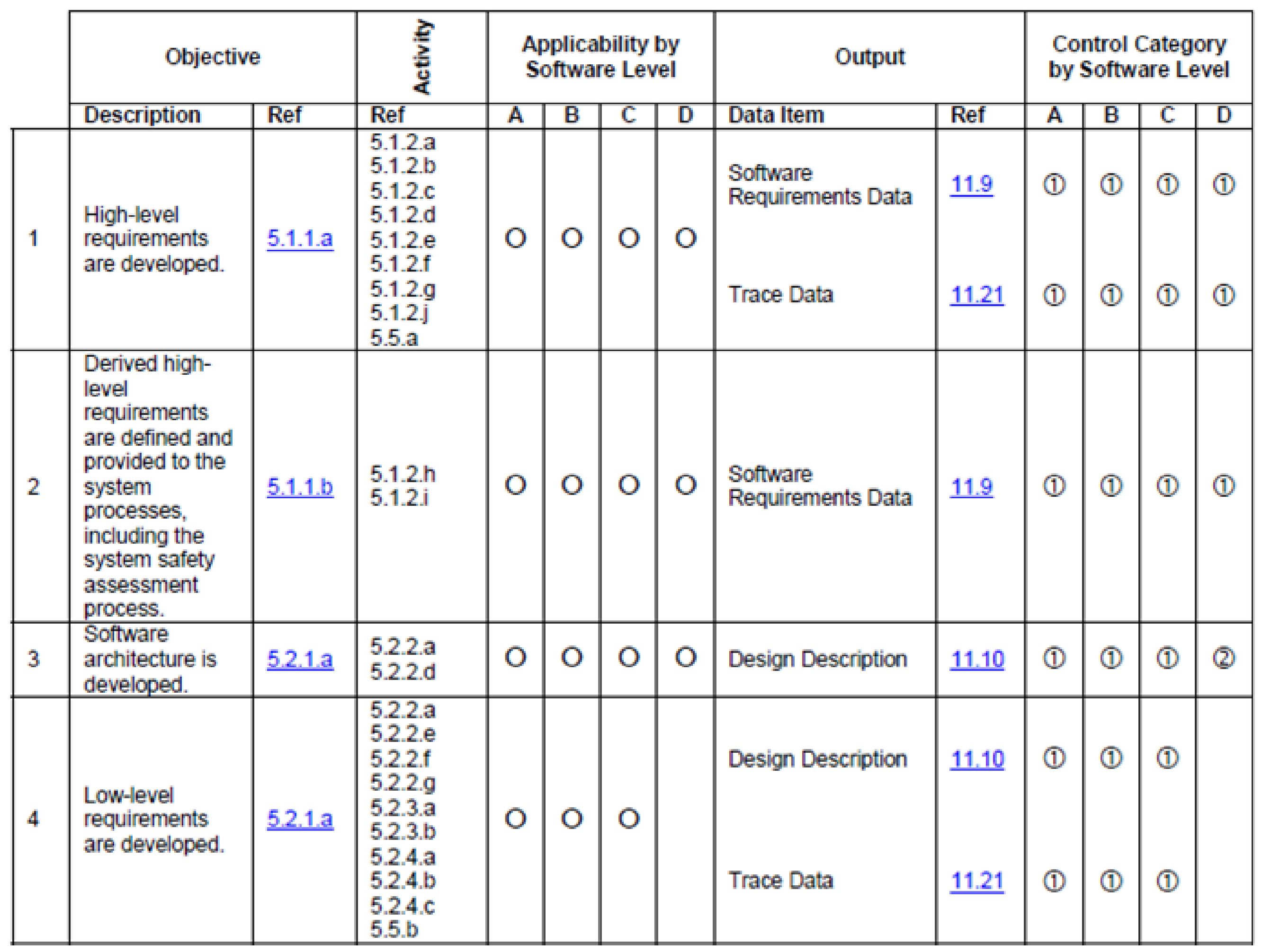
  1. Industrial systems: Cenelec IEC 61508 (Safety integrity level (SIL): SIL1, SIL2, SIL3, SIL4)
  2. Airborne systems and equipment: DO-178C/ED-12C (Design Assurance Level (DAL): E, D, C, B, A)
  3. Automotive: ISO 26262 (Automotive Safety Integrity Level (ASIL): SO, S1, S2, S3)

Quality Assurance (Certification) DO-178C/ED-12C example

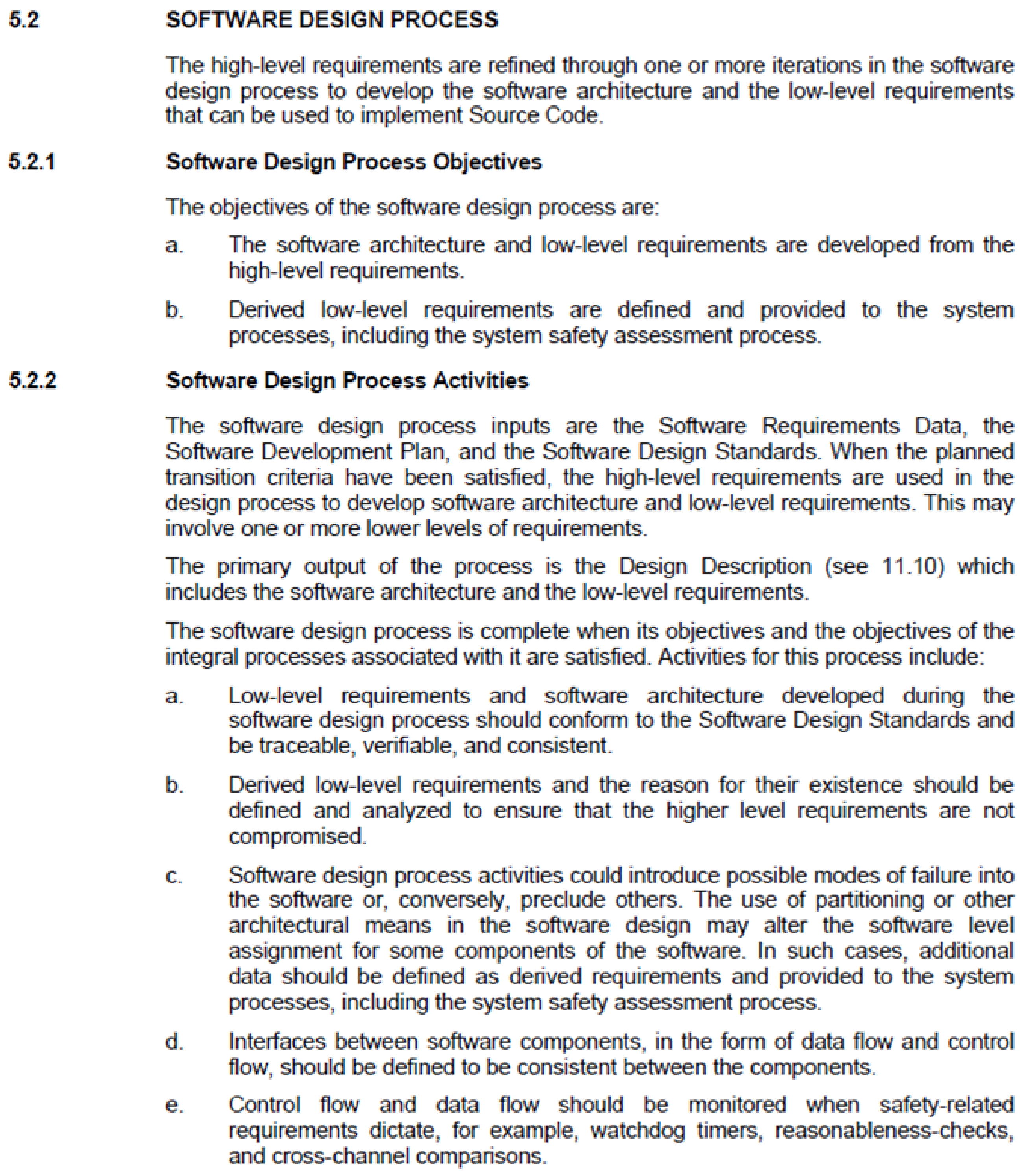
- It is a methodological guide based on good practices to developt a critical software
  - It explains **"What to do"** but it doesn't explain **"How to do"**
  - It explains the **objectives** but not the **methods**
- It leads to:
  1. Write what you are doing (Plan)
  2. Do what you have written (Processus)
  3. Keep evidence of what you have done (traceability)
  4. Compare what was done with what is written (Audit)
  5. Correct deviations
  6. Improve the method

Quality Assurance (Certification) DO-178C/ED-12C Table example

TABLE A-2: SOFTWARE DEVELOPMENT PROCESSES



	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	High-level requirements are developed.	5.1.1.a	5.1.2.a 5.1.2.b 5.1.2.c 5.1.2.d 5.1.2.e 5.1.2.f 5.1.2.g 5.1.2.j 5.5.a	○	○	○	○	Software Requirements Data	11.9	①	①	①	①
								Trace Data	11.21	①	①	①	①
2	Derived high-level requirements are defined and provided to the system processes, including the system safety assessment process.	5.1.1.b	5.1.2.h 5.1.2.i	○	○	○	○	Software Requirements Data	11.9	①	①	①	①
3	Software architecture is developed.	5.2.1.a	5.2.2.a 5.2.2.d	○	○	○	○	Design Description	11.10	①	①	①	②
4	Low-level requirements are developed.	5.2.1.a	5.2.2.a 5.2.2.e 5.2.2.f 5.2.2.g 5.2.3.a 5.2.3.b 5.2.4.a 5.2.4.b 5.2.4.c 5.5.b	○	○	○		Design Description	11.10	①	①	①	
								Trace Data	11.21	①	①	①	



5.2 SOFTWARE DESIGN PROCESS

The high-level requirements are refined through one or more iterations in the software design process to develop the software architecture and the low-level requirements that can be used to implement Source Code.

5.2.1 Software Design Process Objectives

The objectives of the software design process are:

a. The software architecture and low-level requirements are developed from the high-level requirements.

b. Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process.

5.2.2 Software Design Process Activities

The software design process inputs are the Software Requirements Data, the Software Development Plan, and the Software Design Standards. When the planned transition criteria have been satisfied, the high-level requirements are used in the design process to develop software architecture and low-level requirements. This may involve one or more lower levels of requirements.

The primary output of the process is the Design Description (see 11.10) which includes the software architecture and the low-level requirements.

The software design process is complete when its objectives and the objectives of the integral processes associated with it are satisfied. Activities for this process include:

a. Low-level requirements and software architecture developed during the software design process should conform to the Software Design Standards and be traceable, verifiable, and consistent.

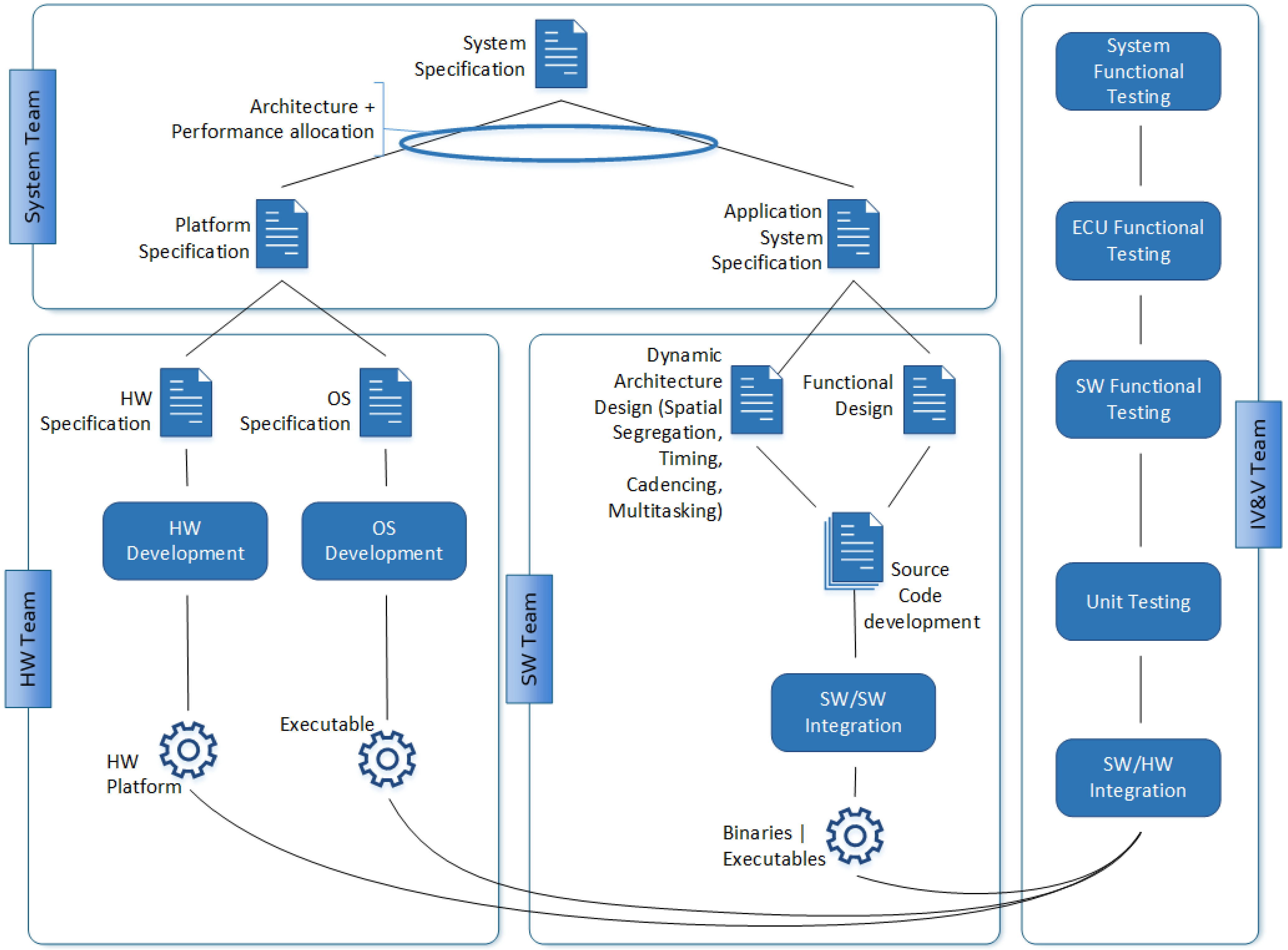
b. Derived low-level requirements and the reason for their existence should be defined and analyzed to ensure that the higher level requirements are not compromised.

c. Software design process activities could introduce possible modes of failure into the software or, conversely, preclude others. The use of partitioning or other architectural means in the software design may alter the software level assignment for some components of the software. In such cases, additional data should be defined as derived requirements and provided to the system processes, including the system safety assessment process.

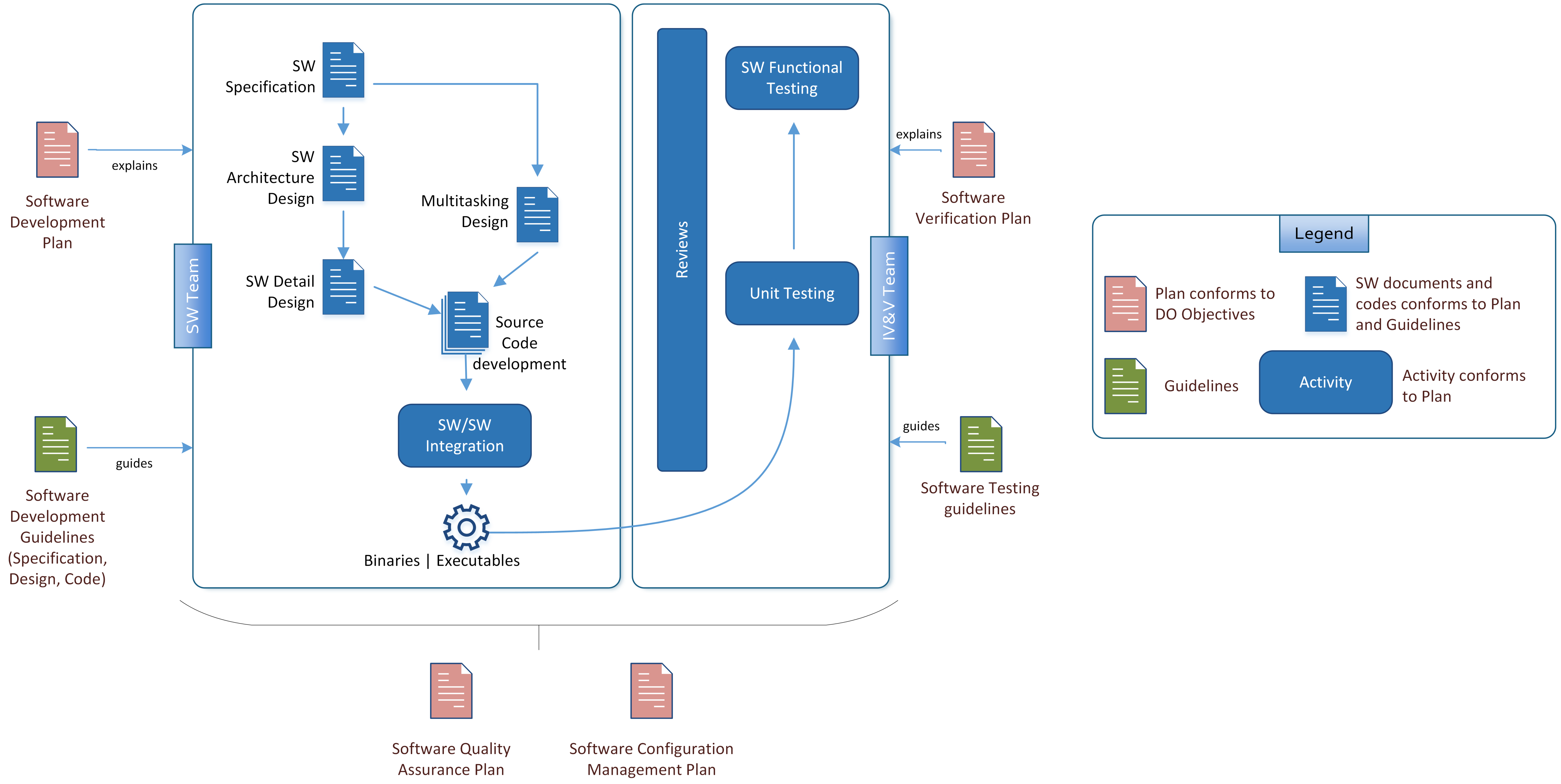
d. Interfaces between software components, in the form of data flow and control flow, should be defined to be consistent between the components.

e. Control flow and data flow should be monitored when safety-related requirements dictate, for example, watchdog timers, reasonableness-checks, and cross-channel comparisons.

Concrete System Engineering for safety software

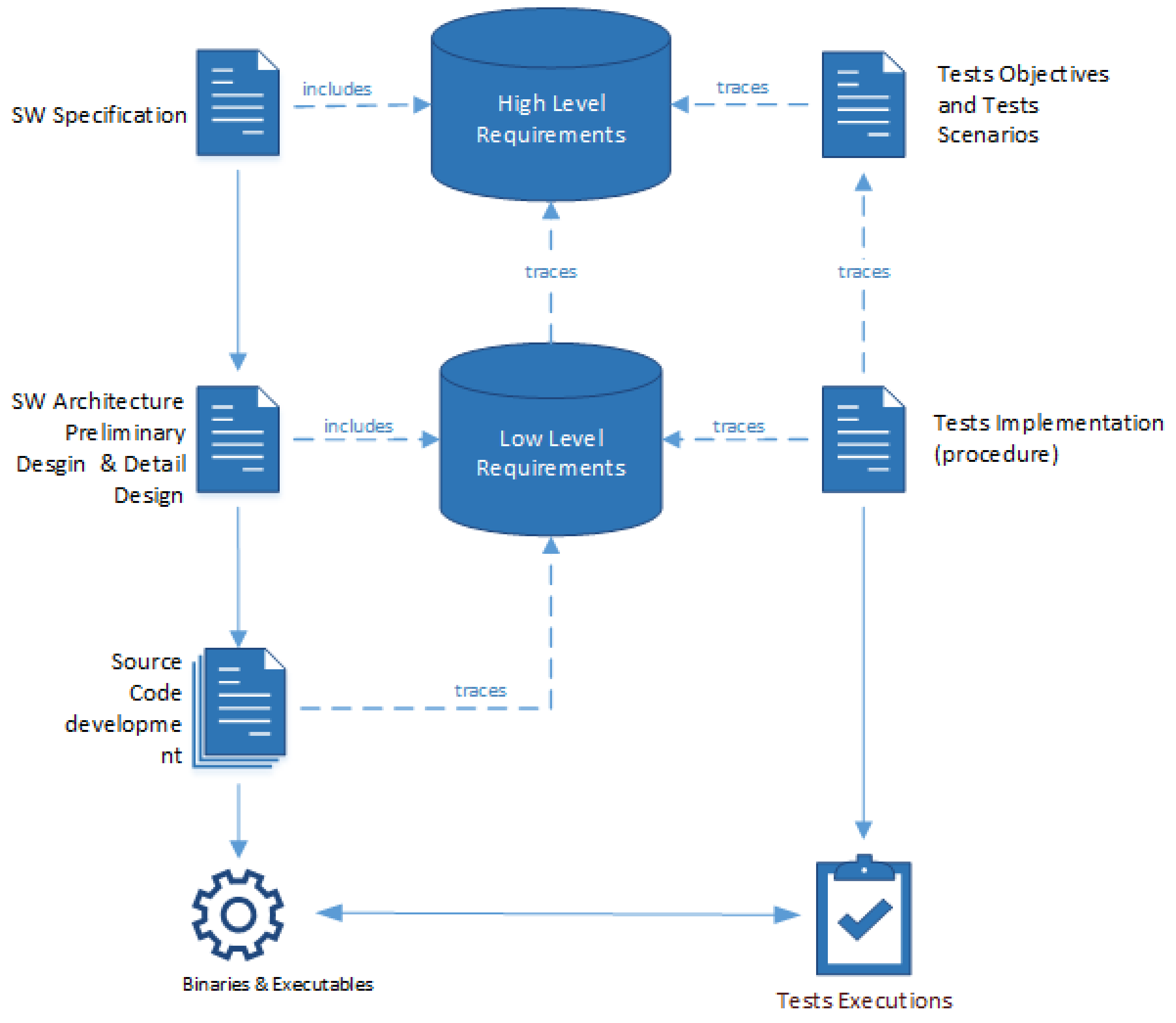

System Team
System Specification
Architecture + Performance allocation
Platform Specification
Application System Specification
HW Team
HW Specification
OS Specification
HW Development
OS Development
HW Platform
Executable
SW Team
Dynamic Architecture Design (Spatial Segregation, Timing, Cadencing, Multitasking)
Functional Design
Source Code development
SW/SW Integration
Binaries | Executables
IV&V Team
System Functional Testing
ECU Functional Testing
SW Functional Testing
Unit Testing
SW/HW Integration

Concrete System Engineering for safety software with DO-178C/ED-12C certification objectives (1/2)



Software Development Plan
explains
Software Development Guidelines (Specification, Design, Code)
guides
SW Team
SW Specification
SW Architecture Design
SW Detail Design
Multitasking Design
Source Code development
SW/SW Integration
Binaries | Executables
Reviews
SW Functional Testing
Unit Testing
IV&V Team
explains
Software Verification Plan
guides
Software Testing guidelines
Software Quality Assurance Plan
Software Configuration Management Plan
Legend
Plan conforms to DO Objectives
SW documents and codes conforms to Plan and Guidelines
Guidelines
Activity
Activity conforms to Plan

Concrete System Engineering for safety software with DO-178C/ED-12C certification objectives (2/2)



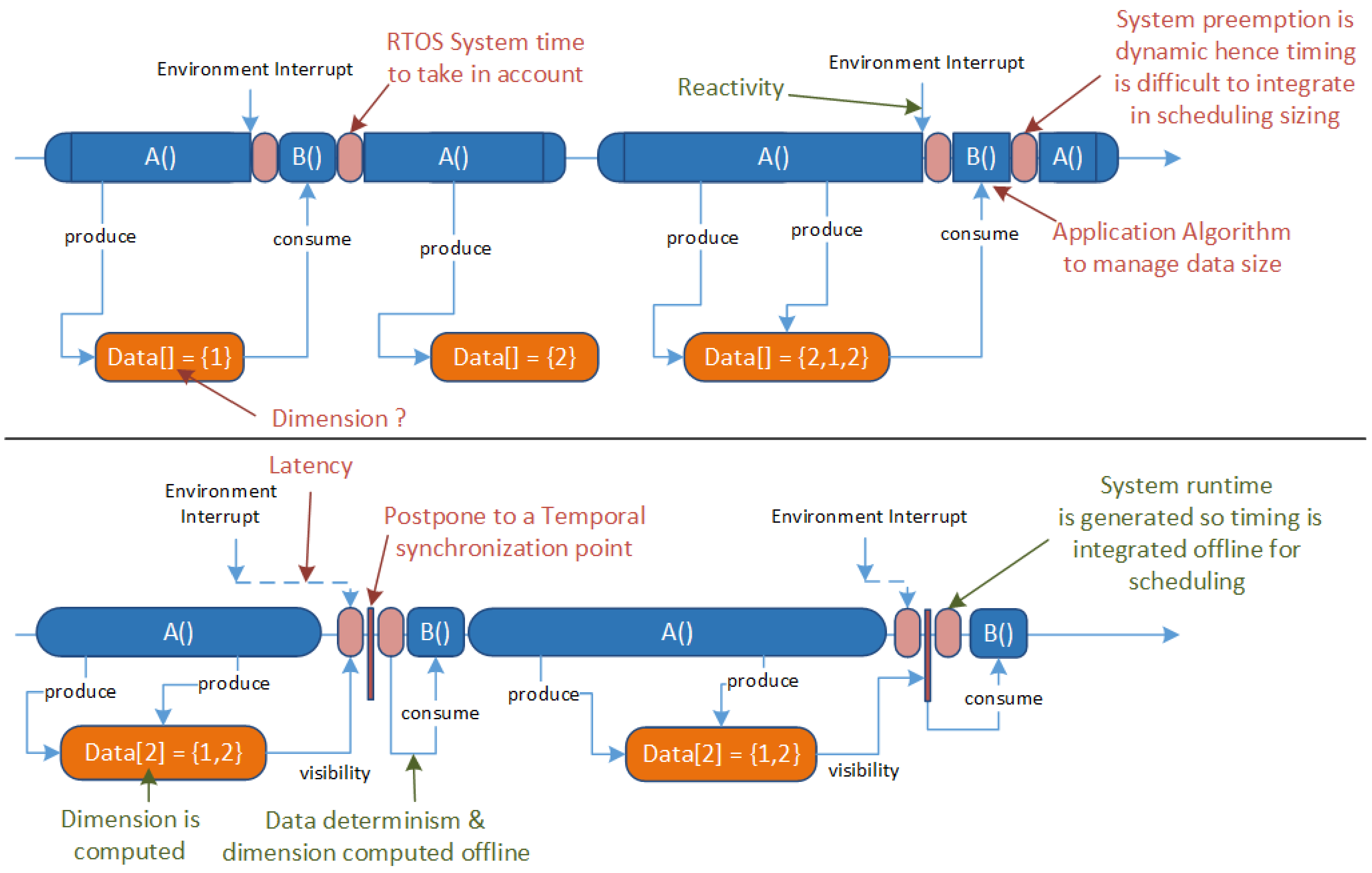
Design approaches and methodologies

On which criteria should I choose an approach to design my application ?

Summary

Summary

Design approaches and determinism (1/3)


Environment Interrupt
RTOS System time to take in account
Reactivity
Environment Interrupt
System preemption is dynamic hence timing is difficult to integrate in scheduling sizing
A()
B()
A()
A()
B()
A()
produce
consume
produce
produce
produce
consume
Application Algorithm to manage data size
Data[] = {1}
Data[] = {2}
Data[] = {2,1,2}
Dimension ?
Latency
Environment Interrupt
Postpone to a Temporal synchronization point
Environment Interrupt
System runtime is generated so timing is integrated offline for scheduling
A()
B()
A()
B()
produce
produce
consume
produce
produce
consume
Data[2] = {1,2}
visibility
Data[2] = {1,2}
visibility
Dimension is computed
Data determinism & dimension computed offline

Design approaches and determinism (2/3)

- Event-Triggered (asynchronous approach): determinism is not provided by construction
  - - Processing are not bounded: unpredictable end to end execution time which depends of the hardware performances
  - - Data communication buffer are not bounded offline: unpredictable overflow
  - - Temporal constraints are traducted to function priorities: determinism is not ensure during integration
  - - Developers/Designers shall demonstrate timing partitionning
  - - CPU workload
  - + Reactivity

Design approaches and determinism (3/3)

- Time-Triggered (synchronous approach):
  - + Temporal constraints shall be given at the design step
  - + Communicated data shall be time stamped at design
  - + All processing are bounded between two instants
  - + All data transfers are done at a temporal synchronization point (Observability): timing partitionning by construction
  - + CPU Load leveled
  - - Latency

Safety Design Concepts: Separation of concerns

Targeted Issues	Increase design complexity Design errors Oversizing
Features	Declarative design approach Decrease design problem in some smaller and simpler ones

- Design tools shall ensure:
  1. Automatic design verification
  2. Automatic application sizing
  3. Automatic implementation generation

Safety Design Key Concepts: Execution support abstraction

Targeted Issues	Problem to assimilate hardware/architecture complexity Non conformance between behaviors and specification
Features	Determinism (each system state is reproducible) Mastering execution (each system state is known) Portable application / Generic execution platform

- API shall be transparent: System call are generated and verified
- All interrupts shall be managed by underlying system: System state is precisely controlled
- Application shall be independent from platform execution: Execution platform and application shall be reusable
- Hardware mutlicore synchronization abstraction

Safety Design Key Concepts: Completeness of the description

Targeted Issues	**Bad specification interpretation** **Error specification transcription** **Bad design or V&V**
Features	Determinism (each system state is reproducible) Mastering execution (each system state is known) Portable application / Generic execution platform

- System control is based on complete description of software behaviors
- All hypothesis shall be describe explicit and without ambiguities

1. Whole interactions designed and controlled
2. Expected behavior descriptions shall be independent of the architecture performances
3. Faulty behaviors shall be describe and integrate in the sizing proofs

Safety Design Key Concepts: Error confinement

Targeted Issues	**Interference among functions** **Not reproducible behaviors** **Spatial partitionning**
Features	System is always in a controlled state

- Confinement objective: Avoid errors propagation in the whole software in order to allow services continuity
- Errors must be confined near their source: To reduce their impacts and to produce a software in known and safe states

Safety Design Key Concepts: Temporal expressions

Targeted Issues	Bad design of temporal constaints Bad treatments order
Features	Explicit expression of all expected behaviors on time

- To mix expression of algorithmic processing and expected temporal behavior
- To specify rate at the design step
- To include communication (inter-processing and input/output)

Safety Design Key Concepts: CPU load management (Sizing management)

Targeted Issues	Oversizing, deadline miss
Features	Online & offline timing verification

- Timing property have to be
  1. Ensured and verified at compile time
  2. Checked at runtime
- Offline sizing demonstration is required

Summary

Design Approaches: Event trigger approache example (1/2)

- UML has been designed to specify and analyze the architecture of large software
  - Class diagram and component diagram to define the architecture
  - Sequence diagram to describe the event sequences
  - Activity and State Diagram to define the behaviors
  - A set of editor and code generator
  - Based on asynchronous event trigger computation model
  - No formal execution
  - No formal time model
  - No Safety model features

UML

Design concepts	Approach Evaluation	Comments
Determinism	-	Execution model is not formally defined.
Separation of concerns	++	Declarative approach. Concepts to describe the designs but no formal execution semantics
Execution support abstraction	-	Execution model is not formally defined. No execution environment
Completeness of the description	++	Multiple level declaration (design, software, multitasking, hardware allocation) but no timing extension
Error confinement	-	No error specific management included in the langage
Temporal description	-	No specific langage artifacts to manage temporal description
CPU load management	--	No Sizing available as no temporal descriptions
Certification	--	

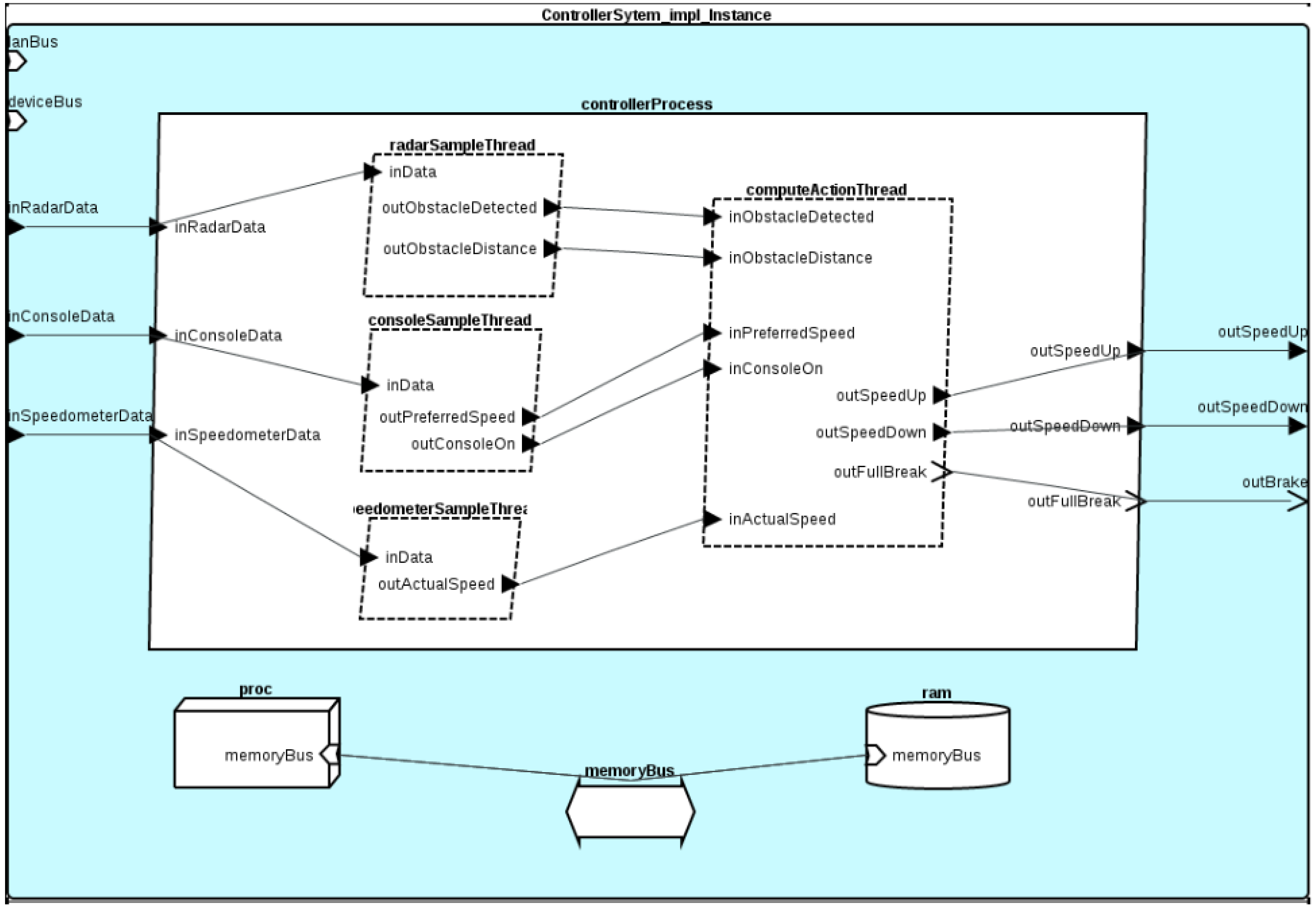
UML

- Use UML in the Software Architecture Description with specific guidelines
  - Components diagrams to describe the architecture and the allocations
  - Sequences diagrams to describe the thread, the sequence of actions

Design Approaches: Event trigger and Time trigger approache example (2/2)

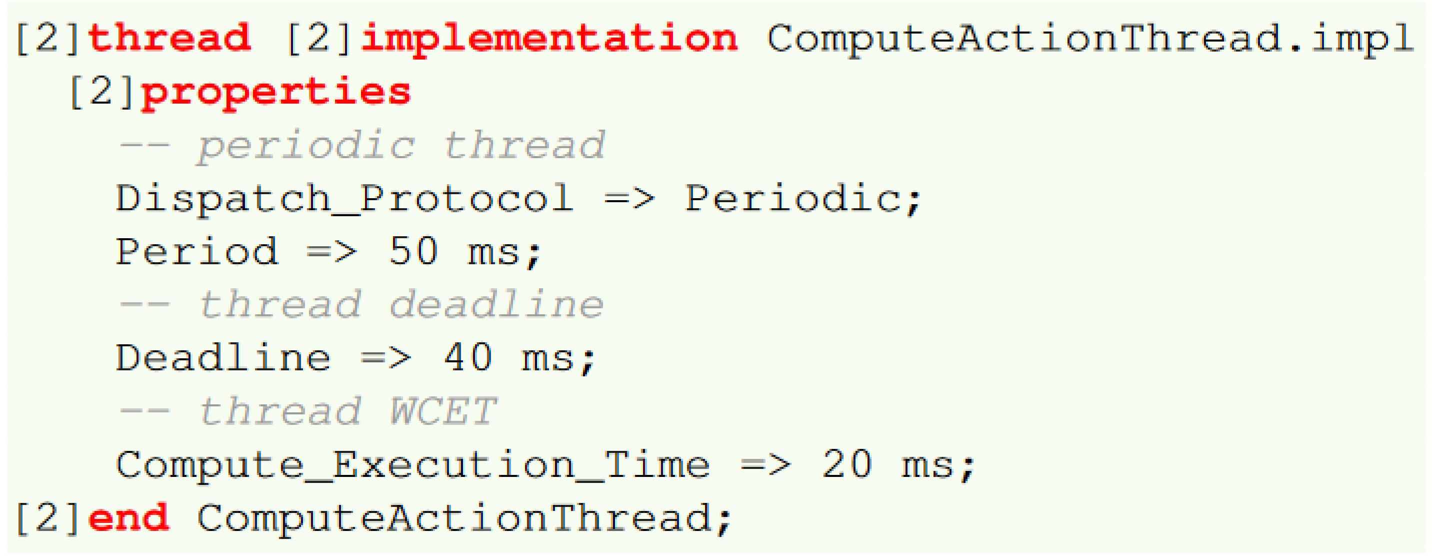
- AADL has been designed to specify and analyze the architecture of large embedded real-time systems
  - Component based model: Architecture Description Langage (definition of the thread interfaces)
  - A set of editor, analyzer and code generator(e.g. Ocarina)
  - Based on both synchronous and asynchronous event trigger computation model
  - AADL Standard specifies an execution model as a virtual runtime environment

AADL Cruise Control Example



Extracted from "Formal semantics of behavior specifications in the architecture analysis and design language standard"

AADL Cruise Control Example: timing properties definition

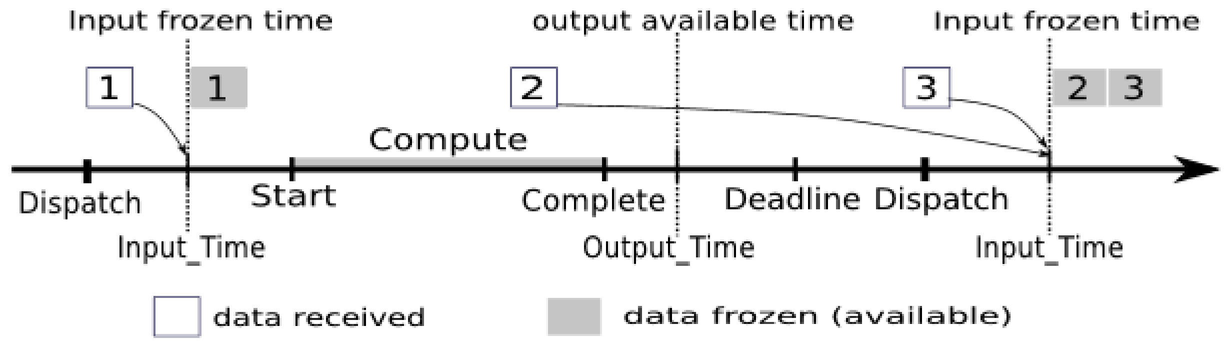


```
[2]thread [2]implementation ComputeActionThread.impl
  [2]properties
    -- periodic thread
    Dispatch_Protocol => Periodic;
    Period => 50 ms;
    -- thread deadline
    Deadline => 40 ms;
    -- thread WCET
    Compute_Execution_Time => 20 ms;
[2]end ComputeActionThread;
```

Extracted from "Formal semantics of behavior specifications in the architecture analysis and design language standard"

AADL

- AADL timing execution model: A thread is activated to perform a computation at start time, and has to be finished before the deadline. A complete event is sent at the end of the execution. The received inputs are frozen at a specified time (Input Time), by default the dispatch time.



Extracted from "Formal semantics of behavior specifications in the architecture analysis and design language standard"

AADL

- Example of AAD Thread
  - Periodic Thread: Thread is awake periodicaly.
  - Sporadic thread: Thread is awake when receiving events/messages. Message timing properties are defined (lower and upper bound).
  - Aperiodic thread: Thread is awake when receiving messages. No timing properties defined for messages

AADL

Design concepts	Approach Evaluation	Comments
Determinism	+	Execution model is formally defined by a specific annex.
Separation of concerns	++	Declarative approach. Concepts to describe the designs.
Execution support abstraction	+	Execution model is based on a virtual execution environment implemented in specific middlewares
Completeness of the description	++	Multiple level declaration (design, software, multitasking, hardware allocation). Behaviors is described by C codes.
Error confinement	+	Process concept defines memory partition. Memory partition is dependant of the operating system used.
Temporal description	-	Declaration of timing properties (periodic behaviors). Time is not defined by clocks.
CPU load management	-	Tools exists to analyse the schedulability. Schedule computed is not the one that is executed. Scheduling is dependant of the operating system used. Aperiodic thread descriptions.
Certification	--	No certified middlewares

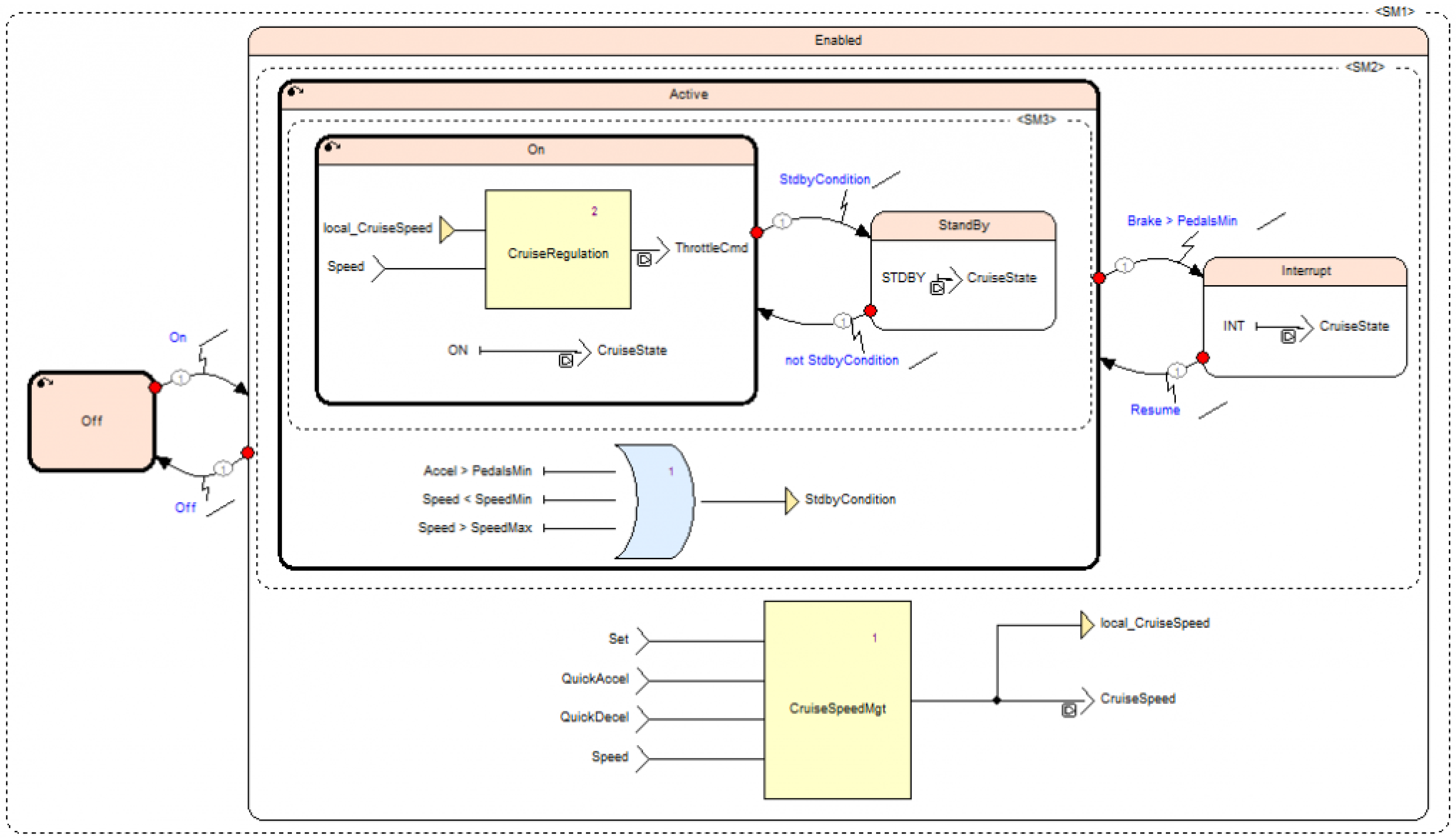
Design Approaches: Time trigger approaches (1/3)

- Time-Triggered Protocole for ethernet communication
- Integrated modular avionics (IMA)
    - No specific languages but a design approach with memory and timing partitions
    - ARINC-653 operating systems implement that design approach
    - Micro-designs to define the timing partitions
    - ARINC-653 example

Design Approaches: Time trigger approaches (2/3)

- Lustre & Esterel based model
  - Synchronous approach: execution is immediate and activities are periodic
  - Formal model of time, Formal simulation
  - Industrial success: SCADE (Esterel-Technologies company) is used for programming critical control sofware (e.g., planes, nuclear plants).
  - No Partitioning model
  - No specific embedded kernel
  - No specific kernel and no scheduling generator among multiple synchronous application (Pb with mixing applicaiton on the same CPU)

Scade Suite Cruise Control Example



Extracted from "SCADE: Industrial Success of a Synchronous Language and its Future Challenges (2007) by Amar Bouali"

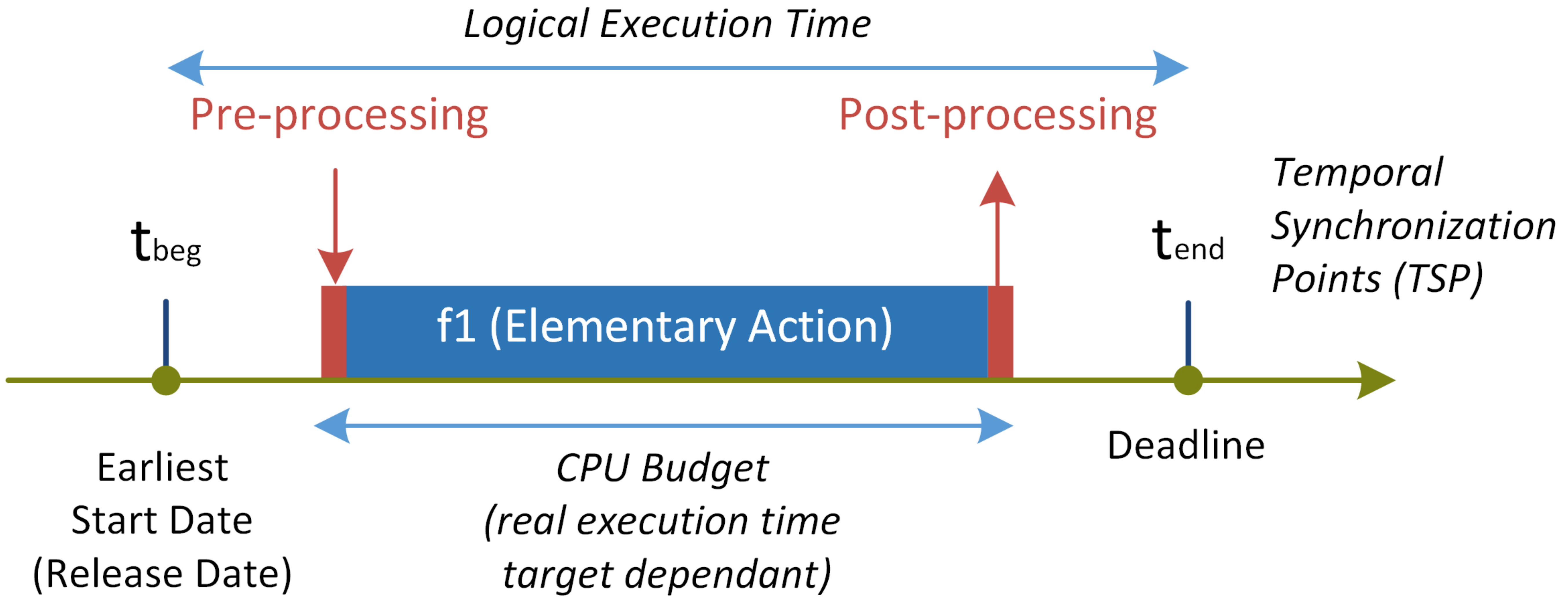
Lustre & Esterel based model

Design concepts	Approach Evaluation	Comments
Determinism	++	Synchronous computation model is determinist
Separation of concerns	++	Declarative approach, dataflow and stateflow descriptions, automatic verification, automatic qualified code generation
Execution support abstraction	++	As execution is immediate, hardware independance
Completeness of the description	++	Description of all functional and temporal behaviors (dataflow and state flow descriptions)
Error confinement	-	No specific implementation model, no specific partitionning model (this is an operating system problem from scade point of view)
Temporal description	++	Clocks definitions and synchronous state machine description
CPU load management	-	As execution is immediate, temporal sizing is not computed at design level
Certification	++	Code generator from Scade models that has been qualified as a development tool for DO-178B software up to Level A, DO-178C/DO-330 at TQL-1, certified for IEC 61508 at SIL 3, and for EN 50128 at SIL 3/4, and qualified for ISO 26262 software up to ASIL D

Design Approaches: Time trigger approaches (3/3)

- GIOTTO
  - Code interpretation on embedded
  - Safety oriented
- Psy Approach
  - Synchronous & Asynchronous approach: execution is not immediate and activities are not only periodic
  - Formal model of time, Formal simulation, Safety oriented
  - Industrial implementation
  - Specific Kernel for embedded systems
  - Code and safety scheduling plan generation

Logical Execution Time Model


Logical Execution Time
Pre-processing
Post-processing
Temporal Synchronization Points (TSP)
t_{beg}
t_{end}
f1 (Elementary Action)
Earliest Start Date (Release Date)
CPU Budget (real execution time target dependant)
Deadline

Psy Approach

Design concepts	Approach Evaluation	Comments
Determinism	++	Psy computation model is determinist
Separation of concerns	+	Declarative approach and automatic verification. Behviors are described in C.
Execution support abstraction	++	Psy computation model is not dependant of the scheduling and hardware cores available
Completeness of the description	+	Description of all functional and temporal behaviors
Error confinement	++	Psy computation model is based on agent that are memory partition. Memory partition configuration is automatically generated offline
Temporal description	++	Clocks and constraint definition in the C langage
CPU load management	++	Scheduling is computed at design level (offline) from the clocks and the timing constraints
Certification	-	Not yet qualified