



Open License Society

www.OpenLicenseSociety.org

Unifying and systematic system development methodologies
with trustworthy embedded components

Eric.Verhulst@OpenLicenseSociety.org

What is Open License Society?

- Privately funded R&D institute (non-profit)
- Sponsoring member: Melexis (automotive)
- Objectives
 - Systematic & Unified Systems Engineering Methodology
 - 'Interacting Entities' paradigm at all levels:
 - OpenComRTOS as runtime environment (formal development in IWT project)
 - Implies 'Trustworthy Components'
 - => Open License (source code + all design, test, docs)
- Why? 70 % of all SE projects do not deliver
- Focus:
 - Embedded Systems:
 - Constraints driven development
 - Real-time, distributed, hardware & software, ...

The Wall of Complexity challenge(1)

- Murphy's law: things going wrong is a matter of probability, and the odds are getting worse
- Focus domain: embedded systems
- Products become systems:
 - ,Smart' by using 10's of processors
 - Distributed operation
 - Connected to other systems (wireless)
 - Human in the loop
 - Requirements:
 - High quality, high reliability
 - High level of safety, fault-tolerance
 - Increasing need for Secure operation
 - And as well:
 - Cost-efficient (life-cycle cost)
 - Competitive
 - Upgradeable

The Wall of Complexity challenge(2)

- The solution is NOT to link the myriad of existing processors, software tools, etc.
- Because they are semantically too different
- Because we have too many of them
- But none supports scalability requirement
- Few support “graceful degradation and scalability”
- Time to apply occam`s razor:
 - Back to basics
 - Get rid of unnecessary complexity and historical ballast
 - More engineering, less crafting, more scalability and real re-use
- => ,Trustworthy Embedded Components`
 - Reliability, correctness
 - Safety, fault-tolerance
 - Security
 - Scalability, graceful degradation
 - Formally developed and validated software & IP
 - Open License: source code + validation, design, test data

Challenges in achieving functional safety: IEC

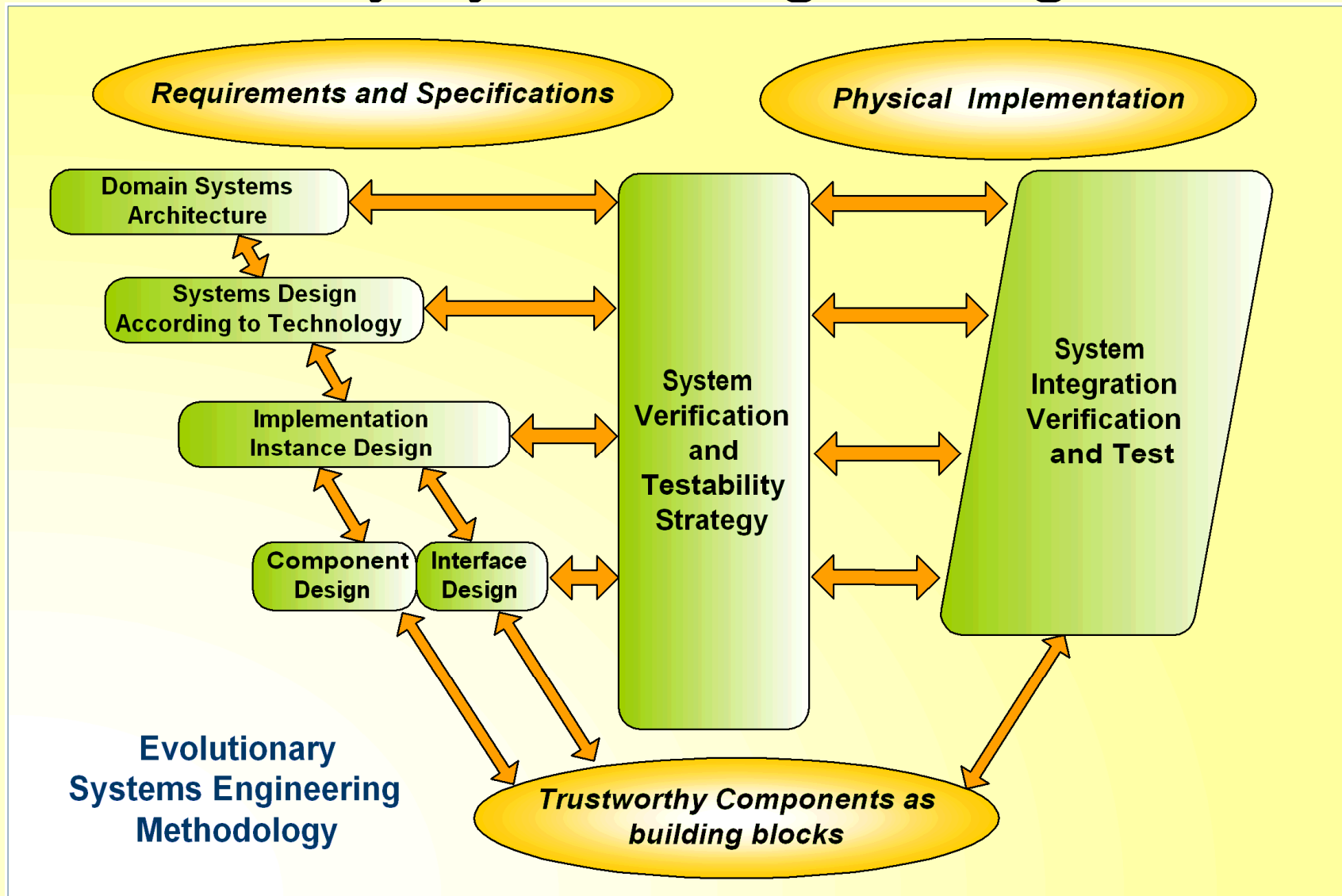
61508 standard quoted:

- "Safety functions are increasingly being carried out by electrical, electronic or programmable electronic systems. These systems are usually complex, making it impossible in practice to fully determine every failure mode or to test all possible behaviour. It is difficult to predict the safety performance, although testing is still essential."
- The challenge is to design the system in such a way as to prevent dangerous failures or to control them when they arise. Dangerous failures may arise from:
 - incorrect specifications of the system, hardware or software;
 - omissions in the safety requirements specification;
 - random hardware failure mechanisms;
 - systematic hardware failure mechanisms;
 - software errors;
 - common cause failures;
 - human error;
 - environmental influences (e.g. electromagnetic, temperature, mechanical, ...)
 - supply system voltage disturbances

A Systems Engineering point of view

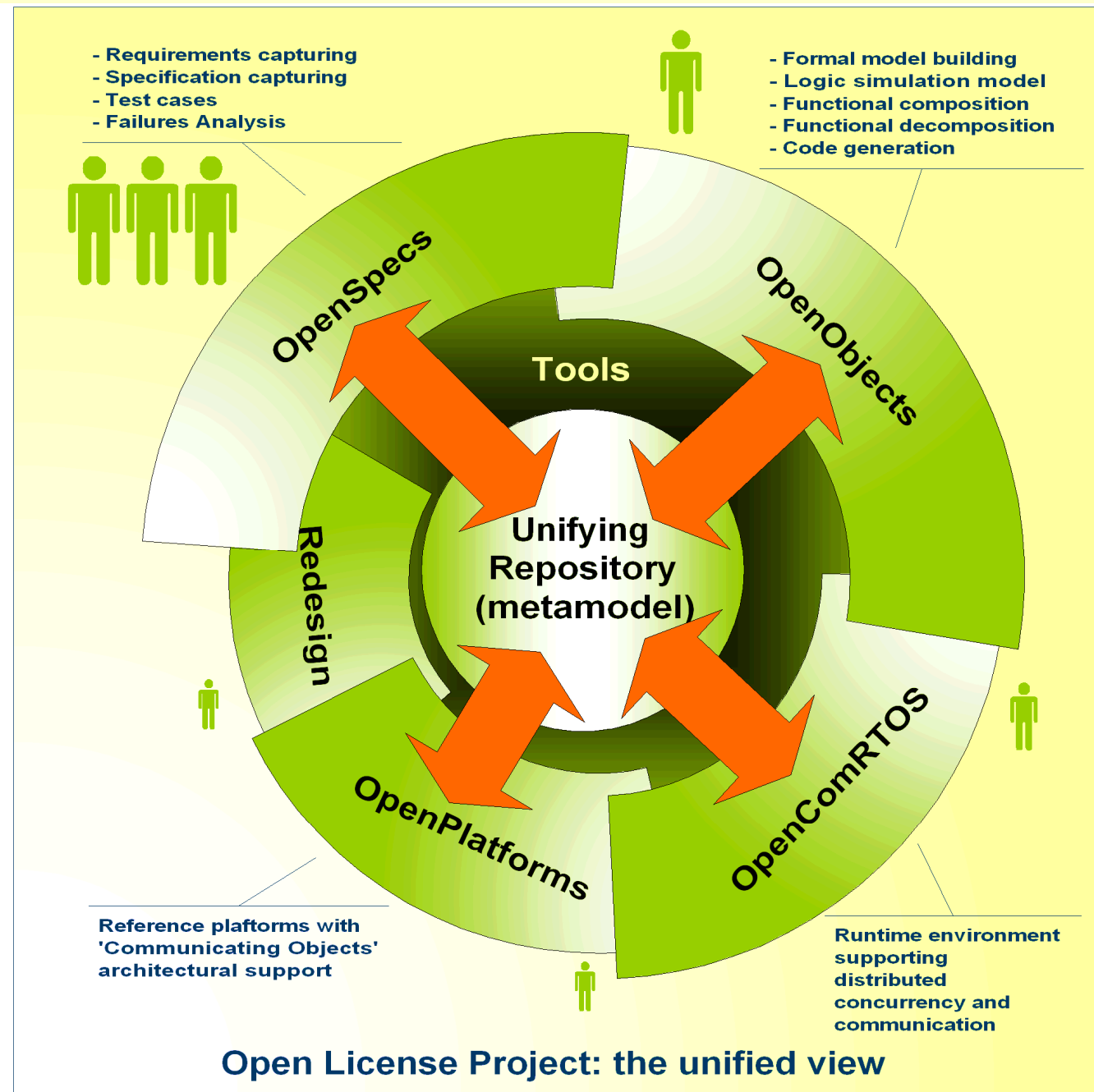
- A system is an implementation of a desired system functionality in a physical object (from the very small to the very large)
- This functionality must exhibit certain properties: (-> Requirements)
 - Correctness
 - Reliability
 - Safety & Fault Tolerance
 - Security
 - But also: price, ease-of-use, size, ...
- The methodology to reach those requirements is generic!

Evolutionary Systems Engineering

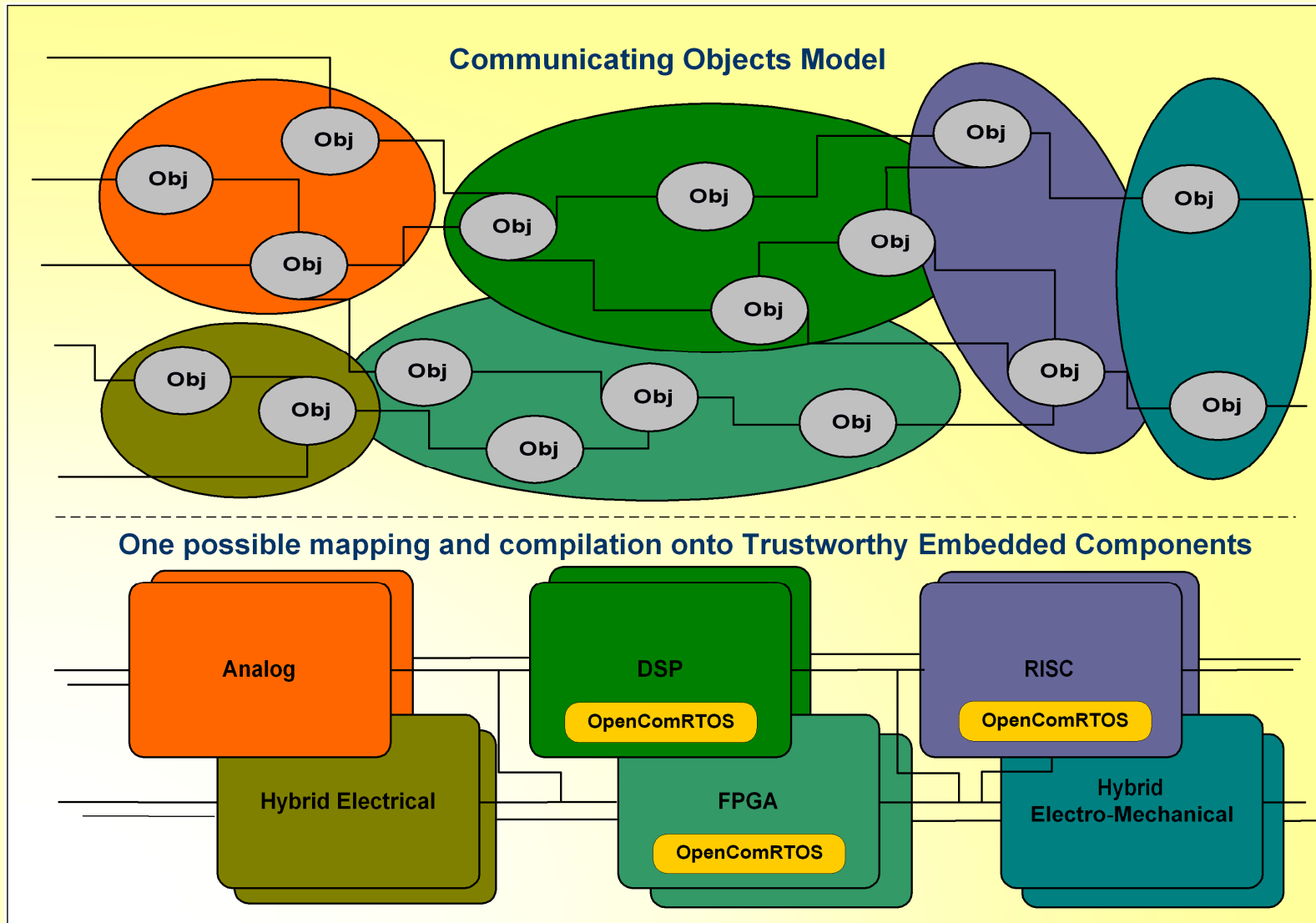


Requirements for Evo (ideally)

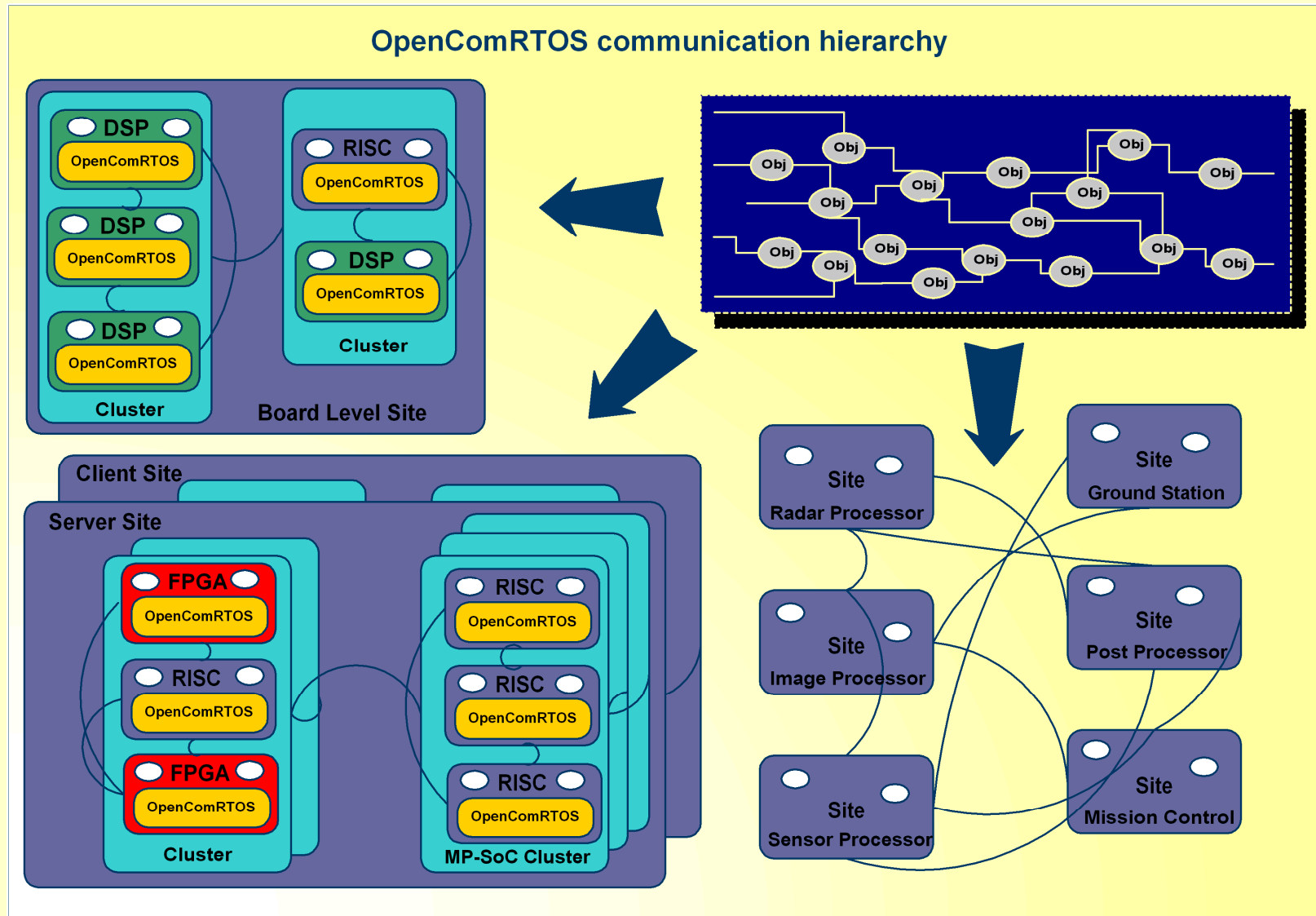
- All supporting tools must be integrated
 - Requires common ,semantics`
 - All activities from design to software to hardware must be compatible semantically, ideally syntactically as well
 - Instant notification of any changes to all activities
- Scale of development must be consistent:
 - Modular architecture
 - Information hiding and isolation between modules
 - Formal basis
- Selected paradigm:
 - ,Interacting Entities`: universal and simple
 - Formal foundation: CSP, CommUnity, ...
 - Concurrent programming model



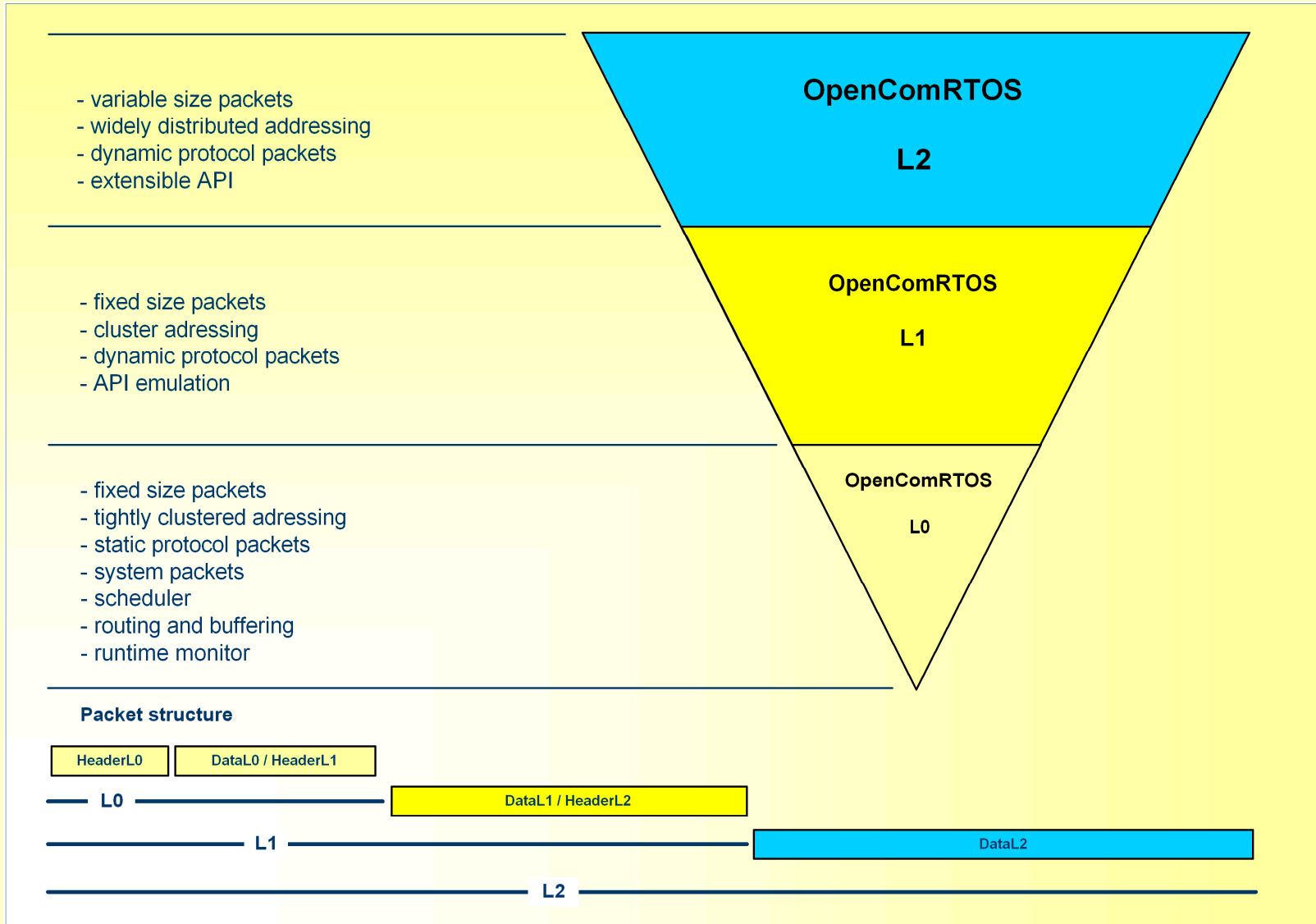
Unifying paradigm (1): Interacting Entities

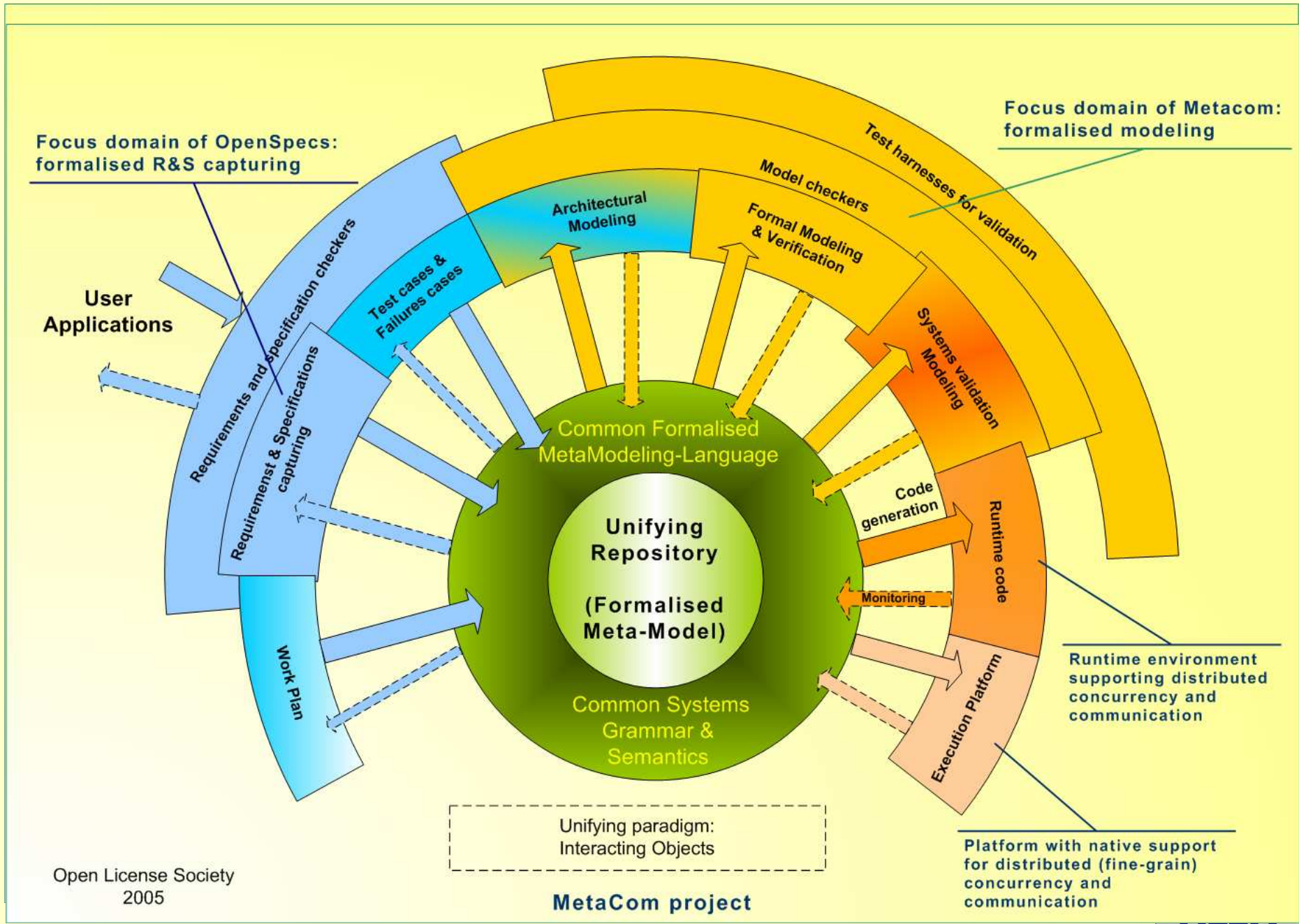


Unifying paradigm (2): Scalable Communication



OpenComRTOS: formally developed

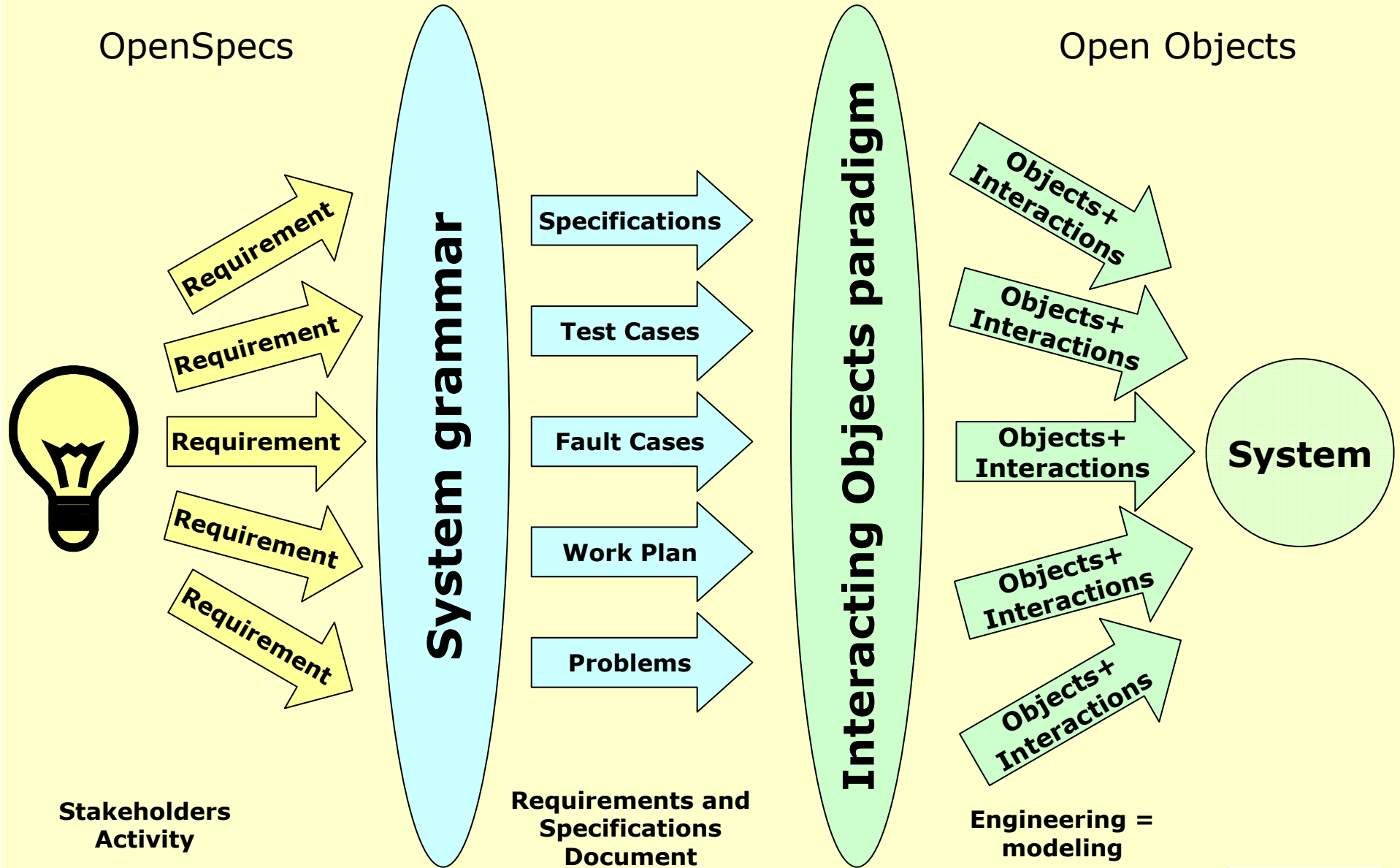




Requirements & Specifications capturing

- From natural language Requirements to Specifications in formalised language
- => incl. Test Cases, Failure Cases
- System = All aspects, incl. workflow
- SE environment: = different views
- Minimum system
 - = 3 Objects + Interactions:
 - System under development
 - Environment
 - Operator
- Tools to check for consistency and completeness

General Structure of Systems Engineering Process



Systems grammar (example)

System CONSISTS_OF

Entities (2-N) AND
Interactions (1-N)

Entity IS_DEFINED_BY

Properties (2-N) AND
Actions (0-N) AND
Systems (0-N) AND
Interfaces (1-N)

Entity HAS_ATTRIBUTES

Requirements (1-N) AND
Specifications (1-N) AND
TestCases (1-N) AND
FailureCases (1-N) AND
WorkPlanTasks (1-N) AND
WorkPlanStatus (1)

Towards a SE metamodeling-language

- **Why?**
 - Gap between “natural language” and implementation
 - Natural language is not precise enough
- **Solution: “formalised language”**
 - Sentences following “systems grammar and semantics”
 - Independently of modeling domains: high level
 - Stick to essential concepts first, instances can be adapted to specific domain
 - Benefits: unique description
- **Challenges:**
 - Covers many “views” of systems engineering process
 - Can it be used a “programming language” ?
 - Systems compiler(s) ?
 - Can it be descriptive and executable?

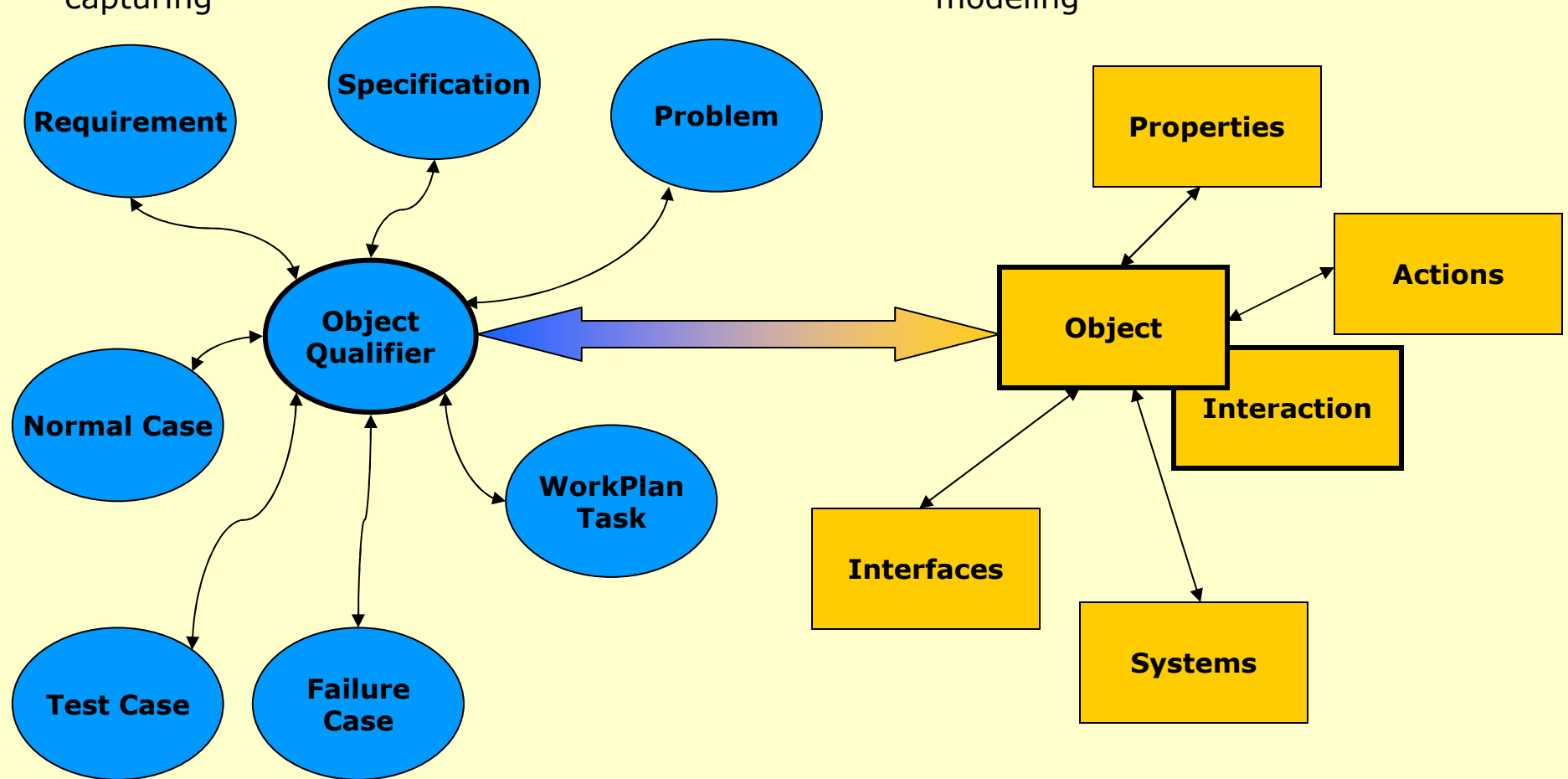
Modeling: the main engineering activity

- All aspects, roughly 3 classes:
 - structural and operational aspects (architecture)
 - functional and state aspects (essential properties)
 - global aspects (virtual prototyping, what-if-scenarios)
- Dependency analysis, evolutionary view
- Ideally just one model: but too heavy
 - Each results in a specific "view"
- Model checkers automate the checking
- Test harness generator:
 - Automated
 - Confidence test, less to find errors (unless in the model)

Gnosiological and ontological levels

Requirements and Specifications capturing

Architectural and functional modeling



Code generation for modeling

- Each model can be seen as a specific execution engine (virtual machine)
 - Writing at low level is error-prone
 - Each execution engine should be independent
- Solution is to insert metamodeling-layer
 - Using a HL metamodeling-language
 - Metamodeling language exploits meta-model
 - Formalised description expressed in formalised concepts, systems grammar and syntax, sentences
 - Generic but covering all views
- 'Systems compiler':
 - Preserve the properties from R&S till implementation

Runtime environment (software)

- Objects and their interaction are 'linked' with (proven) runtime components
- Ideally = proven and tested model
- Extra boundary conditions:
 - Real-time behaviour, performance, power consumption
 - Cost and size
 - Should be scalable
 - Later: inherently safe and secure
 - Monitoring for confidence and post-fault analysis

Formal methods in the Systems Engineering context

- Objectives:
 - Engineering-wise:
 - Verify before implementation (ALL aspects)
 - Analyse for correctness before testing (ALL aspects)
 - Simplify the design (but not too much! -> minimum but necessary)
 - Validate the design before testing
 - Testing becomes a confidence test, not for detecting errors
 - Business-wise:
 - Higher success-rate
 - Lower development costs
 - Short start-up times, immediate higher productivity
 - Reduced recurrent costs (re-use of IP)
 - “trustworthy components”: certification by design
 - Reduced life-cycle cost:
 - Note sure awareness is strong yet
 - Reduced business risk: control and ownership

Problems identified (1)

- Academic vs. industry-strength tools
 - Stability of code
 - Windows and Linux support
 - Work as well on larger systems: proven on real designs
 - Fast enough
- Not yet unified:
 - Different tools and methods for different properties
 - Completeness?
- Notational barriers:
 - Mathematical notation:
 - Models must be discussed with all stakeholders
 - Learning curve
 - Keyboard input

Problems identified (2)

- **FM are labor intensive (well, not always ...)**
 - Modeling requires detailed descriptions
 - Productivity low (compared with e.g. high level language programming) but not when considering total system life
 - Need for higher abstraction levels
 - Use of certified libraries ?
 - High level front-end tools
 - Graphical input
 - HL or metamodeling languages
 - Needs to be addressed else applicability will remain limited due to economic constraints
- **Essence is “formalised thinking”**
 - Applies to all activities: from R&S, modeling, platform
 - Even without formal methods, the results should be much more predictable

When will industry apply FM?

- **Cost-efficiency:**

- Awareness of consequences when not applying FM
 - Crisis situations that cost money: e.g. Automotive
- Tools are affordable
- Ownership of process

- **Skill-efficiency:**

- Useable by average engineer
- Education and training must be changed !

- **Time-efficiency:**

- Same level of productivity as with current practice

Conclusion

- **Unified Systems Engineering is essential:**
 - Addressing the challenge of “first-time-right” development
 - Covers hardware, software, ...: all domains
 - Domain engineering breeds expertise
 - Methodology also works for e.g. business processes!
 - Economic:
 - Better quality with same effort and resources
 - Safety and security are becoming a “must-have”
 - Competitiveness
 - Liability issues
 - Bootstrapping a new market of “Trustworthy components”
 - Will divide the market in two classes: verified or not verified
 - Cfr. www.verisoft.de