Lessons Learnt from the Adoption of Formal Model-based Development

Alessio Ferrari[1] Alessandro Fantechi[1,2]
Stefania Gnesi[1]

[1]ISTI-CNR, Pisa, Italy
[2]DSI, University of Florence, Florence, Italy

April 3, 2012

Overview

Domain

- Railway signalling
- Standard-regulated framework

Technologies

- Formal model-based development & Code generation

Tools

- Simulink/Stateflow toolsuite

Goals

- Identification of a safe-subset of the modelling language
- Evidence of the behavioural conformance between the generated code and the modelled specification
- Integration of the modelling and code generation technologies within the process that is recommended by the regulations

Tuning of the approach across 3 Projects

Formal model-based design

Formal methods

- Ten Commandments of Formal Methods...ten years later

  (Bowen, J.P., Hinchey, M.G., *IEEE Computer*, 1995-2006)
- Formal methods are still perceived as experimental technologies

Model-based design

- Model-based design is born later, but gained ground much faster
- Graphical simulation is more intuitive than formal verification
- Simulink/Stateflow is a *de-facto* standard

Challenges

- How to ensure that the generated code is compliant with the modelled application?
- How to integrate model-based practices with traditional certified processes?
- Formal model-based design

Problem Statement

History

- Medium-size company operating in safety-related railway signalling systems development
- Introduction of Simulink/Stateflow as design tools
- 4 years research activity in collaboration with academia to adopt code generation

Problem Statement

Define and implement a methodology for the adoption of the code generation technology from formal models by a railway signalling company

Goals

Goal 1 - Modelling language restriction

- Safety-critical code shall conform to specific quality standards
- Companies use coding guidelines in order to avoid usage of improper constructs
- Identification of a safe subset of the modelling language

Goal 2 - Generated code correctness

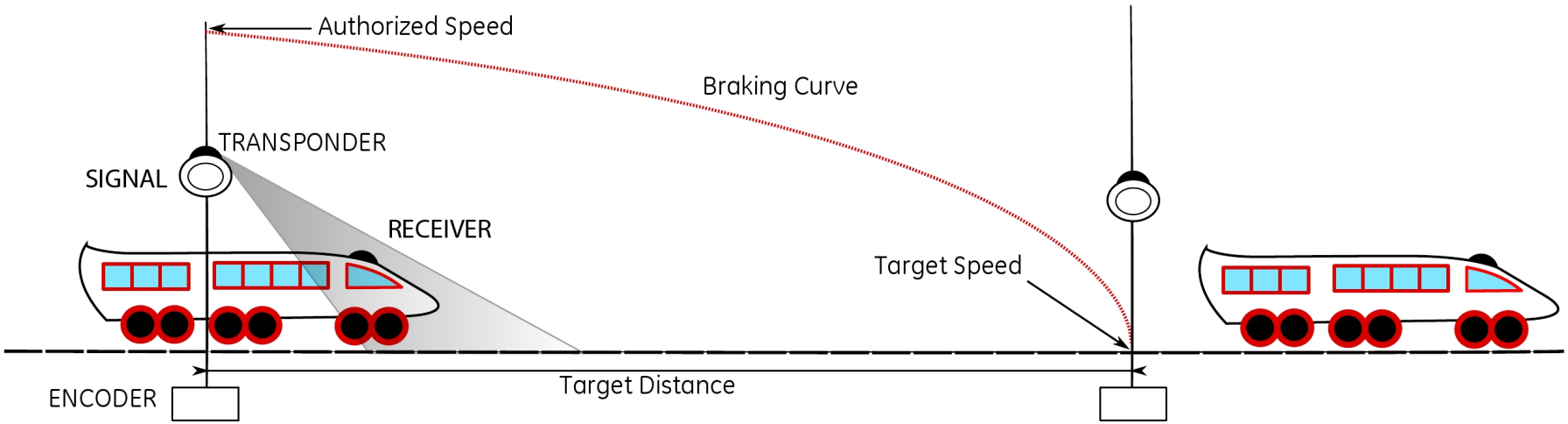
- Proven-in-use translator required
- Compliance between generated code and model behaviour

Goal 3 - Process integration

- Introduction of new technologies in an established process requires adjustments to the process structure
- Process according to normative prescriptions

Projects

Automatic Train Protection (ATP) Systems



Project 1 History

Prototyping

- A Stateflow model was designed in collaboration with the customer in order to define and assess the system requirements
- Assessment of the potentials of modelling for prototype definition and requirements agreement
- Succesful deployment of the system

Code Generation Experiments

- Experiments started when the system was already operational
- Definition of a first set of modelling guidelines
- Proper code synthesis of the single model units was achieved through Stateflow Coder
- No V&V process still defined
- Project 1 remained an hand-crafted system

Project 2 History

Towards a V&V process

- Ten times larger than Project 1 in terms of features
- Internal guidelines integrated with MAAB (Matlab Automotive Advisory Board) recommendations
- Static anlaysis at model level for guidelines verification
- Functional unit-level verification
  - Model-based testing
  - Abstract interpretation (Polyspace)

Evolution

- Strict timing of the project
- Model-based testing and guidelines verification only partially employed
- *Ad-hoc* solution for code verification

Project 3 History

A formal development process

- ATP for metro signalling system
- Project 1 < Project 3 < Project 2
- Hierarchical derivation approach with UML support
- Real-time Workshop Embedded Coder adopted
- Functional unit-level verification
  - Translation validation (back-to-back testing)
  - Abstract interpretation (Polyspace)

Experiments with formal verification

- Simulink Design Verifier
- Experiments at module level
- 95% of the requirements can be verified with the tool to achieve a cost reduction of 50% to 60% in terms of man-hours

Goals

Projects *vs* Goals

Year	Project	Technologies (Full or Partial Adoption)	Goal
2007-2008	Project 1	Modelling guidelines (25) Code generation (Stateflow Coder R2007b)	1
2008-2010	Project 2	Modelling guidelines + MAAB (43) Code generation (Stateflow Coder R2007b) Guidelines verification Model-based testing Abstract interpretation (Polyspace 7.0)	1 2 3
2009-2011	Project 3	Modelling guidelines + MAAB (43) Semantics restrictions UML + hierarchical derivation Code generation (RTW Embedded Coder R2010a) Translation Validation Abstract interpretation (Polyspace 8.0) Formal Verification (Simulink Design Verifier R2010a)	1 2 3

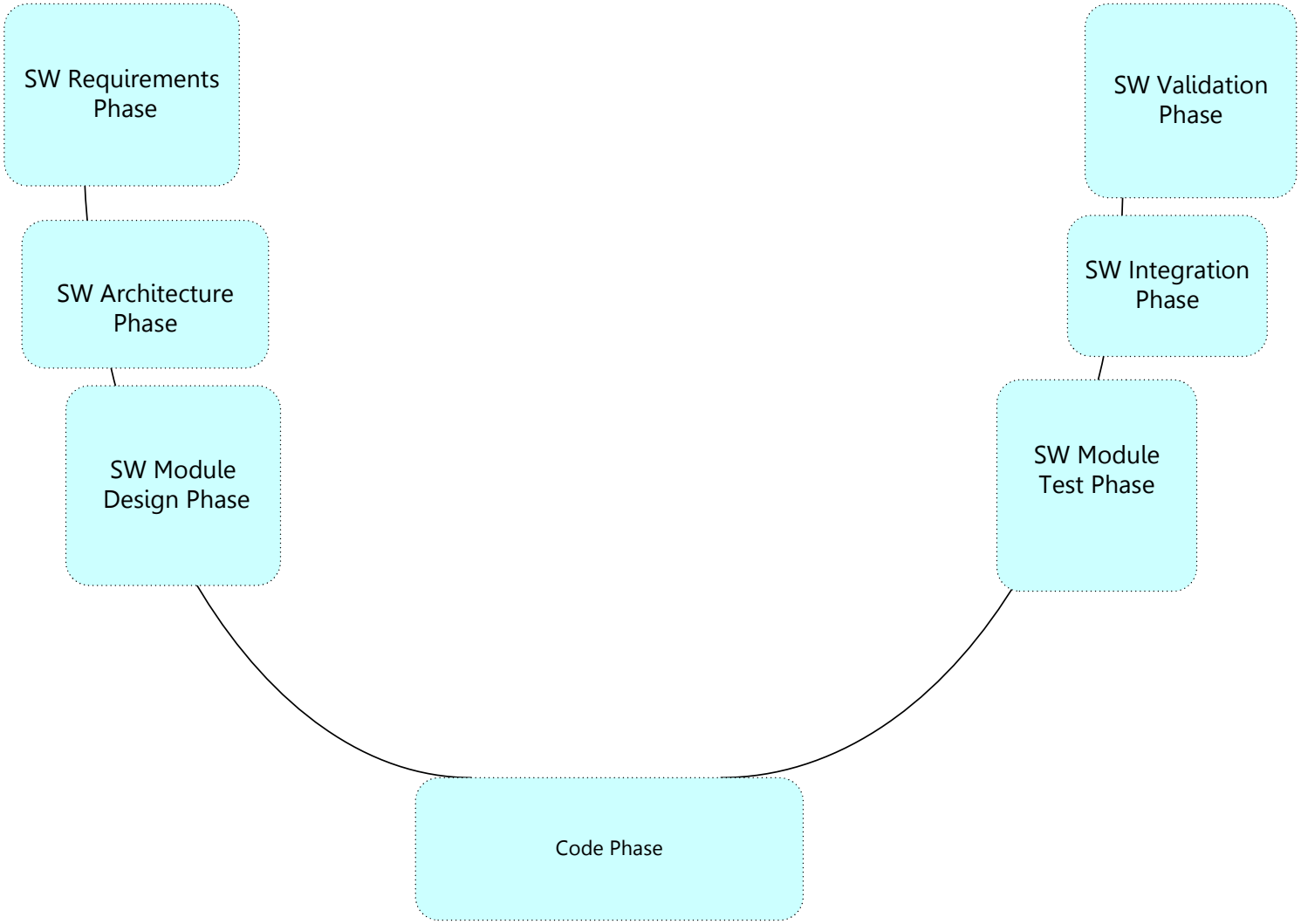
Goal 1 - Modelling Language Restriction

- **Project 1 - Identify a proper subset of the language**
  - Analysis of the violations of the quality standard issued by the code generated from the original model
  - Definition of sub-models and evaluation of the translation of single graphical constructs
  - Definition of a preliminary set of guidelines
- **Project 2 - A more general set of guidelines**
  - Previous guidelines could lack of generality since they were derived from a specific model
  - A comparison with the experience of other safety-critical domains was needed $\longrightarrow$ MAAB
  - Guidelines oriented also to define well-structured models
- **Project 3 - Enable formal analysis and verification**
  - Reduction of the Simulink/Stateflow language to a semantically unambiguous set
  - Inspired by the translation of Simulink/Stateflow into Lustre
  - The models produced are independent from the simulation engine

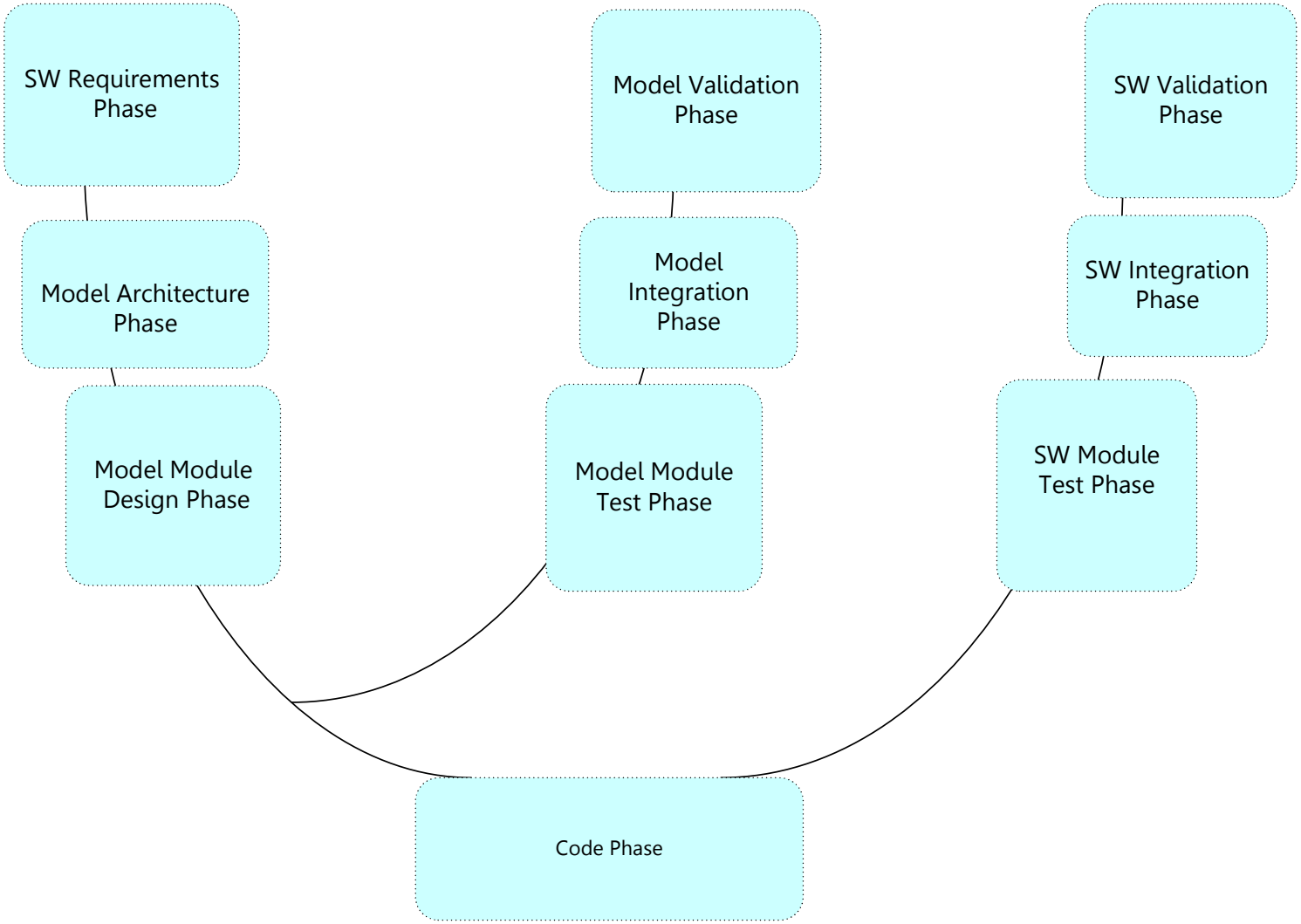
Goal 2 - Generated Code Correctness

- **Project 2 - Preliminary definition of a verification approach**
- **Project 3 - Operational implementation of the approach**
  - Translation validation
    - ★ Model/code back-to-back execution of unit tests
    - ★ Comparison of the structural coverage obtained at model and at code level
  - Abstract interpretation
    - ★ Verification of runtime errors
    - ★ The tools implementing abstract interpretation work on a conservative and sound approximation of the variable values in terms of intervals
    - ★ Finding errors in this larger approximation domain does not imply that the bug also holds in the program
    - ★ Problem of false positive cases
    - ★ First step: large over-approximation set, in order to discover systematic runtime errors and identify classes of possible false positives
    - ★ Second step: constrained abstract domain, derived from the first analysis, and the number of uncertain failure states to be manually reviewed is drastically reduced

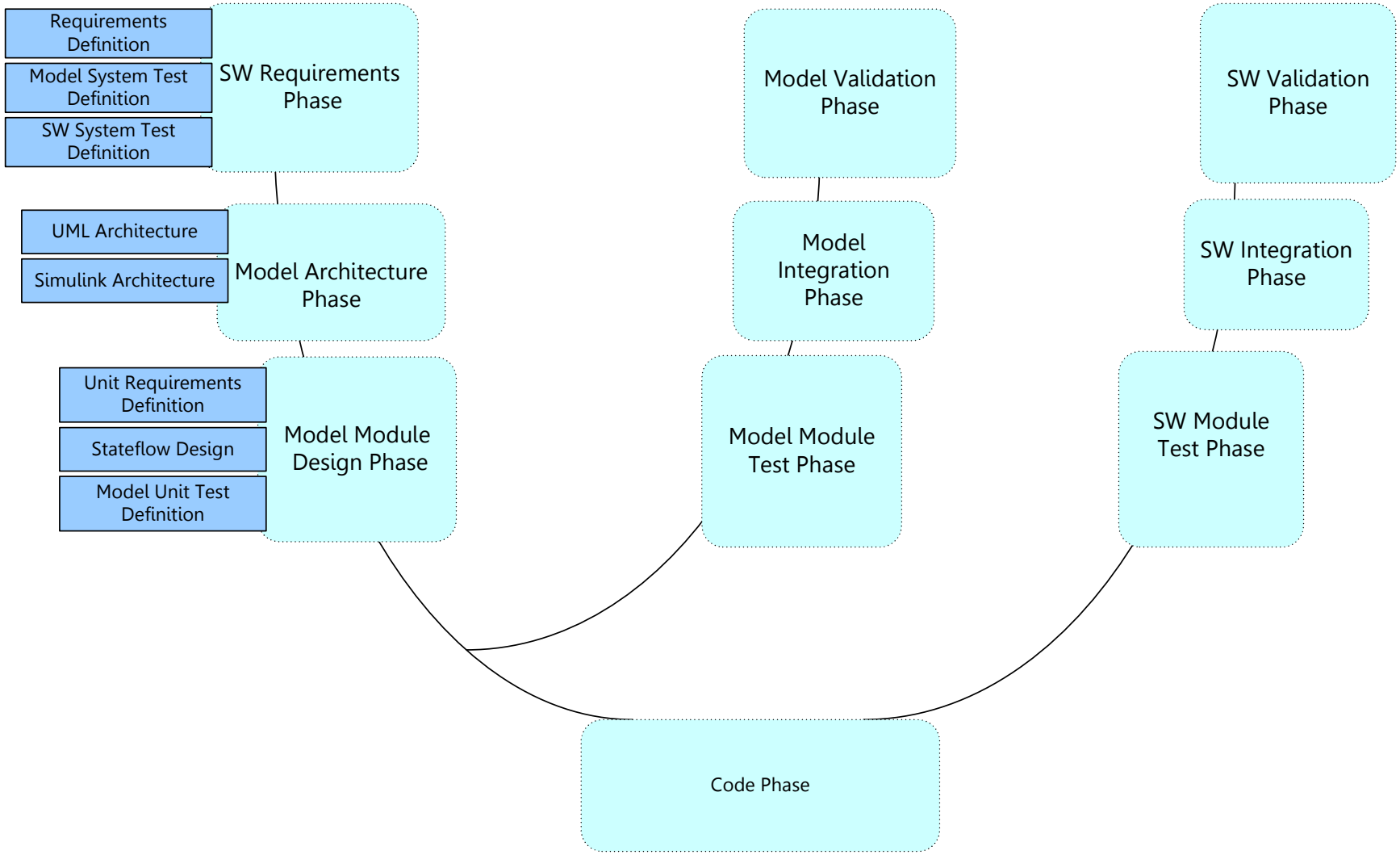
Goal 3 - Process Integration



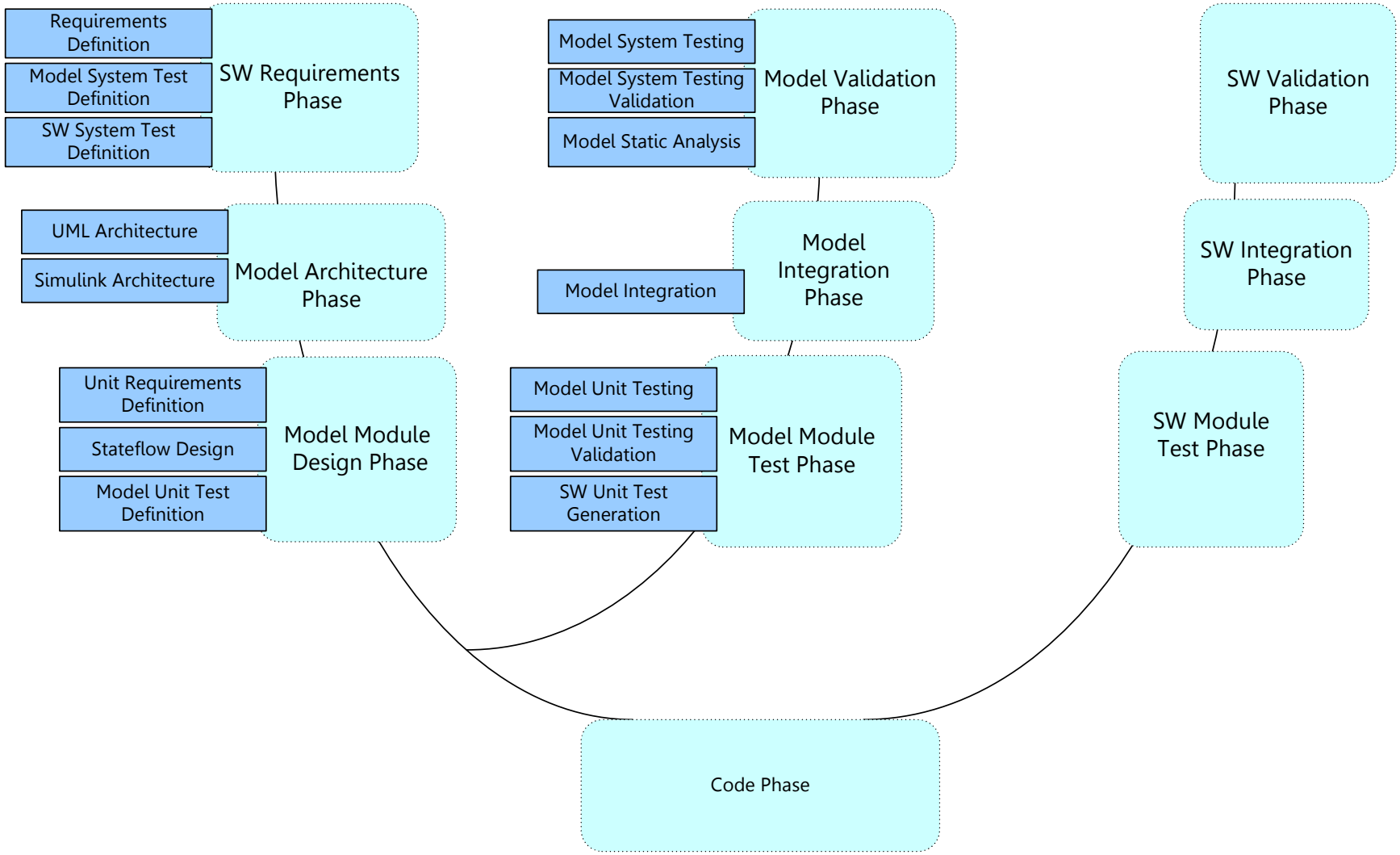
Goal 3 - Process Integration



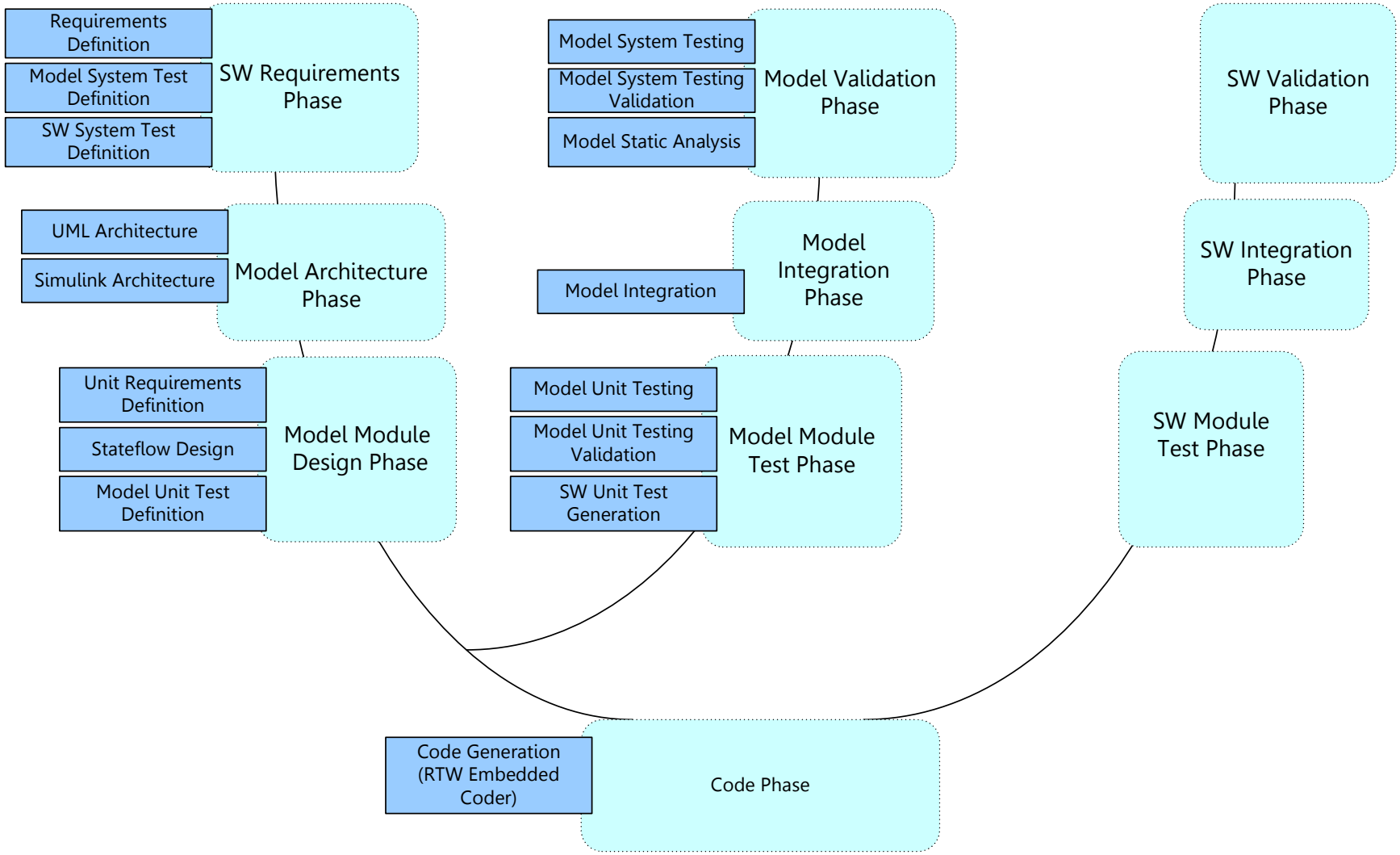
Goal 3 - Process Integration



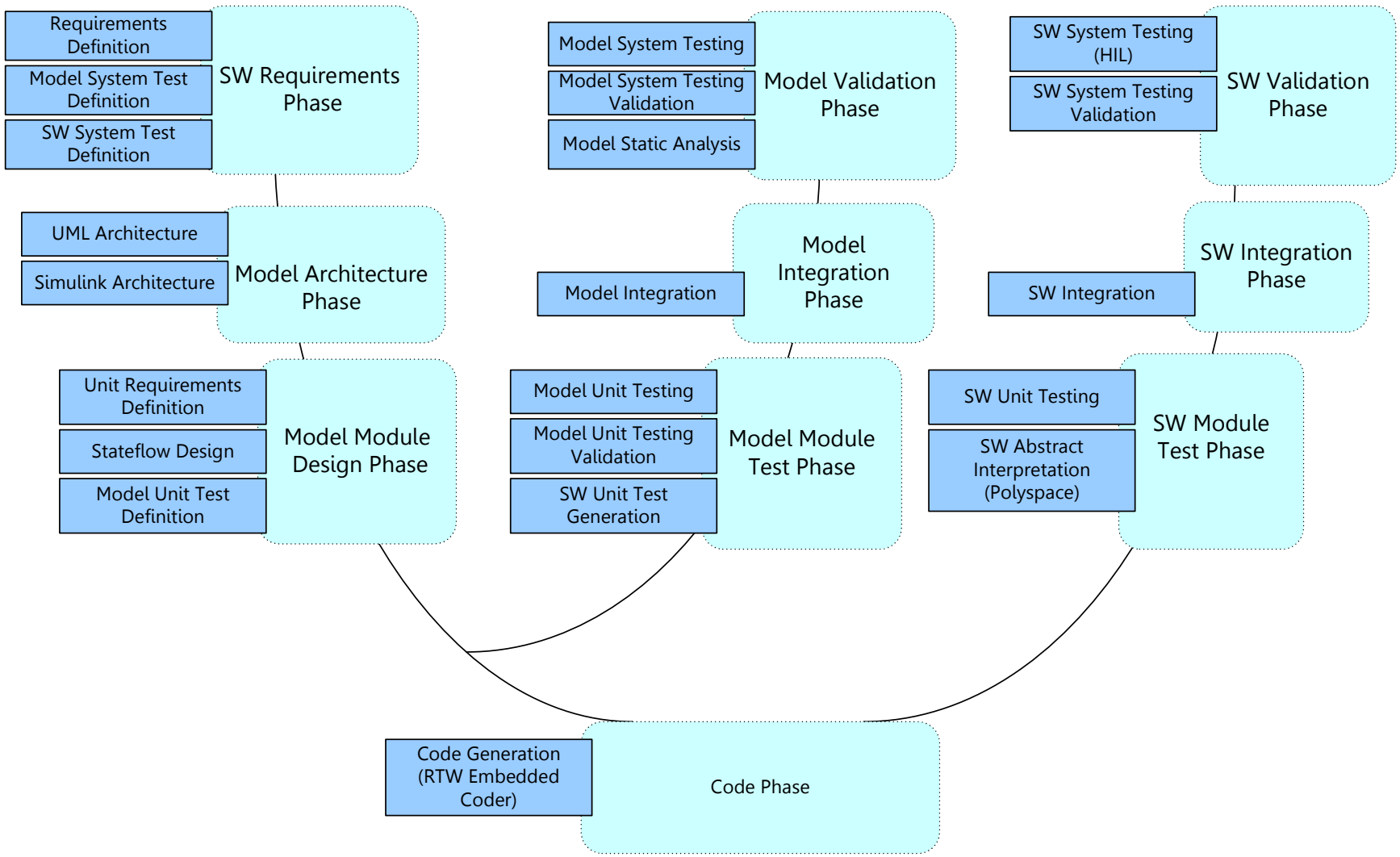
Goal 3 - Process Integration



Goal 3 - Process Integration



Goal 3 - Process Integration



What We Learnt

What We Learnt

- Abstraction
  - Models can be manipulated better than code
  - Definition of test scenarios at module level without disrupting model structure
- Expressiveness
  - Graphical models are closer to the natural language requirements
  - Unambiguous mean to exchange or pass artefacts among developers
- Cohesion & Decoupling
  - Interfaces among functionalities are based solely on data
  - Control-flow is simplified since there is no cross-call among different modules
- Uniformity
  - Generated code has a repetitive structure, which facilitates the automation of the verification activities
  - One could look at the generated code as if it would be the software always written by the same programmer

What We Learnt

- Traceability
  - Software modules are directly traceable with the corresponding blocks of the modelled specification
  - Navigable links between the single code statements and the requirements
- Control
  - Greater control over the components
  - Software with less bugs already before the verification activities
- Verification Cost
  - When passing from traditional code unit testing based on *structural* coverage objectives, to testing based on *functional* objectives aided with abstract interpretation, it was possible to reduce the verification cost of about 70%
  - Recent experiments with formal verification have shown that this cost can be further reduced by 50-66%

What We Learnt

- Performance
  - The complexity of the generated code is higher
  - Optimization are not allowed for safety-critical software
  - More powerful hardware platforms are required
- Manual Test Definition
  - Bottleneck of the verification process
  - 60-70% of the overall unit-level verification cost
  - Formal verification can address this issue
- Knowledge Transfer Process
  - Research assistant focused on the new technology
  - Development team putting into practice the technology
  - Balance between independence and participation

Conclusion

- Introducing the formal design and code generation technologies within the development process of a railway signalling manufacturer
- Radical changes in the verification process
- Formal model-based design has opened the door to model-based testing, has facilitated the adoption of abstract interpretation, and has allowed performing the first successful experiences with formal verification
- Four years and three projects to be defined and consolidated
- Incremental tuning of the process also thanks to the flexibility of the toolsuite
- Research management model has been crucial
- What if UML-centered tools are going to be used?