Curriculum for

Certified Professional for
Software Architecture (CPSA)®
Advanced Level

**Module
FM**

Formal Methods

2024.1-rev0-EN-20250116

Table of Contents

List of Learning Goals

- LG 1-1: Propositional Logic
- LG 1-2: Predicate Logic
- LG 1-3: Temporal Logic
- LG 1-4: Logical Calculi
- LG 1-5: Intuitionistic vs. Classical Logic
- LG 2-1: Notions of specification
- LG 2-2: Formal Specifications
- LG 2-3: Specification Languages
- LG 2-4: Refinement
- LG 3-1: Applicability
- LG 3-2: Relevant Qualities
- LG 3-3: Relevance Specification
- LG 3-4: Tradeoffs
- LG 3-5: Gradual Introduction
- LG 3-6: Appropriate Selection of Methods
- LG 3-7: Architecture Evaluation
- LG 4-1: Property-Based Testing
- LG 4-2: Type Systems
- LG 4-3: Model Checking
- LG 4-4: Proof Assistants
- LG 4-5: SMT Solvers
- LG 4-6: Abstract Interpretation

Introduction: General information about the iSAQB Advanced Level

What is taught in an Advanced Level module?

- The iSAQB Advanced Level offers modular training in three areas of competence with flexibly designable training paths. It takes individual inclinations and priorities into account.

- The certification is done as an assignment. The assessment and oral exam is conducted by experts appointed by the iSAQB.

What can Advanced Level (CPSA-A) graduates do?

CPSA-A graduates can:

- Independently and methodically design medium to large IT systems

- In IT systems of medium to high criticality, assume technical and content-related responsibility

- Conceptualize, design, and document actions to achieve quality requirements and support development teams in the implementation of these actions

- Control and execute architecture-relevant communication in medium to large development teams

Requirements for CPSA-A certification

- Successful training and certification as a Certified Professional for Software Architecture, Foundation Level® (CPSA-F)

- At least three years of full-time professional experience in the IT sector; collaboration on the design and development of at least two different IT systems

  ◦ Exceptions are allowed on application (e.g., collaboration on open source projects)

- Training and further education within the scope of iSAQB Advanced Level training courses with a minimum of 70 credit points from at least three different areas of competence

- Successful completion of the CPSA-A certification exam

CPSA-A®
Certified Professional for Software Architecture
Advanced Level

Essentials

What does the module "FM" convey?

Among the responsibilities of a software architect is making sure that the architecture design is correctly refined and implemented, meaning that the implementation is architecturally compliant. On one hand, the implemented system should conform to the system's software architecture, i.e. the interfaces, and the relationships defined in the software architecture. On the other hand, the models that describe the system functionality should be correct regarding the system requirements.

Correctness seems self-evident as a requirement, but it rarely makes it into explicit design documents or design artifacts. Often, correctness is not directly applicable to the "messy" aspects of a software system which may have co-evolved with vaguely formulated requirements, where correctness is in the eye of the beholder. Yet some application domains require correctness of certain aspects of a system implementation with respect to its requirement specification. Examples include software applications that:

- control mission-critical hardware,

- secure highly sensitive pieces of information, or

- make revenue-vital calculations.

Traditional architecture focuses on producing a workable and maintainable system. Its functionality is assured through manual automated testing, which only operates on individual examples. These can, as the old adage goes, only show the presence, but not the absence of errors: Properties underlying the tests are not typically constructed systematically, and only a small part of the state space is covered.

Thus, traditional software architecture and development methods are insufficient to ensure correctness as in the examples above. This curriculum intends to provide a collection of formal methods to supplement and replace the traditional architect's arsenal. These methods produce mathematical proofs of critical system properties. Note that such proof can not typically be produced as an afterthought to a system's architecture. Instead, the architecture needs to be designed from the start to be amenable to such proof. While designing and building systems that are amenable to various flavors of automated testing – such as unit, integration, acceptance, and property testing – is established practice and well-supported by common technology stacks, formal methods require a substantially higher effort to incorporate in a design. For instance, establishing a formal connection between specification and implementation requires a careful selection of specification and programming languages, respectively.

To apply formal methods, architects need to formulate important properties of the software system mathematically, construct an architecture capable of ensuring these properties, and then proceed to verify them formally. Creating an architecture fit for verification requires careful consideration, and a high degree of architectural competency. In particular, architects need to:

- choose appropriate properties to verify,

- create verifiable models for the whole system under analysis as well as for its components,

- partition the system into verifiable components, and ensuring that the component properties are composable into system properties, which in turn are verified at system level

- choose appropriate formal techniques and tools to verify the properties

- integrate verification into the software development process and tool chain

Software architects can employ these skills and techniques, particularly in early phases of the design of

the system architecture. Formal verification methods are not only used to show correctness of architecture models regarding specific system requirements, their application can, in fact, be guided and directed by the functional system architecture itself. Relatedly, a strong focus on formal methods in system design may feed into better "traditional" testability, for example by the ability to automatically generate test cases, drastically increasing software quality while simultaneously increasing development.

Curriculum Structure and Recommended Durations

Content	Recommended minimum duration (minutes)
1. Logic	180
2. Specification and Implementation	300
3. Formal Methods and the Development Process	120
4. Tools	300
5. Examples	180
Total	1080 (18h)

Duration, Teaching Method and Further Details

The times stated below are recommendations. The duration of a training course on the FM module should be at least **3** days, but may be longer. Providers may differ in terms of duration, teaching method, type and structure of the exercises, and the detailed course structure. In particular, the curriculum provides no specifications on the nature of the examples and exercises.

Licensed training courses for the FM module contribute the following credit points towards admission to the final Advanced Level certification exam:

Methodical Competence:	**10** Points
Technical Competence:	**10** Points
Communicative Competence:	**10** Points

Prerequisites

Participants **should** have the following prerequisite knowledge:

- basic knowledge of algebra
- basic knowledge of logic

Knowledge in the following areas may be **helpful** for understanding some concepts:

- functional programming
- equational reasoning over programs
- programming-language semantics

Structure of the Curriculum

The individual sections of the curriculum are described according to the following structure:

- **Terms/principles**: Essential core terms of this topic.

- **Teaching/practice time**: Defines the minimum amount of teaching and practice time that must be spent on this topic or its practice in an accredited training course.

- **Learning goals**: Describes the content to be conveyed including its core terms and principles.

This section therefore also outlines the skills to be acquired in corresponding training courses.

Supplementary Information, Terms, Translations

To the extent necessary for understanding the curriculum, we have added definitions of technical terms to the iSAQB glossary and complemented them by references to (translated) literature.

1. Logic

Duration: 120 min	Practice time: 60 min

1.1. Terms and Principles

formal system, logic, propositional logic, predicate logic, temporal logic, intuitionistic logic, classical logic, syntax, semantics, conjunction, disjunction, implication, quantifier, logical calculus, natural calculus, sequent calculus, deduction, inference, resolution

1.2. Learning Goals

LG 1-1: Propositional Logic

Know the basic structure of propositional logic:

- (atomic) propositions and their syntax

- logical connectives such as conjunction

- Conjunctive and disjunctive normal forms

- Semantics of propositions

- Meta-logical concepts of models, validity, satisfiability, and equivalence

- Decidability of propositional logic

LG 1-2: Predicate Logic

Know the basic structure of predicate/first-order logic:

- terms and their syntactic structure (variables, functions)

- formulas and their syntactic structure (predicates, quantifiers)

- predicate logic as an extension of propositional logic

- Non-trivial syntactic operations on formulas

  - renaming and substitution, avoiding variable capture

  - skolemization and equisatisfiability

  - quantifier elimination

- Semantics of first-order formulas

  - structures comprising universes and interpretations

  - coincidence of structures

  - extension with equality

- semi-decidability of predicate logic

LG 1-3: Temporal Logic

Know the basic structure of temporal operators:

- eventually and forever

- Different logics and interpretations, for example LTL vs. CTL
- Connection to automata

LG 1-4: Logical Calculi

Understand the basic concepts of a logical calculus:

- Inference rules operating on syntactic structure of a formula
- Different characteristics regarding supported fragments and runtime complexity
- Completeness vs. refutation completeness
- Natural deduction
- Sequent calculus
- Resolution

LG 1-5: Intuitionistic vs. Classical Logic

Understand the difference between intuitionistic and classical logic:

- Constructive vs. non-constructive proofs
- Axioms and inferences only admissible in classical logic (LEM, double negation elimination)
- Correspondence of intuitionistic logic to programmign and type systems

1.3. References

[Schöning 2008], [Troelstra and Schwichtenberg 2012], [Harrison 2009], [Fitting 1996], [Enderton 2001], [Ebbinghaus et al. 2021], [Gallier 2015]

2. Specification and Implementation

Duration: 200 min	Practice time: 100 min

2.1. Terms and Principles

specification, property, formal specification, mechanized specification, specification language, refinement, model

2.2. Learning Goals

LG 2-1: Notions of specification

- Know different notions of specification, specifically:
  - examples
  - properties
  - formal
  - mechanized
- Understand that specification may pertain to different kinds of components, such as:
  - functions
  - data types
  - algorithms
  - systems
- Understand that specifications may pertain to different qualities, such as:
  - functionality
  - performance effiicency
  - security
  - safety

LG 2-2: Formal Specifications

- Understand that applying formal methods requires formal specifications.
- Understand that formal specifications involve universal quantifiers rather than just examples for desired behavior.
- Understand the difference between functional properties that admit only one output per input, and relational properties that admit multiple outputs per input.

LG 2-3: Specification Languages

- Know that many different specification languages exist, among them some with tool support.
- Know that at least three such languages, and their distinguishing properties.

LG 2-4: Refinement

- Understand the notion of refinement.

- Understand that it may be helpful to develop a separate model between specification and implementation, which satisfies the specification and is equivalent to the implementation.

2.3. References

[ISO 24765], [Knuth 1997], [Milner 1973], [Nipkow 2014], [Paulson 1993]

3. Formal Methods and the Development Process

Duration: 75 min	Practice time: 45 min

3.1. Terms and Principles

SPE model, expressiveness, static typing, property-based testing

3.2. Learning Goals

LG 3-1: Applicability

- Know that not all parts of a typical software system are susceptible to formal methods.

- Know taxonomies like the SPE model that classify software according to their degree of precise specification, and to use the taxomonies for identifying the parts of a software system amenable to formal methods.

LG 3-2: Relevant Qualities

- Know the qualities that typically indicate the use of formal methods, such as reliability, safety, and security.

LG 3-3: Relevance Specification

- Know that the successful application of formal methods depends on the precise specification of the properties they require the software system to exhibit.

LG 3-4: Tradeoffs

Know the tradeoffs inherent in the choice of a particular formal method, in particular with respect to:

- expressiveness of its specification formalism

- effort needed to assist the formal method in establishing the desired result

- the qualificiations required of those who apply the method.

LG 3-5: Gradual Introduction

- Know techniques for gradually introducing formal methods into a software project such as adding static typing or property-based testing.

LG 3-6: Appropriate Selection of Methods

Be able to choose an appropriate formal method depending on the form of the specification such as:

- use (probalistic) model checking for specifications of state machines

- use SMT/SAT solving for constraint systems

- abstract interpretation for technical properties of the code

- abstract interpretation for properties of the technical architecture

- proof assistants for more general specifications

(See section Tools.)

LG 3-7: Architecture Evaluation

Understand that formal methods can support architecture evaluation:

- support quantitative architecture evaluation techniques

- support in the architecture evaluation process to deal with architectural topics uncovered with qualitative evaluation (e. g., review)

- using SMT/SAT solving for constraint problems for continuous architecture refinement (e. g., application to scheduling theory to address Performance Efficiency)

- using abstract interpretation for continuous architecture refinement (e. g., application to architecture and code to address Performance Efficiency and Security)

3.3. References

[Aniculaesei et al. 2018], [Aniculaesei et al. 2021], [Ball 2000], [Boca et al. 2009], [Brinkmann et al. 2018], [Drechsler 2018], [Gnesi et al. 2013], [Klein et al. 2009], [Kuper 2017 a], [Kuper 2017 b], [Lehman 1980], [Leroy 2009], [Merz et al. 2008]

4. Tools

Duration: 150 min	Practice time: 150 min

4.1. Terms and Principles

property-based testing, type system, dependent types, model checking finite automat0on, binary decision diagram (BDD), computation tree logic (CTL), propositional linear temporal logic (PLTL), proof assistant, SMT solver, abstract interpretation, static analysis

4.2. Learning Goals

LG 4-1: Property-Based Testing

- Understand the concept of property-based testing, can identify for which properties it is amenable, and guide the implementation of property-based testing in a software project.

LG 4-2: Type Systems

- Understand that type systems can express and ensure properties of software system.

- Know of the expressive power of different type systems.

- Know the concept of dependent types.

- Can choose languages and tools to exploit the capabilities of a type system in a software system.

LG 4-3: Model Checking

- Understand that Model Checking describes methods to check whether a logic formula is satisfied by a given model.

- Understand that a model is represented explicit, for example as finite automata, or symbolic, for example when using Binary Decision Diagrams (BDD).

- Understand that various kinds of logic can be used, like:

  - Computation Tree Logic (CTL)

  - Propositional Linear Temporal Logic (PLTL).

- Know about the consequences of the used representation and logic with respect to:

  - complexity (e.g. space and time costs, state explosion, combinatorial explosions)

  - usability

  - limitations

LG 4-4: Proof Assistants

- Understand the mode of operation of proof assistants.

- Know at least one proof assistant.

- Understand how to structure a software system to make it amenable to the use of proof assistants.

- Understand how they can translate requirements on a software projects into proof obligations for a proof assistant.

LG 4-5: SMT Solvers

- Understand the concept of satisfiability modulo theories (SMT) and its application and constraints as a tool for checking properties of software systems.

- Can identify which properties can be checked with the help of SMT solvers.

- Guide the implementation of SMT-assisted computations in a software project.

LG 4-6: Abstract Interpretation

- Understand that the possible dynamic behavior of programs can be soundly predicted using static analysis.

- Understand that the abstract interpretation technique for static analysis operates by executing the program over domains that are abstracted from the domains of the regularly-running program.

- Know the kinds of information that abstract interpretation might yield, such as pointer aliasing, resource usage, control and data flow.

4.3. References

[Claessen and Hughes 2002], [Chlipala 2022], [Nipkow 2014], [Kaufmann et al. 2000], [Stump 2016], [Bove et al. 2009], [Cousot 2021]

5. Examples

Duration: 120 min	Practice time: 60 min

This section is not examinable.

5.1. Terms and Principles

In every licensed training session, at least one example for the application opf formatl methods must be presented.

Type and structure of the examples presented may depend on the training and participants' interests. They are not prescribed by iSAQB.

References

A

- [Aniculaesei et al. 2021] Aniculaesei Adina, Vorwald, Andreas, Zhang, Meng, and Rausch, Andreas. Architecture-based hybrid approach to verify safety-critical automotive system functions by combining data-driven and formal methods. In Cyrille Artho and Rudolf Ramler, editors, 2021 IEEE International Conference on Software Architecture Companion (ICSA-C), pages 139–148. IEEE, 2021.

- [Aniculaesei et al. 2018] Aniculaesei Adina, Howar, Falk, Denecke, Peer, and Rausch Andreas. Automated generation of requirements-based test cases for an adaptive cruise control system. In Cyrille Artho and Rudolf Ramler, editors, 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST@SANER), pages 11–15. IEEE, 2018

B

- [Baader et al. 2012] Baader F., Nipkow T. (2012). Term Rewriting and All That. Cambridge University Press. https://doi.org/10.1017/CBO9781139172752

- [Ball 2000] Ball Thomas and Rajamani Sriram: Checking Temporal Properties of Software with Boolean Programs, Workshop on Advances in Verification, 2000.

- [Barrett et al. 2018] Barrett, C., Tinelli, C. (2018). Satisfiability Modulo Theories. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds) Handbook of Model Checking. Springer, Cham. https://doi.org/10.1007/978-3-319-10575-8_11

- [Berard et al. 2001] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., McKenzie, P. (2001): Systems and Software Verification - Model-Checking Techniques and Tools. Springer.

- [Biere et al. 2021] Biere, A., Heule, M., Van Maaren, H., Walsh, T. (2021). Handbook of Satisfiability (Second Edition): Volume 336 Frontiers in Artificial Intelligence and Applications. IOS Press.

- [Boca et al. 2009] Boca P., Bowen J. P., Siddiqi J. (2009). Formal Methods: State of the Art and New Directions. Springer London. https://doi.org/10.1007/978-1-84882-736-3

- [Bove et al. 2009] Ana Bove, Peter Dybjer and Ulf Norell (2009) . A Brief Overview of Agda - A Functional Language with Dependent Types. International Conference on Theorem Proving in Higher Order Logics.

- [Brinkmann et al. 2018] Brinkmann, R., Kelf, D. (2018). Formal Verification—The Industrial Perspective. In: Drechsler, R. (eds) Formal System Verification. Springer, Cham. https://doi.org/10.1007/978-3-319-57685-5_5

C

- [Chlipala 2022] Adam Chlipala (2022). Certified Programming with Dependent Types. MIT Press, 2022.

- [Clarke et al. 2018] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, Roderick Bloem (2018). Handbook of Model Checking. Springer, Cham. https://doi.org/10.1007/978-3-319-10575-8

- [Cousot 2021] Patrick Cousot. Principles of Abstract Interpretation. MIT Press, 2021.

D

- [Drechsler 2018] Drechsler, R. (2018). Formal System Verification - State-of the-Art and Future Trends. Springer Cham. https://doi.org/10.1007/978-3-319-57685-5

E

- [Ebbinghaus et al. 2021] Ebbinghaus Heinz-Dieter, Flum Jörg, Thomas Wolfgang (2021). Mathematical Logic. Springer. https://doi.org/10.1007/978-3-030-73839-6

- [Enderton 2001] Enderton, Herbert B. (2001). A Mathematical Introduction to Logic. Academic Press.

F

- [Fitting 1996] Fitting, Melvin (1996). First-Order Logic and Automated Theorem Proving. Springer. https://doi.org/10.1007/978-1-4612-2360-3

G

- [Gallier 2015] Gallier Jean H. (2015). Logic for Computer Science. Dover Publications.

- [Gnesi et al. 2013] Gnesi, S. Margaria, T. (2013). Formal Methods for Industrial Critical Systems: A Survey of Applications. IEEE. https://doi.org/10.1002/9781118459898

- [Grumberg et al. 2012] Grumberg, O., Nipkow, T., Hauptmann, B. (2012). Software Safety and Security: Tools for Analysis and Verification: Volume 33 of NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press.

H

- [Harrison 2009] Harrison, John (2009). Handbook of Practical Logic and Automated Reasoning. Cambridge University Press. https://doi.org/10.1017/CBO9780511576430

I

- [ISO 13568] ISO/IEC 13568:2002 Information technology — Z formal specification notation — Syntax, type system and semantics.

- [ISO 24765] ISO/IEC/IEEE 24765:2017. Systems and software engineering.

H

- [Claessen and Hughes 2002] Koen Claessen, John Hughes (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP).

K

- [Kaufmann et al. 2000] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore (2000). Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers. https://www.cs.utexas.edu/users/moore/publications/acl2-books/car/index.html

- [Klein et al. 2009] Klein, Gerwin, Elphinstone, Kevin, Heiser, Gernot, Andronick, June, Cock, David, Derrin, Philip, Elkaduwe, Dhammika, Engelhardt, Kai, Kolanski, Rafal, Norrish, Michael, Sewell, Thomas, Tuch, Harvey and Winwood, Simon. seL4: formal verification of an OS kernel. SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. October 2009, pages 207–220.

- [Knuth 1997] Knuth, D. (1997) The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3rd edition. Addison-Wesley. https://www-cs-faculty.stanford.edu/~knuth/taocp.html

- [Kroening et al. 2017] Kroening D., Strichman O. (2017): Decision Procedures - An Algorithmic Point of View. Springer Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-50497-0

- [Kuper 2017 a] Kuper, L. (2017): Proving that safety-critical neural networks do what they're supposed to: where we are, where we're going (part 1 of 2). https://decomposition.al/blog/2017/05/30/proving-that-safety-critical-neural-networks-do-what-theyre-supposed-to-where-we-are-where-were-going-part-1-of-2/

- [Kuper 2017 b] Kuper, L. (2017): Proving that safety-critical neural networks do what they're supposed to: where we are, where we're going (part 2 of 2). https://decomposition.al/blog/2017/05/31/proving-that-safety-critical-neural-networks-do-what-theyre-supposed-to-where-we-are-where-were-going-part-2-of-2/

L

- [Lamport 2022] Leslie Lamport (2022). Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley.

- [Lehman 1980] Meir Lehman (1980). Programs, Life Cycles, and Laws of Software Evoluation. Proceedings of the IEEE, Volume: 68, Issue: 9, September 1980.

- [Leroy 2009] Xavier Leroy: Formal verification of a realistic compiler, Communications of the ACM 52(7) 2009, pages 107—115.

M

- [Marques-Silva et al. 2018] Marques-Silva J., Malik S. (2018). Propositional SAT Solving. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds) Handbook of Model Checking. Springer, Cham. https://doi.org/10.1007/978-3-319-10575-8_9

- [Merz et al. 2008] Merz S., Navet N. (2008). Modeling and Verification of Real-Time Systems: Formalisms and Software Tools. ISTE Ltd. https://doi.org/10.1002/9780470611012

- [Milner 1973] Milner, R. (1973) Models of LCF. Stanford Artifical Intelligence Laboratory Memo AIM-186.

N

- [Nipkow 2014] Nipkow, T., Klein, G. (2014) Concrete Semantics with Isabelle/HOL. Springer. http://www.concrete-semantics.org/

P

- [Paulson 1993] Paulson, Lawrence C. (1993). Isabelle: The Next 700 Theorem Provers. CoRR, cs.LO/9301106. https://arxiv.org/abs/cs/9301106

S

- [Schöning 2008] Schöning, Uwe (2008). Logic for Computer Scientists. Birkhäuser Boston. https://doi.org/10.1007/978-0-8176-4763-6

- [Stump 2016] Aaron Stump (2016). Verified Functional Programming in Agda. ACM.

T

- [Troelstra and Schwichtenberg 2012] Troelstra A. S., Schwichtenberg H. (2012). Basic Proof Theory.

Cambridge University Press. https://doi.org/10.1017/CBO9781139168717

W

- [Wayne 2018] Hillel Wayne (2018). Practical TLA+: Planning Driven Development. Apress.