# 1 Introduction

**The method.** This book introduces a systems engineering method which guides the development of software and embedded hardware–software systems seamlessly from requirements capture to their implementation. It helps the designer to cope with the three stumbling-blocks of building modern software based systems: size, complexity and trustworthiness. The method bridges the gap between the human understanding and formulation of real-world problems and the deployment of their algorithmic solutions by code-executing machines on changing platforms. It covers within a single conceptual framework both *design and analysis*, for procedural single-agent and for asynchronous multiple-agent distributed systems. The means of analysis comprise as methods to support and justify the reliability of software both *verification*, by reasoning techniques, and experimental *validation*, through simulation and testing.

The method *improves current industrial practice* in two directions:

– On the one hand by accurate high-level modeling at the level of abstraction determined by the application domain. This raises the level of abstraction in requirements engineering and improves upon the loose character of human-centric UML descriptions.
– On the other hand by linking the descriptions at the successive stages of the system development cycle in an organic and effectively maintainable chain of rigorous and coherent system models at stepwise refined abstraction levels. This fills a widely felt gap in UML-based techniques.

Contrary to UML, the method has a *simple scientific foundation*, which adds precision to the method's practicality. Within the *uniform conceptual framework* offered by the method one can consistently relate standard notions, techniques and notations currently in use to express specific system features or views, each focussed on a particular system aspect, such as its structure, environment, time model, dynamics, deployment, etc. (see Sect. 7.1). Thereby the method supports a *rigorous integration of common design, analysis and documentation techniques* for model reuse (by instantiating or modifying the abstractions), validation (by simulation and high-level testing), verification (by human or machine-supported reasoning), implementation and maintenance (by structured documentation). This improves upon the loose ties between different system design concepts as they are offered by the UML framework.

**Target audience.** This book combines the features of a handbook and of a textbook and thus is addressed to hardware–software system engineers (architects, designers, program managers and implementers) and researchers as well as to students. As a handbook it is conceived as a Modeling Handbook for the Working Software Engineer who needs a practical high-precision design instrument for his daily work, and as a Compendium for Abstract State Machines (ASMs). As a textbook it supports both self-study (providing numerous exercises) and teaching (coming with detailed lecture slides in ppt and/or pdf format on the accompanying CD and website `http://www.di.unipi.it/AsmBook/`). We expect the reader to have some experience in design or programming of algorithms or systems and some elementary knowledge of basic notions of discrete mathematics, e.g. as taught in introductory computer science courses. Although we have made an effort to proceed from simple examples in the earlier chapters to more complex ones in the later chapters, all chapters can be read independently of each other and unless otherwise stated presuppose only an understanding of a rather intuitive form of abstract pseudo-code, which is rigorously defined as ASM in Sect. 2.2.2. We have written the text to enable readers who are more interested in the modeling and less in the verification aspects to skip the proof sections.[1] The hurried reader may skip the numerous footnotes where we refer to interesting side issues or to related arguments and approaches in the literature.

There is another book through which the reader can learn the ASM method explained in this book, namely [406], which contains the up-to-now most comprehensive non-proprietary real-life ASM case study, covering in every detail ground modeling, refinement, structuring, implementation, verification and validation of ASMs. The focus of that book however is an analysis of Java and its implementation on the Java Virtual Machine (including a detailed definition and analysis of a compiler and a bytecode verifier), as a consequence it uses only basic and turbo ASMs (see Sect. 2.2, 4.1). The present book is an *introduction* to practical applications of the ASM method via small or medium-size yet characteristic *examples from various domains*: programming languages, architectures, embedded systems, components, protocols, business processes. It covers also real-time and asynchronous ASMs. In addition it provides the *historical and* the *theoretical background* of the method. We hope this book stimulates further technological and research developments, ranging from industrial applications to theoretical achievements.

[1] Mentioning this possibility does not mean that we consider system verification as an optional. It reflects the support the method presented in this book provides to systematically *separate different concerns within a well-defined single framework* so that one can ultimately tie the different threads together to achieve a design which via its analysis is certifiable as trusted (see Sect. 2.1).

In various places we state some problems whose solution we expect to contribute to the further advancement of the ASM method. They are collected in a list at the end of the book.

In the rest of this introduction we state in more detail the practical and theoretical goals of the book and survey its technical contents.

## 1.1 Goals of the Book and Contours of its Method

Through this book we want to introduce the reader into a hierarchical modeling technique which

– makes accurate virtual machine models amenable to mathematical and experimental analysis,
– links requirements capture to detailed design and coding,
– provides on the fly a documentation which can be used for inspection, reuse and maintenance.

The open secret of the method is to use abstraction and stepwise refinement, which often are erroneously understood as intrinsically "declarative" or syntax-based concepts, on a *semantical* basis and to combine them with the *operational* nature of machines. Such a combination (Sect. 1.1.1) can be obtained by exploiting the notion of Abstract State Machines (Sect. 1.1.2) – which gave the name to the method – and results in considerable practical benefits for building trustworthy systems (Sect. 1.1.3). We also shortly describe here what is new in the ASM method with respect to the established method of stepwise refining pseudo-code (Sect. 1.1.4) and what it adds to UML based techniques (Sect. 1.1.5).

### 1.1.1 Stepwise Refinable Abstract Operational Modeling

The hardware and software system engineering method that this book introduces is based upon semantical abstraction and structuring concepts which resolve the tension deriving from the simultaneous need for *heterogeneity*, to capture the richness and diversity of application domain concepts and methods, and for *unification*, to guarantee a consistent seamless development throughout. In fact it allows one to efficiently relate the two distant ends of each complex system development effort, namely the initial problem description for humans and the code running on machines to solve the problem. More precisely, using this method the system engineer can

– derive from the application-domain-governed understanding of a given problem, gained through requirements analysis, a correct and complete human-centric task formulation, called the *ground model*, which is the result of the requirements capture process, is expressed in application-domain

terms, is easy to understand and to modify, and constitutes a binding contract between the application domain expert (in short: the customer) and the system designer,[2]
- refine the ground model by more detailed descriptions which result from the relevant design decisions, taken on the way to the executable code and documented by the intermediate models which typically constitute a *hierarchy of refined models*,
- link the most detailed specification to *generated code*, to be run on various platforms and implementing computers and to be shown to correctly solve the problem as formulated in the ground model (the contract with the customer).

The conceptual key for crossing these rather different and for complex software systems numerous levels of abstraction is to maintain, all the way from the ground model to the code, a *uniform algorithmic view*, based upon an *abstract notion of run*, whether of agents reacting to events or of a virtual machine executing sequences of abstract commands. Having to deal in general with sets of "local actions" of multiple agents, an encompassing concept of basic "actions" is defined as taking place in well-defined abstract local states (which may depend on environmently determined items) and producing well-defined next states (including updates of items in the environment). We interpret simultaneous basic local actions as a generalized form of Dijkstra's guarded commands [185]: under explicitly stated conditions they perform updates of finitely many "locations", which play the role of abstract containers for values from given domains of objects, considered at whatever given level of abstraction. Those objects residing in locations, together with the functions and relations defined on them, determine the abstract states the computation is about.[3] The simple and intuitive mathematical form we adopt to represent this idea of transformations of abstract states for the system engineering method explained in this book is the notion of Abstract State Machines (ASM).[4]

[2] This does not preclude evolution of the ground model during the development process. Ground models need to be developed "for change", but at each development stage *one* version of a well-defined ground model is maintained; see below.

[3] Object-oriented methods in general and UML in particular share this first-order logic "view of the world" as made up of "things" ("abstractions that are first-class citizens in a model" [69]) and their "relationships".

[4] The idea of using "abstract" state transformations for specification purposes is not new. It underlies the event driven version of the B method [5, 6] with its characteristic separation of individual assignments from their scheduling. It underlies numerous state machine based specification languages like the language of statecharts [271] or Lampson's SPEC [316], which besides parallelism (in the case of SPEC including the use of quantifiers in expressions) and non-determinism offer constructs for non-atomic (sequential or submachine) execution, see Chap. 4. It underlies the wide-spectrum high-level design language COLD (see Sect. 7.1.2). It also underlies rule-based programming, often characterized as repeated local-

### 1.1.2 Abstract Virtual Machine Notation

This book explains the three constituents of the *ASM method*: the notion of ASM, the ground model technique and the refinement principle. The concept of ASMs (read: pseudo-code or Virtual Machine programs working on abstract data as defined in Sects. 2.2, 2.4) offers what for short we call *freedom of abstraction*, namely the unconstrained possibility of expressing appropriate abstractions directly, without any encoding detour, to

– build ground models satisfying the two parties involved in the system contract, tailoring each model to the needs of the problem as determined by the particular application, which may belong to any of a great variety of conceptually different domains (see Sect. 2.1.1), and keeping the models simple, small and flexible (easily adaptable to changing requirements),
– allow the designer to keep control of the design process by appropriate refinement steps which are fine-tuned to the implementation ideas (see Sect. 2.1.2).

Most importantly ASMs support the practitioner in exploiting their power of *abstraction in terms of an operational system view* which faithfully reflects the natural intuition of system behavior,[5] at the desired level of detail and with the necessary degree of exactitude. The underlying simple mathematical model of both synchronous and asynchronous computation allows one to view a system as a set of cooperating idealized mathematical machines which step by step – where the chosen level of abstraction determines the power of a step – perform local transformations of abstract global states. Essentially each single machine (driven by an agent, also called a thread) can be viewed as executing pseudo-code on arbitrary data structures, coming with a clear notion of state and state transition. This empowers the designer to work at every level of development with an accurate yet transparent concept of system *runs* for modeling the dynamic system behavior, whether the execution is done mentally (for the sake of high-level analysis) or by real machines (for the sake of high-level testing of scenarios). The availability of the concept of a run at each level of abstraction provides the possibility of also modeling non-functional features, like performance or reliability, or run time inspection of metadata associated with components as offered by CORBA and COM. Due to the mathematical nature of the concepts involved, established structuring, validation and verification techniques can be applied to ASM models, supporting architectural structuring principles and providing platform and

ized transformations of shared data objects (like terms, trees, graphs), where the transformations are described by rules which separate the description of the objects from the calculations performed on them and on whose execution various constraints and strategies may be imposed.

[5] See the observation in [399, Sect. 2.4] that even the knowledge base of experts has an operational character and guarded command form: "in *this* situation do *that*", which is also the form of the ASM transition rules defined in Sect. 2.2.2.

programming language-independent executable models which are focussed on the application-domain-relevant problem aspects and lend themselves to reuse in a design-for-change context.

### 1.1.3 Practical Benefits

The need to improve current industrial software engineering practice is widely felt. To mention only a few striking examples: too many software projects fail and are canceled before completion or are not delivered on time or exceed their budget,[6] the energy spent on testing code is ever increasing and tends to represent more than half of the entire development cost, the number of errors found in complex software is often rather high, there is almost no software warranty whatsoever, but again and again the world is surprised by Trojan horses and security holes, etc.

The major benefit the ASM method offers to practitioners for their daily work is that it provides a simple precise *framework to communicate and document design ideas* and a *support for an accurate and checkable overall understanding* of complex systems. Using a precise, process-oriented, intuitive semantics for pseudo-code on arbitrary data structures, the developer can bind together the appropriate levels of abstraction throughout the entire design and analysis effort. This implies various concrete benefits we are going to shortly mention now.

First of all the ASM method supports *quality from the beginning* using hierarchical modeling, based on ground modeling and stepwise refinement coupled to analysis. The method establishes a discipline of development which allows structuring (into appropriate abstractions), verification and validation to become criteria for good design and good documentation.[7] By its conceptual simplicity and the ease of its use, the ASM method makes the quality of the models depend only on the expertise in the application or design domain and on the problem understanding, not on the ASM notation. This is the reason why we expect to contribute with this book to making the method become part of every development activity which aims at ensuring that the produced model has the desired properties (e.g. to satisfy the needs of the future users), instead of waiting until the end of the development process to let another team (or the users) remove bugs in the code.[8]

[6] Some figures from the Standish Group 1998: 9% out of the 175 000 surveyed software projects are delivered on time and under budget, 52% go over budget by an average of 18%, 31% are canceled before completion.

[7] A quote from the recommendations of the UK Defense Standard 00–54, 1999 (Requirements for Safety-Related Electronic Hardware in Defense Equipment): "A formally-defined language which supports mathematically-based reasoning and the proof of safety properties shall be used to specify a custom design." See `http://www.dstan.mod.uk/data/00/054/02000100.pdf`.

[8] We believe it to be mistaken to relegate the specification and verification work, if done at all, to separate so-called "formal methods" teams. The work of such

The freedom ASMs offer to model arbitrarily complex objects and operations directly, abstracting away from inessential details (e.g. of encoding of data or control structures), allows one to *isolate the hard part of a system* and to turn it into a precise model which exposes the difficulties of the system but is simple enough to be understood and satisfactorily analyzed by humans.

As a by-product such core models and their refinements yield valuable *system documentation*: (a) for the customers, allowing them to check the fulfillment of the software contract by following the justification of the design correctness, provided in the form of verified properties or of validated behavior (testing for missing cases, for unexpected situations, for the interaction of to-be-developed components within a given environment, etc.), (b) for the designers, allowing them to explore the design space by experiments with alternative models and to record the design rationale and structure for a checkable communication of design ideas to peers and for later reuse (when the requirements are changed or extended, but the design diagrams on the whiteboard are already erased or the developers are gone),[9] (c) for the users, allowing them to get a general understanding of what the system does, which supports an effective system operator training and is sufficiently exact to prevent as much as possible a faulty system use,[10] and (d) for the maintainers, allowing them to analyse faulty run-time behavior in the abstract model.

Despite the abstract character of ASM models, which characterizes them as specifications, they can and have been refined to machine executable versions in various natural ways (see Sect. 8.3). Due to their abstract character they *support generic programming, hardware–software co-design as well as portability* of code between platforms and programming languages – the major goal of the "model driven architecture" approach to software development. ASM models are in particular compatible with cross-language interoperable implementations as in .NET. Since they are tunable to the desired level of abstraction, they support information hiding for the management of software development and the formulation of well-defined interfaces for component–based system development. The general refinement notion supports a method of stepwise development with traceable links between different system views

teams may contribute to a better high-level understanding of the system under development, but if the program managers and the implementers do not understand the resulting formalizations, this does not solve the fundamental problem of keeping the models at the different abstraction levels in sync with the final code – the only way to make sure the code does what the customer expects from the agreed upon ground model. The ASM method is not a formal method in the narrow understanding of the term, but supports any form of rigorous design justification. This includes in particular mathematical proofs, which represent the most successful justification technique our civilization has developed for mental constructions – the type of device to which models and programs belong. See the footnote to the "language and communication problem" at the beginning of Sect. 2.1.1.

[9] In this way a design does not remain only in the head of its creator and can play a role in later phases of the software life cycle.

and levels. In particular this allows one to localize the appropriate design level where changing requirements can be taken into account and where one can check that the design change is not in conflict with other already realized system features.

### 1.1.4 Harness Pseudo-Code by Abstraction and Refinement

As is well-known, pseudo-code and the abstraction-refinement pair are by no means new in computer science, often they are even looked at with scepticism – by theoreticians who complain about the lack of an accurate semantics for pseudo-code, by practitioners who complain about the difficulty of understanding the mathematics behind formalisms like abstract data types, algebraic specifications, formal methods refinement schemes, etc. So what value does the ASM method add to these omnipresent ideas and how does it avoid the difficulty to apply them in practical software engineering tasks?

The ASM method makes the *computational meaning of abstraction and refinement* available explicitly, in a mathematically precise but simple, easily understandable and easily implementable pseudo-code-like setting, including the crucial notion of runs. The formulation uses only *standard mathematical and algorithmic terms*, circumventing the unnecessary formalistic logico-algebraic complications that practitioners so often rightly complain about, so that the method comes as a set of familiar intuitive concepts which naturally support the practitioners' daily development work. To read and write ASMs no knowledge of the underlying theory is needed, though it is the mathematical underpinning which makes the method work. This is analogous to the role of axiomatic set theory, which provides the precise setting in which mathematicians work without knowing about the logical foundation.

Looking back into the history of computing reveals that the ingredients of the concept of the Abstract State Machine were there for decades before they appeared combined in the definition discovered in [248], triggered by a purely foundational concern:[11] (a) pseudo-code, (b) IBM's concept of virtual machines [305] and Dijkstra's related concept of abstract machines [183] (both born as operating system abstractions), and (c) Tarski structures as the most general concept of abstract states [325, 265, 359, 210]. It is the mathematical character one can attach through the notion of ASM to the semantically open-ended loose notions of pseudo-code and virtual machines which turns these concepts into elements of a scientifically well-founded method; it is the natural expression of fundamental intuitions of computing through ASMs and the simplicity of their definition which make the ASM method comprehensible for the practitioner and feasible for large-scale industrial applications.

[10] An important example of an erroneous use of a system whose control is shared by humans and computers, e.g. in modern aircrafts, is known as *mode confusion*. See [322] and the notion of control state ASM in Sect. 2.2.6, which provides a way to make the overall mode structure of a system transparent.

[11] See Chap. 9 for details.

The ASM method does nothing else than putting the elementary definition of local updates of abstract states together with Wirth's original stepwise refinement approach [429] and with the concept of ground model [71, 72, 76] to link requirements capture to code generation in a coherent framework. Historically speaking the ASM method "complete(s) the longstanding structural programming endeavour (see [164]) by lifting it from particular machine or programming notation to truly abstract programming on arbitrary structures" [86, Sect. 3.1]. It also appears as a natural completion of the evolution during the last century from programming to generic programming and high-level platform-independent modeling: leading from programming-any-which-way in the 1950s to programming-in-the-small in the 1960s to programming-in-the-large in the 1970s to programming-in-the-world since the 1990s[12] where, due to the evergrowing hardware performance, security, robustness and reusability play a larger role than time or space efficiency.

### 1.1.5 Adding Abstraction and Rigor to UML Models

UML exercises a strong attraction by the multitude it offers for radically different interpretations of crucial semantical issues. On the tool side this is reflected by the numerous "UML+...-systems", e.g. UML+RUP, UML+XP, UML+IBM Global Services Method, etc. However, this conceptual multitude is not mediated (except for being simply declared to constitute so-called "semantical variation points"), in addition it is represented by a limited graphical notation and prevents UML from supporting precise practical refinement schemes (see Sect. 2.1.2, 3.2). Furthermore, the drive in UML to structure models right from the beginning at the class level imposes a rather low level of abstraction, typically close to procedural code, besides leading to the risk of a conceptual explosion of the class hierarchy. It also makes it difficult to model features which relate to multiple classes and which are often spread in the class hierarchy (e.g. safety, security, logging), or to describe crosscutting concerns relating to one feature at different class levels. Furthermore, it has no clear semantical model of "atomic" versus "durative" actions and of asynchronous computation of multiple threads.

The ASM method we explain in this book provides a means to handle such architectural features in an accurate yet transparent way and at a higher level of abstraction than UML, providing support for a truly human centric yet precise algorithmic design and analysis, which is *completely* freed from the shackles of programming language constructs and of specific typing disciplines. For example, it provides a simple accurate semantics for standard diagram techniques (see the notion of control state ASMs in Sect. 2.2.6 and the ASM definition of UML activity diagrams in Sect. 6.5.1) and for use cases and their refinements to rigorous behavioral models (see Chap. 3), it

[12] The wording is taken from Garlan's lectures on *Software Architecture* held at the Lipari Summer School on *Software Engineering*, July 2002.

supports component techniques (see Sect. 3.1.2 and [406]), it uniformly relates sequential and asynchronous computations capturing the data exchange of interacting objects (see Sect. 6), it provides a clear definition of “atomic” and “composed” computation step (see Chap. 4), etc. To realize further design steps, high-level ASM models can be *refined* by object-oriented mappings to classes, by introducing type disciplines where useful, by restricting runs when needed to satisfy specific scheduling principles, etc.

## 1.2 Synopsis of the Book

This book was conceived to serve the double purpose of (a) a modeling handbook and textbook, teaching how to practically apply the ASM method for industrial system design and analysis (including its management and its documentation), and of (b) an ASM compendium, providing the underlying theory and a detailed account of ASM research. The domains of application cover sequential systems (e.g. programming languages and their implementation), synchronous parallel systems (e.g. general and special-purpose architectures), asynchronous distributed systems and real-time systems (network and communication and database protocols, control systems, embedded systems). This also determines the structure of the book, which leads from the definition of *basic* single-agent ASMs in Chap. 2 with an illustration of the principles of hierarchical system design by ground model construction and stepwise refinements in Chap. 3 to *structured ASMs* in Chap. 4, *synchronous* multi-agent ASMs in Chap. 5, and *asynchronous* multi-agent ASMs in Chap. 6. This is followed by Chap. 7 on the universality of ASMs, Chap. 8 on ASM tool support (on computer-supported verification of ASMs and on ASM execution and validation techniques). It concludes with Chap. 9, which surveys the ASM research, together with its applications and industrial exploitations, from its beginning to today and comes with an, as we hope, complete annotated bibliography of ASM related papers from 1984–2002.

A detailed index (including also the major ASMs defined in this book) and lists of figures and tables aid navigation through the text. The use of the book for teaching is supported by numerous exercises, most of them coming with solutions on the accompanying CD, and by pdf and powerpoint format slides on the CD, covering most of the chapters or sections of the book. Additional material (including lecture slides) and corrections are solicited and will be made available on the ASM book web site at `http://www.di.unipi.it/AsmBook/`. This includes the set of LaTeX macros we have used to write the ASMs in this book. They come with a tutorial explaining to the reader how to write his own ASMs in a strikingly simple and elegant way using this set of macros, which by its very nature can be extended and tailored to specific needs.

**Central themes of the chapters.** In Chap. 2 we introduce the three constituents of the ASM approach to system design and analysis: the concept of *abstract state machines*, the *ground model method* for requirements capture, and the *refinement method* for turning ground models by incremental steps into executable code. The notion of *basic ASMs* is defined which captures the fundamental concept of “pseudo-code over abstract data”, supporting its intuitive understanding by a precise semantics defined in terms of abstract state and state transition. Finite State Machines (FSMs) are extended by *control state ASMs*.

In Chap. 3 we illustrate the *ground model method* for reliable requirements capture (formulating six fundamental categories of guideline questions) and the *refinement method* for crossing levels of abstraction to link the models through well-documented incremental development steps. The examples are control state ASMs for some simple devices (ATM, Password Change, Telephone Exchange), a command-line debugger control model, a database recovery algorithm, a shortest path algorithm and a proven-to-be-correct pipelined microprocessor model. Sect. 3.2.3 presents Schellhorn’s scheme for modularizing and implementing ASM refinement correctness proofs.

In Chap. 4 some standard refinements for structuring ASMs are defined and their applications illustrated. The building blocks of *turbo ASMs* are sequential composition, iteration, parameterized (possibly recursive) submachines; they permit us to integrate common syntactical forms of encapsulation and state hiding, like the notion of a *local state* and a mechanism for *returning values* and *error handling*. We characterize turbo ASM subcomputations as SEQ/PAR-tree computations. *Abstract State Processes* realize the above constructs in a white-box view where interleaving permits us within a context of parallel execution to also follow the single steps of a component computation. As an illustration we provide succinct turbo ASMs for standard programming constructs, including the celebrated Structured Programming Theorem and forms of recursion which are common in functional programming.

In Chap. 5 multi-agent synchronous ASMs are defined which support modularity for the design of large systems. They are illustrated by *sync ASMs* for solving a typical industrial plant control problem (Production Cell) and the Generalized Railroad Crossing problem (verified real-time gate controller).

In Chap. 6 asynchronous multi-agent ASMs (*async ASMs*) are defined and illustrated by modeling and analyzing characteristic distributed network algorithms (for consensus, master–slave agreement, leader election, phase synchronization, load balance, broadcast acknowledgement), a position-based routing protocol for mobile ad hoc networks, a requirements capture case study for a small embedded (Light Control) system, two time-constrained algorithms which support fault tolerance for a distributed service (Kermit and a Group Membership protocol), Lamport’s mutual exclusion algorithm *Bakery* with atomic or with durative actions, and the event-driven UML activity diagrams.

In the foundational Chap. 7 we investigate the universality properties of ASMs. We show that ASMs capture the principal models of computation and specification in the literature, including the principal UML concepts. We explain the ASM thesis, which extends Church's and Turing's thesis, and prove its sequential version from a small number of postulates.

Chapter 8 is dedicated to tool support for ASMs. In Sect. 8.1 we deal with techniques for mechanically verifying ASM properties, using theorem proving systems or model checkers. We present a logic tailored for ASMs and the transformation from ASMs to FSMs which is needed for model-checking ASMs. In Sect. 8.3 we survey various methods and tools which have been developed for executing ASMs for simulation and testing purposes. The history of these developments is presented in Sect. 9.4.3, which is part of Chap. 9, where we survey the rich ASM literature and the salient steps of the development of the *ASM method* from the epistemological origins of the *notion of ASM*.