# Didactical Issues at the Interface of Mathematics and Computer Science

**Viviane Durand-Guerrier, Antoine Meyer and Simon Modeste**

## 1 Introduction

The work supporting this chapter takes place in the context of the ongoing research project *DEMaIn* (Didactics and Epistemology of interactions between Mathematics and Informatics), funded by the French ANR (National Agency for Research). This project addresses the epistemology and the didactics of the relations between mathematics and computer science. Its aim is to gain a better understanding of the relations between these two disciplines by studying the foundations, objects, methods, types of questions and modes of thinking which they may share, or which may be specific to one of them. It also proposes to consider the questions that each field asks the other, and the uses that they may find for each other (as a tool or as an object of study).

The DEMaIn project has two main axes. The first deals with the scientific foundations of mathematics and computer science, in particular regarding logic, algorithms, language and proof. Indeed, thinking of the relationships between mathematics and computer science from an educational perspective leads to taking into consideration, among other questions, issues regarding proofs (seen as scientific texts) and proving (the activity of producing such texts) in both domains, and to identifying the role of logic as a possible lens through which to examine and hopefully better understand their interactions.

This chapter is structured as follows. In Section 2, we provide some additional context and motivation. In Section 3, we highlight a few key aspects of the logical

V. Durand-Guerrier (✉) · S. Modeste
IMAG, Univ Montpellier, CNRS, Montpellier, France
e-mail: viviane.durand-guerrier@umontpellier.fr

S. Modeste
e-mail: simon.modeste@umontpellier.fr

A. Meyer
LIGM (UMR 8049), UPEM, CNRS, ESIEE, ENPC, Université Paris-Est, Marne-la-Vallée, France
e-mail: antoine.meyer@u-pem.fr

G. Hanna et al. (eds.), *Proof Technology in Mathematics Research and Teaching*, Mathematics Education in the Digital Era 14,
https://doi.org/10.1007/978-3-030-28483-1_6

issues in mathematics and computer sciences. In Section 4, we analyze several ways in which algorithms and mathematical proof might interact in an educational context.

## 2 Motivation and Context

### *2.1 The Necessity of Epistemological Insights for Didactical Work*

According to Howson and Kahane (1986), the relationship between mathematics and computer science—especially the influence of computer science in mathematics and the role of mathematics in computer science—is an epistemological and didactical issue that transcends school systems and national contexts. The use of computer tools in the teaching of mathematics and informatics, raises questions about the nature of these tools. This can be connected to the particular role played by mathematics in computer science, the proximity of some aspects of both disciplines and the common nature of some of their questions. For example, in a didactical perspective, is it reasonable to use a chart plotter without questioning the accuracy of calculations or that of the display on the screen? Can we use dynamic geometry software without asking how exactly an intersection or a symmetry are built? Can we simulate random experiments without questioning how a machine can produce, or at least imitate, randomness? Can we implement a numerical or formal calculation without asking how a computer can interpret it or, on the contrary, why it rejects it? Can we design a long program without asking how we can make sure it does not contain errors?

The relationships between Mathematics and Computer science are deep and complex. According to Chabert (1999), they share objects, foundations and a part of their history. Indeed, computer science finds much of its theoretical and practical underpinnings in mathematics and has partly built itself as a branch of applied mathematics and logic before emancipating. In this respect, logic plays an important role in the interaction between mathematics and computer science. According to Sinaceur (1991b), logic (in line with Tarski's development) can be considered as an "effective epistemology" providing means for analysing mathematical practices and hence for understanding mathematical activity (op. cit. pp. 341–342). She also stressed that logic became, through computer science, an applied science, which echoes Aristotle's view of logic as an *Organon*. In Sect. 2, we will present the main logical issues in mathematics and computer science that we identify as relevant for our work.

Several authors consider that computer science raises new questions in mathematics, opens up new areas of research and enriches some traditional fields of mathematics (Colton, 2007; Kahane, 2002). Main aspects concern the modes of validation in mathematics through proofs such as those of the four-color theorem or Kepler's conjecture (e.g., Borwein, 2012), the value of the results by questioning the place of constructive proofs and algorithms (e.g., Basu, 2006), and their methods, in particular concerning the experimental dimension of mathematics (e.g., Perrin, 2007, 2012, 2012). New fields of mathematics such as discrete mathematics and theoretical computer science are developing at the interface between mathematics and computer science. This questions mathematicians and didacticians about how these fields should be passed on to teaching (see, Grenier and Payan, 1998; Hart, 1998; Lovasz, 2007; Ouvrier-Buffet, 2014).

Following Modeste (2016) who studied the introduction of algorithmic in high school in France, we formulate the hypothesis that an introduction of numerical tools or computer science elements in curricula without significant consideration of the epistemology of computer science, mathematics and their links, neither allows nor participates in an in-depth renewal of mathematics education, nor answers the problems of mathematics and computer science mentioned above. The increasing introduction of computer science elements in the teaching of mathematics in the curricula of various countries and in mathematics themselves, supports the importance and urgency of an epistemological and didactic study of interactions between mathematics and computer science.

## 2.2 *Institutional Context in France*

We present here some specifics of the teaching of computer science, algorithms and programming in French public schools. This section summarizes elements developed in Gueudet et al. (2017).

In the 1980s, in line with an international dynamic (Howson and Kahane, 1986), an optional teaching of computer science centered on algorithms and programming was introduced in upper secondary school in France. However there was at the time no social consensus in the country on the purpose and importance of this teaching (Baron and Bruillard, 2011), and computer science disappeared as a school discipline in the 1990s. It was replaced in curricula by a somewhat informal initiation to what is nowadays often referred to as *digital literacy*, namely the set of abilities allowing one to use computers and technology as *tools* for various purposes. These contents were referred to in France as *transversal* to underline the fact that they were not perceived as forming a standalone topic, but their teaching was rather spread amongts several disciplines (and assumed usually by non-specialised teachers).

In the 2000s, the CREM[1] (Kahane, 2002) advocated for the introduction of elements of computer science in mathematics school curricula and teachers' education, and defended the importance of interactions between mathematics and computer science, relying on the following arguments:

[1]*Commission de Réflexion sur l'Enseignement des Mathématiques*, National Commission for Reflection on the Teaching of Mathematics.

- Algorithmic thinking, implicit in the teaching of mathematics, could be developed and enlightened with the instruments of Algorithmic;
- Programming promotes formalized reasoning;
- Questions about effectiveness of algorithms involve mathematics;
- Data processing and digital computations are common in other disciplines;
- Computer Science transforms Mathematics, bringing new points of view on objects, bringing new questions, creating new fields in mathematics that are expanding rapidly, and changing the mathematician's activity with new tools.

Just after this report was published, algorithmic content was introduced in mathematics in grades 11 and 12, in literature series, and in optional mathematics courses in the last year of the economy and sciences series.

Later, between 2009 and 2012 in new official programs, algorithms were introduced as part of the mandatory mathematical content to be taught in all series of the general curriculum (literature, economy, sciences) from grades 10–12. Finally, in the 2010s, computer science reappeared as an autonomous discipline in upper secondary school, together with algorithms as part of the contents in mathematics. Since 2016, computer science is also taught in *cycle 4* (grades 7–9), but divided between two disciplines (mathematics and technology).

This renewal of the teaching of computer science in French curricula in mathematics raises the need for reworking and developing research in didactics of mathematics and informatics and of their interactions, which was the motivation for project DEMaIn. As a first step of the research, we led an epistemological study on these interactions in a didactic perspective, with a main focus on proof and proving. This is developed in Sect. 3.

## 3 Logical Issues in Mathematics and Computer Science

Following Durand-Guerrier and Arsac (2005), we consider that the classical first-order logic, namely the predicate calculus in the semantic perspective opened by Frege, Wittgenstein or Tarski, is a relevant epistemological reference for analysing proof and proving in mathematics education. Following authors such as Gribomont et al. (2000), we hypothesise that it is also the case for computer science. In this section,[2] we give a brief overview of this topic.

It should be noted that, even though more specialized logics and techniques exist in the research literature on programming language semantics and program verification, we do not focus here on the theories underlying automated or computer-assisted proof systems, even though they might be of interest as teaching tools. Since we are concerned here with the practice of proof in secondary or undergraduate education, we hypothesize that classical first-order logic is relevant for most of our goals.

[2]This was presented in an unpublished regular lecture given at ICME 11 (http://www.icme11.org/).

## *3.1 Semantic Perspectives in Logico-Mathematical Disciplines*

In this text, semantics is considered in a logical perspective consistent with the definitions given by Morris (1938): *semantics* concerns "the relation of signs to the objects which they may or do denote" (op. cit. p. 21); *syntax* concerns the "relations of signs to one another in abstraction from the relations of signs to objects and interpreters" (op. cit. p. 13), and *pragmatics* refers to "the relation of signs to their users" (op. cit. p. 29). Morris claims that "Syntactics, Semantics and Pragmatics are components of the single science of semiotic but mutually irreducible components" (op. cit. p. 54). We illustrate the relevance of this approach below.

For example, when considering the addition of natural numbers, the semantic point of view refers to the definition of the sum as the cardinal of the union of two relevant discrete collections; the result is independent of the nature of the involved objects (provided that mixing these objects preserves their integrity). The syntactic point of view arises when addition is defined as the iteration of the successor operation; it does not require any reference to quantities; this provides algorithmic rules in a given system of numeration. Finally the pragmatic aspect concerns the articulation between syntax and semantics that is built by subjects in a back-and-forth between calculation (syntax) and effective counting (semantics). According to Da Costa (1997, p. 42), it is necessary to take in account all three of these aspects in order to gain a proper understanding of logico-mathematical fields.

Regarding computer science, one may consider that syntax is at the very core of the discipline, but there is evidence that semantic and pragmatic aspects are also involved [see for instance Gribomont et al. (2000)].

The semantic perspective in logic appears in Aristotle, and was developed in the late nineteenth and early twentieth centuries, mainly by Frege (1882), Wittgenstein (1921), Tarski (1933, 1943) and Quine (1950). In particular, Tarski (1933, 1943) provides a semantic definition of truth which he describes as *formally correct and materially adequate*, through the crucial notion of satisfaction of an open sentence by an object, and developed a model-theoretic point of view, of which semantics is at the very core.

### 3.1.1 The Semantic Conception of Truth

The main concern of Tarski is to give a definition of truth materially adequate and formally correct (Tarski, 1943). He claims his only intent in this work is to grasp the intuitions formulated by the so-called "classical" theory of truth, i.e. the conception that "truly" has the same meaning as "in agreement with reality" (contrary to a conception that "true" means "useful in such or such regard" (Tarski, 1933).

In order to be formally correct, such a definition ought to be recursive, but recursivity is usually difficult to grasp directly. Tarski's idea was to introduce the notion of satisfaction of a propositional function (in modern terms, a predicate) of a given formal language in a "domain of reality" (a piece of discourse, a mathematical theory etc.). In the field of algebra, this definition coincides exactly with that of solution of an equation. Tarski argues that this definition of satisfaction is the key for a recursive definition of the truth of a complex sentence.

First, there is an extension of logical connectors between propositions, as defined by Wittgenstein, to connectors between propositional functions (predicates). For example, given an interpretation, and $P$ and $Q$ two monadic predicates (with exactly one free variable), and $a$ an element of the discourse universe, $a$ satisfies $P(x) \Rightarrow Q(x)$ if and only if $a$ satisfies $P(x)$ and $Q(x)$, or $a$ does not satisfy $P(x)$.

Second, the two quantifiers "for all" and "there exists at least one" are defined in agreement with common sense. Then, once the logical structure of a sentence is identified (atomic formulae, scope of connectors and quantifiers), it is possible to establish the truth of the whole sentence as soon as one knows the truth-value of the interpretation of each atomic formula.

### 3.1.2 A Model-Theoretic Point of View

The model-theoretic point of view emerged in Tarski (1954, 1955), but the main ideas were already present in previous papers. It relies on a simple and very fruitful idea. At first, Tarski (1936, 1983) considers the notion of model of a formula. Given a formalized language $L$, a syntax providing recursively well-formed statements (formulae): $F$, $G$, $H$..., an interpretative structure (a domain of reality, a piece of discourse, a mathematical theory, a computation model) is a model of a formula $F$ of $L$ if and only if the interpretation of $F$ in this structure is a true statement.

Some formulae are true for every interpretation of their letters in every non-empty domain. They are said to be universally valid (Quine, 1950). This is a generalisation of the notion of tautology in propositional calculus. A classical example is the logical equivalence $\forall x\ (P(x) \Rightarrow Q(x)) \Leftrightarrow \forall x\ (\neg Q(x) \Rightarrow \neg P(x))$ which describes the equivalence between a universal implication and its contrapositive and gives a logical basis to *proofs by contraposition*.

From the concept of model of a formula, Tarski defines the key concept of logical consequence in a semantic perspective: "The sentence $X$ follows logically from the sentences of the class $K$ if and only if every model of the class $K$ is a also a model of the sentence $X$" (Tarski, 1983, p. 417). As was the case for propositional logic in Wittgenstein (1921), logical consequences support classical modes of reasoning. For example $Q(y)$ is a logical consequence of $P(y) \wedge \forall x(P(x) \Rightarrow Q(x))$. It corresponds to the extension of the propositional inference rule named *modus ponens* to predicate calculus.

### 3.1.3 The Methodology of Deductive Sciences

In his famous book *Introduction to logic* (Tarski, 1941; 1995), Tarski introduced in chapter VIII the methodology of deductive sciences. To a given miniature deductive theory (he gave the example of the congruence of line segments), in which there are primitive terms, defined terms, axioms and theorems, one may associate an *axiom system*, with no reference to objects, which takes the form of a language and a set of formulae that can be reinterpreted in the given miniature theory. He then defines a *model* of the axiom system as any interpretation in which the formulae corresponding to the axioms of the given theory are interpreted as true. Of course the initial theory is a model of the obtained axiomatic formal system, but there may also be other models.

This leads to an important result and a powerful method for proving. Tarski proves, along with other logicians, the deduction theorem (in the meaning of Tarski), namely:

> Every theorem of a given deductive theory is satisfied by any model of the axiom system of this theory; and moreover, to every theorem there corresponds a general statement which can be formulated and proved within the framework of logic and which establishes the fact that the theorem in question is satisfied by any such model. We have here a general law from the domain of methodology of deductive sciences, which, when formulated in a slightly more precise way, is known as the law of deduction (or the deduction theorem). (Tarski, 1995, p. 127)

As a consequence, "All theorems proved on the basis of a given axiom system remain valid for any interpretation of the system" (op. cit. p. 128).

This observation leads to the idea of *proof by interpretation*: one way to prove that a given statement is not a logical consequence of the axioms of a certain theory is to provide a model of the theory that is not a model of the formula associated with the statement in question. This can be seen as analogous to the use of a counterexample to the possibility of a proof or to the validity of a proof of a true statement.

Following Sinaceur (1991a), we consider that the model-theoretic point of view offers powerful tools enabling us to take into account both form and content and to distinguish between truth and validity, both crucial issues of the teaching and learning of mathematics. In a didactic perspective, Durand-Guerrier (2008) has shown that this point of view offers fruitful paths to enrich *a priori* analyses and to analyse students' activity in mathematics. We will now attempt to provide evidence that this is also the case in computer science education.

### 3.1.4 Example: Tiling by Dominos

This example is based on our experience with *research situations for the classroom* (Gravier, 2008; Godot and Grenier, 2004). The problem is the following: given a rectangular grid (with integral dimensions), is it possible to tile it with dominoes ($1 \times 2$ rectangles)?

**Theorem 1** *A rectangular grid can be tiled by dominoes iff its area is even.*

A frequent (incorrect) proof of the above theorem given by students is the following. A grid can be tiled by dominos if and only if its area is $2k$ where $k$ is the number of dominos, which means that the area of the grid is even.

The stated theorem is correct but the proof is not. It is sometimes difficult to invalidate an incorrect proof of a true statement. In order to do so, one can notice that the fact that the grid is rectangular was not used in the proof. So, this proof can be used for any shape consisting of an even number of squares. It is easy to see that the shape can not be tiled by dominos but has an even area.

In other words, the set of grids of arbitrary shapes is a model of the theory used in the proof above. But in this model, the theorem becomes false. Hence, the initial proof is invalid (because otherwise it could be transported into the new model).

## *3.2 Logic and Proof in Computer Science*

In Hopcroft et al. (2007, p. 5), the authors give the following remark:

> In the USA of the 1990s [sic] it became popular to teach proof as a matter of personal feelings about the statement. While it is good to feel the truth of a statement you need to use, important techniques of proof are no longer mastered in high school. Yet proof is something that every computer scientist needs to understand.

In this section, we focus on the privileged role that logic and proof play in computer science, in particular to reason about programs and algorithms. We start by giving a few ideas on the interplay between syntax and semantics in the context of programming, then on the issues underlying the translation of an ideal algorithm into the rigid syntax of a programming language. We present a few classical types of proofs required in the study of algorithms, provide an example of such a proof using an ad-hoc deduction system, and close this section by a very brief presentation of the links between logic and well-known computation models called finite automata.

### 3.2.1 Syntax and Semantics of Programming

The distinction between syntax and semantics is somewhat more obvious in computer science (in particular in programming) than in mathematics, due to the way computers interpret programs. Indeed, for an algorithm to be executable by a machine, one first has to express it as a *text* amenable to automated treatment, from the lowest possible description level (elementary machine instructions) to the highest (modern programming languages). In this context, "syntax" refers to the rules of composition of a valid text in the chosen language, and "semantics" to the expected effect on the actual machine (or a model thereof) of each construct in that language and of their combinations.

Important pieces of software called *compilers* and *interpreters*, which rely on theoretical advances from the last decades of the 20th century, enable automatic translations of higher-level programs into machine-level lists of instructions (see for instance, Aho et al. 1986). These tools proceed in several phases, the first of which (lexical and syntactic analysis) aim at ensuring that the text of a program respects the formal syntax of the chosen language. Further steps are mostly of a semantic nature: checking for type errors, modifying parts of the program, translating it into another, possibly lower-level language while preserving its meaning, or even running (*interpreting*) the program directly.

Contrary to low-level program descriptions such as assembly languages, modern languages are designed to be executable on several (possibly any) computer architectures. This "abstraction" from material constraints is an essential aspect of modern programming, in that it allows one to work at a level closer to general algorithmic ideas rather than being distracted by technical issues. Enforcing the semantic consistency of programs through each of these transformations regardless of the final target architecture is thus an essential responsibility of programming language designers and compiler implementers.

The study of programming language semantics is a wide and very active field of research, with numerous links to deep mathematical theories. It is essential for the design and understanding of whole paradigms of programming.

### 3.2.2 From Algorithms to Programs

Describing an algorithm as a machine-executable program is somehow similar to translating an informal mathematical statement in some formal (for instance logical or axiomatic) language, which can then be interpreted in the appropriate mathematical model. Indeed, algorithms are often described informally, using either natural language, mathematical notations, *pseudo-programs* expressed in some semi-formal language inspired by actual programming languages, or a mix of all three.

When one wishes to actually produce an executable program realizing the tasks described by such an informal algorithm (its *implementation*), one therefore has to remove any possible ambiguity, and ensure that this translation process faithfully renders the ideas and principles which allow the algorithm to solve the problem at hand. In some sense, like a mathematician's, a programmer's activity therefore has to do with *pragmatics*: it proceeds as a constant back-and-forth between syntax and semantics, with the additional parameter of technical constraints. This pragmatic work is very similar to the work of producing the formal proof of a mathematical result using a proof assistant. In some sense, formalizing an algorithm into a program is an activity of the same nature as producing a formal proof from a standard mathematical proof.

However, as is the case in mathematics, ensuring that some formal statement is syntactically correct is not enough to guarantee that it is "true", or in this case that it actually performs the task it was meant to perform. Therefore, in order to ascertain the actual *correctness* of a program (or even of the algorithm it is supposed to implement), one usually has to resort to external arguments which are of a logical or mathematical nature.

### 3.2.3 Reasoning About Programs or Algorithms

One of the most obvious questions one may ask about an algorithm or a piece of program is "Does it work?". Trying to state this question more precisely leads to a formal definition of *computation problems*, which one may summarize as: "mathematical relations between a set of *instances* and a set of *results* (or *answers*, or *solutions*)". One further distinguishes *decision* problems, where possible answers are simply truth values. In this case, a problem might equivalently be described as its *set* of positive instances, instances which are mapped to the value *true*. Such problems play an important role in the more theoretical aspects of computer science, in particular in formal languages, automata and computation theories [see for instance Hopcroft (2007)].

**Correctness** Let $P$ denote an algorithmic problem. Seeing $P$ as a map between instances and outcomes, let us write $P(x)$ the outcome associated with some admissible instance $x$. To say that an algorithm or program $A$ solves problem $P$ means that given any admissible instance $x$ of $P$, $A$ is able to provide (indeed *compute*), after a finite sequence of *elementary operations*, a description of $P(x)$. In view of this, the question of knowing whether some algorithm $A$ "works" (i.e. "proving" $A$) comes down to establishing the following two properties, whose conjunction might be seen as expressing the (full) correctness of $A$:

*Termination*: on any instance of $P$, $A$ performs at most a finite number of computation steps.

*Partial correctness*: on any instance $x$ of $P$, the value computed by $A$ is $P(x)$.

**Complexity** The above questions are sometimes complemented by questions regarding $A$'s efficiency, in terms of the number of computation steps it performs on instances of a certain size (assuming some appropriate notion of *size* on instances). One typical property of interest is:

*Worst-case upper bound*: Function $f$ is a worst-case upper bound for the complexity of $A$ if there exists a positive constant $c$ such that, on any instance of size $n$ of $P$, the number of computation steps performed by $A$ is at most $c \cdot f(n)$ for $n$ large enough.

Similar questions can be asked of the amount of memory required by an algorithm (*space* complexity). Such concerns form the well-established fields of *complexity theory* and *algorithm analysis*, which strongly rely on tools and techniques from algebra and combinatorics. In the above statement, one is concerned with worst-case guarantees on the number of performed computation steps. Other natural questions concern the behaviour of $A$ on *typical* cases, bringing into play the question of probabilistic distributions on instances, and possibly involving powerful techniques from probability theory and analysis (see for instance Arora and Barak, Arora and Barak (2009) for an introduction to the field, or Sedgewick and Flajolet (2013) for more in-depth material on average-case analysis).

**Lower bounds and optimal algorithms** Finally, interesting questions lie *beyond* the analysis of a single algorithm solving a problem $P$, and study the intrinsic complexity of $P$ itself. For instance, in the so-called *comparison tree model*, in which all executions of an algorithm are decided through a series of elementary, binary comparisons between numbers, it can be shown that the well-known problem of sorting a list of numbers *cannot* be solved using less than $n \log n$ comparisons, up to a constant factor [for more details, see (Cormen et al. 2009)].

This impossibility result comes at the price of a rather involved argument, with several "layers" of quantification: one has to consider the longest computation, on any instance of some size $n$, of the (hypothetical) most efficient algorithm solving $P$. This supports the claim that proficiency with logic and reasoning are a prerequisite for a reasonably complete understanding of algorithmic concepts.

Modeste (2012, 2013) showed that this theoretical view of algorithms, which leads to adopting a definition of *problem* as a set of instances and a question about any of the instances, can be used as a relevant didactic tool, in particular to help develop an epistemological model for didactical purposes, to analyse curricula and to design didactical situations (see Sect. 4). Meyer and Modeste (2018) give a example of the didactical analysis of an algorithmic question (about the binary search algorithm and the bisection method).

### 3.2.4 An Example: Partial Correctness Using a Deduction System

In Gribomont et al. (2000), several key examples of particularly fruitful uses of logic in a computer science setting are given. The first such example is that of Hoare logic (Hoare, 1969), which may be used to show the partial correctness of a sequential program.

The main bulding block of Hoare logic takes the form of triples $\{P\}\,C\,\{Q\}$, called *Hoare triples*, where $P$ and $Q$ are assertions (usually written in classical predicate logic) and $C$ is a program statement. Such a triple expresses the fact that, whenever $P$ holds in some state of the machine (or model thereof) over which statement $C$ is executed, it must be the case that $Q$ holds in the state which is reached after $C$ is performed. Hoare triples are manipulated using deduction rules, which are very reminiscent of classical proof systems. One of the simplest rules describes the semantics of sequential composition:

$$\frac{\{P\}\,C\,\{Q\} \quad \{Q\}\,D\,\{R\}}{\{P\}\,C;\,D\,\{R\}}$$

This rule expresses the fact that if $\{P\}C\{Q\}$ and $\{Q\}D\{R\}$ are both valid Hoare triples, then by performing statements $C$ and $D$ from a state verifying assertion $P$, one may guarantee that assertion $R$ holds. Combining several rules of this kind and additional mathematical knowledge about manipulated values (for instance arithmetic) allows one to formally prove that, if and when a program terminates, some assertion holds at the end of its execution. See Gribomont et al. (2000) for a more detailed description and example, or Reynolds (1998) for a textbook covering this topic among others.

The issue of termination is of a different nature and cannot be established using this technique. It has to be proven separately, often relying on some kind of infinite-descent argument. Other examples of how logico-mathematical formalisms may be used in order to reason about other kinds of programs are given in Gribomont et al. (2000). One may cite in particular the cases of functional programs (where recursion and more particularly structural induction play a central role), concurrent or parallel programs, etc.

**A word on structural induction** To conclude this section, let us remark that the correctness of the final assertions obtained by applying the above technique actually relies on a *structural induction* argument: indeed, the syntactic structure of a program's text can be described by its so-called *abstract syntax tree*, whose nodes are program constructs and whose leaves are essentially identifiers and values. Successively applying deduction rules such as the one described above for sequential composition actually comes down to inductively labelling each node of this tree, from the leaves to the root, with sets of assertions. Finally, assertions carried by the root of the tree represent true facts about the whole program.

Other (simpler) examples of the usefulness of structural induction are provided in Gribomont et al. (2000), in the context of correctness proofs for functional programs written in the language LISP or SCHEME.

### 3.2.5 Modelling Program Behaviour Through Logic and Automata

Another bridge between logic and computer science illustrated by Gribomont et al. (2000) concerns the study of a class of computation models called *automata*, which stem from a long line of research originating in the 1960s and have known many interesting developments. These results are collectively referred to as *automata theory* (see for instance Hopcroft et al. (2007) for a classical textbook, Straubing and Weil (2012) or Thomas (1997) for a more logic-oriented exposition).

**Finite-state automata** A finite-state automaton is characterized by a finite directed edge-labelled graph, whose vertices and edges are respectively called *states* and *transitions*. Some states are marked as *initial*, others as *terminal* or *accepting*. The labels of edges are called *letters*, they belong to a finite set called *alphabet*. A sequence of letters, or *word*, is said to be *accepted* by an automaton if its letters label the successive transitions along a path from some initial state to some final state (or in the case of infinite words, to some kind of accepting "repetition"). Each automaton therefore accepts a *language*, which is the set of all words it accepts.

Finite automata have very good and well-understood algorithmic properties. For instance, one can write algorithms to decide whether a given automaton accepts the empty language or the set of all possible words, build an automaton whose language is the union or intersection of the languages of two other automata or the complement of the language of a given automaton. A particularly strong connection between automata and logic, discovered in the 1960s and much developed since, is that the languages of some classes of automata coincide with the languages defined by some classes of logics (see, Thomas, 1997). Furthermore, in several cases there exist algorithms able to transform a logical formula into an equivalent automaton and vice-versa.

**Modelling and verifying programs** A typical application of automata theory has to do with automated program verification or *model-checking*. In this framework, a program is modeled as a (potentially very large) finite-state automaton, say $A$, in such a way that each execution of the real program corresponds to a (possibly infinite) path in $A$, but $A$ may exhibit additional behaviours. A property $\varphi$ to be verified on the program might then be expressed in a well chosen logical framework. This formula is then negated, and translated into another finite automaton $A_{\neg\varphi}$.

Determining whether the abstract program satisfies the property stated by formula $\varphi$ then amounts to checking whether the languages of automata $A$ and $A_{\neg\varphi}$ are disjoint, which can be done algorithmically. By construction of $A$, an erroneous answer may occur only in the case where some behaviour of $A$ which violates $\varphi$ is detected, but this behaviour does not exist in the original program (this is called a *false positive*). Otherwise, if no such execution is found, it is guaranteed that all executions of the actual program respect the property $\varphi$.

This verification procedure has shown great success despite the fact that it deals with finite-state computation models. Extending it to more realistic models while conserving good algorithmic properties is one of the challenges undertaken by the research field of program verification [see for instance Bérard (2001)].

## *3.3 First Conclusion*

Adopting a semantic point of view, and being situated at a meta-mathematical level, a model-theoretic point of view provides on the one hand a frame to analyse *a priori* the situations under both mathematical and didactical aspects, and on the other hand to analyse students' activity, in particular by providing the researcher with a methodology to identify and study the elaboration, the evolution and the eventual overtaking of the local axiomatic all along the resolution and/or the proving process.

It seems clear that Tarski's meta-mathematical project goes beyond mathematics and echoes key questions in computer science education, such as the relationship between deductive systems and models, including the issue of limits of validity of these models, the relationships between proofs and programs, the notion of proof of an algorithm.

It should also be noted that the pervasive use of logic and other mathematical tools in computer science has provided and will most likely continue to provide new ideas, questions and perspectives to the fields of logic and mathematics themselves.

## 4 Modes of Interaction Between Mathematics and Computer Science

The content of this section mainly originates from Modeste (2012). In a didactical perpective and on the basis of an epistemological analysis, it proposes to distinguish three main modes of interaction between mathematical proof and algorithms, and two kinds of problems in which algorithms appear. This allows one to better understand and analyse the interactions between both fields, and give meaning to several possible relationships with the concept of *algorihm*, that is, *conceptions* of algorithm.

This work draws upon the model of conceptions as developed by Vergnaud and enriched by Balacheff. The $cK¢$ model (conception, knowing, concept) was developed by Balacheff (2013) to build a bridge between mathematics education and research in educational technology. It proposes a model of learners' conceptions inspired by the theory of didactical situations (Brousseau, 1997) and the theory of conceptual fields (Vergnaud, 2009).

In this model, conceptions are defined as quadruples $(P, R, L, \Sigma)$ in which $P$ is a set of problems, $R$ a set of operators, $L$ a representation system and $\Sigma$ a control structure. $P$ and $L$ directly correspond with situations and representations in Vergnaud's model, $R$ and $\Sigma$ distinguish between operational invariants, those which allow one to act on problems and those which allow one to control the actions. For this reason, the $cK¢$ model is very relevant for emphasizing the dimension of proof.

### 4.1 Proof Paradigms

We first distinguish three frameworks in which our conceptions will be described, which we call *paradigms*. They are the *algorithmic proof* (AP), the *mathematical algorithm* (MA) and the *computer algorithm* (CA). These paradigms essentially correspond to three possible *habitats*[3] of the notion of algorithm in mathematical and algorithmic activity.

[3]The term *habitat* was coined by Artaud (1998) in the context of the so-called *ecological* approach to didactics.

- The **AP paradigm** corresponds to an activity of the form Problem–Theorem–Proof, where the proof is of a finite, constructive nature and can thus be seen as algorithmic in nature. Induction proofs in particular fall into this category.
  In this paradigm, algorithms and proofs are not dissociable: algorithms are not written directly but are implicit in the theorem's proof. They may be made explicit outside or after the proof, as a corollary or consequence thereof, in a fashion similar to the second paradigm (MA).
- The **MA paradigm** corresponds to an activity of the form Problem–Algorithm–Proof. For a given problem, one describes an algorithm solving all admissible instances, then provides a proof that this algorithm is indeed (totally) correct, in other words a termination proof (the algorithm provides an answer in a finite amount of steps on any instance) together with a partial correctness proof (whenever an answer is provided on an instance, it is the correct one). In this paradigm, algorithm and proof are clearly dissociated.
- The **CA paradigm** corresponds to an activity of the form Problem–Program–Validation. The algorithm solving the problem is expressed as a program which is expected to be executed on a machine.
  The term "validation" should be understood here in its ordinary meaning. It may or may not be of a mathematical nature, and may include all relevant tools and practices such as syntactic analysis and type-checking performed by the compiler, manual, semi-automatic, or automatic testing procedures, verification techniques (for instance model-cheking tools as described in Sect. 3.2.5), etc. Note that mathematical validation of the program or the underlying algorithm via proof (as in paradigm MA) may also provide a form of validation in CA.

A single problem can be addressed in different paradigms. Its study may even draw on several of them simultaneously or successively. What distinguishes the three paradigms is therefore not the kind of problems which are addressed but rather the way in which they are treated, the kind of solution obtained, and the kind of validation which is provided. They also differ in the concrete form in which algorithms are expressed. This encourages us in our choice to make use of the model of *conceptions* in order to formalize these differences in terms of operators, representation systems and control structures.

Representation systems for algorithms are an important criterion for distinguishing the three paradigms, but are not the only one. We must however acknowledge their impact on the way algorithms are expressed.

It remains to ask which kind of problems are the most appropriate for giving meaning to the concept of algorithm in the classroom.

## *4.2 The Tool–Object Dialectic*

We propose to adapt here a definition which comes from the theory of algorithmic complexity (see Sect. 3.2.3). According to this definition, a problem (e.g. finding the gcd) is given by a pair $(I, Q)$, with:

- $I$ a set of instances (e.g. $\mathbb{N}^2$, the set of all pairs of natural numbers) ;
- $Q$ a question about these instances (e.g. what is the gcd of the 2 provided numbers?).

This definition of problems allows to formalize what an algorithm is. An algorithm is a systematic method which must give an answer for all instances of the problem, after a finite number of steps (e.g. Euclid's algorithm solves the problem of gcd for any pair of natural numbers).

Additionally, we say a problem is *instantiated* when one chooses a particular instance $i$ and tries to answer the question $Q(i)$ for this particular case (e.g. what is the gcd of 3654 and 76?). To grasp the concept of algorithm in its full generality, it is important not to address only instantiated questions but to study a problem in all of its instances.

It is also important to distinguish two kinds of problems giving sense to the concept of algorithm:

- The set $\mathscr{P}_a$ of problems that may be solved using an algorithm (e.g. the problem of finding the gcd);
- The set $\mathscr{P}_A$ of problems that concern algorithms (which includes the problem of determining if a given problem is in $P_a$, the problem of determining the complexity of an algorithm, etc.).

In the first case, the algorithm is seen as a tool, and in a teaching context it is important that at least some of the chosen exercises and problems are general, and not instantiated (in some high school textbooks, several exercises in the same chapter turn out to be instantiated versions of the same problem). In the second case, the algorithm is seen as an object, and a problem can be instantiated (on a specific algorithm for instance).

We argue that the tool–object dialectic (Douady, 1986) can be useful to think about the interaction between mathematics and computer science, in particular to deal with proof issues.

Computer Science can be seen as a tool for mathematics (simulating experiences, or testing small cases) or an object (probabilities for analysing the complexity of an algorithm). Conversely, mathematics can be seen as an object or a tool for computer science, according to whether one is studying the mathematical or computer science aspects of a situation (as presented above).

## *4.3 Six Conceptions to Analyse Algorithmic Activity*

We now revisit the three paradigms defined above in the light of the tool–object dialectic. We saw how the distinction between problems in $\mathscr{P}_a$, where the algorithm is a tool, and $\mathscr{P}_A$, where it is the object of study, on one hand, and between instantiated and non-instantiated problems on the other hand, provide insights on the tool–object dialectic.

**Table 1** Conceptions in paradigm AP

| *(a) The AP-tool conception* | |
|---|---|
| $P$: | Problems in $\mathscr{P}_a$ |
| $R$: | Operators are those of proof restricted to “constructive” modes of reasoning (recurrence, induction, infinite descent, existence of lower or upper bounds of finite sets...), excluding in particular proofs by contradiction or using the law of excluded middle |
| $L$: | The representation system is that of *mathematical language*. The involved objects are mathematical object. In this conception, one manipulates information. At all times, all information provided by the instance, and all deduced information is usable. The only variables which are used are *mathematical variables* |
| $\Sigma$: | The control structure is that of *mathematical logic*, together with known properties of occurring objects |

| *(b) The AP-object conception* | |
|---|---|
| $P$: | Problems in $\mathscr{P}_A$ |
| $R$: | Operators are those of mathematical proof and operations on theoretical computation models (algorithmic reductions, simulations, etc.) |
| $L$: | The representation system is that of mathematical language together with theoretical computation models (automata, Turing machines, recursive functions, decision trees...) |
| $\Sigma$: | The control structure is that of *mathematical logic*, together with properties of occurring objects |

We therefore further refine the three paradigms presented above, by distinguishing in each case a tool-conception and an object-conception. This yields a total of six conceptions which we will not describe in detail.

Tables 1, 2 and 3 describe the different components ($P$: problems, $R$: operators, $L$: representation structures and $\Sigma$: control structures) of each of these conceptions. Each emphasized term in the description of a conception is explained below.

- **The AP-tool conception** (Table 1a) concerns inherent, implicit algorithms in the description of certain constructive mathematical proofs, whose object is *not* an algorithm or algorithmic fact itself. One may relate this to the notion of *constructive* mathematical proof.
  In the table, by *mathematical language* we mean the language commonly used in mathematical writings. By *mathematical variables* we refer to the different kinds of variables used in mathematics. By *mathematical logic*, we mean the set of implicit reasoning rules used in mathematical activity (in contrast with formal logic).
  **Example**: A proof of the characterization of Eulerian graphs (graphs which possess a cycle traversing each edge exactly once) as graphs whose vertices all have even degree, written in usual mathematical terms, where each step is constructive and the structure of reasoning attests to this constructiveness, might fall into this conception.

**Table 2** Conceptions in paradigm MA

| *The MA-tool conception* | |
|---|---|
| $P$: | Problems in $\mathscr{P}_a$ |
| $R$: | Operators are explicit algorithmic constructs (conditions, iterations, recursion) and effective, constructive operations on numbers, sets or other combinatorial or mathematical objects |
| $L$: | The representation system might be some type of pseudo programming language, mixing mathematical language and vocabulary inspired by programming practices. Manipulated objects are mathematical objects, sometimes belonging to the culture of computer science (for instance *abstract data types*), each admitting a known set of effective operations. Variables can be mathematical variables or *computer variables* |
| $\Sigma$: | The control structure is that of algorithm proof (correctness and termination) using all appropriate concepts and formalisms (logical proof systems, inductive proofs, infinite descent, invariants...). |
| *The MA-object conception* | |
| $P$: | Problems in $\mathscr{P}_A$ |
| $R$: | Operators are those of mathematical proof: algorithm proof, properties of algorithms, invariants, computational complexity... |
| $L$: | The representation system is that of mathematical language |
| $\Sigma$: | The control structure is that of mathematical logic, reasoning rules and properties of occurring objects |

**Table 3** Conceptions in paradigm CA

| *The CA-tool conception* | |
|---|---|
| $P$: | Problems in $\mathscr{P}_a$ |
| $R$: | Operators are those provided by a given programming language, possibly including instructions, conditional structures, loops, functions, and various predefined operations on data |
| $L$: | The representation system is a programming language. Manipulated objects are data values which encode (or model) objects from the original problem using data structures which allow certain operations |
| $\Sigma$: | The control structures are provided by various computer programming tools and practices, either manual or automatic, formal or informal |
| *The CA-object conception* | |
| $P$: | Problems in $\mathscr{P}_A$ |
| $R$: | Operators are those of formal program verification |
| $L$: | The representation system is that of mathematical language, together with the vocabulary and notations of appropriate analysis techniques and tools, possibly including *ad-hoc* proof systems or formal logic frameworks |
| $\Sigma$: | The control structure is provided by various relevant theories, including programming language theory, language semantics, computation models, and formal logic |

- **The AP-object conception** (Table 1b) deals with mathematical proofs about algorithmic objects or facts. Here, an algorithm or algorithmic problem may be provided as part of the question.
  **Examples**: A proof on the intrinsic complexity of some algorithmic problem might fall in this category: for instance proofs of the fact that *any* comparison-based sorting algorithm must perform at least $O(n \log n)$ comparisons in the worst case, or that the knapsack problem is NP-complete.
- **The MA-tool conception** (Table 2a) has to do with proofs explicitly providing an algorithm solving the mathematical problem (whose object is not itself explicitly algorithmic), and possibly providing some justification that the proposed algorithm is correct.
  In the table, the notion of *abstract data type* refers to the description of a specific data domain (for instance finite lists and maps, stacks or queues, graphs) through a set of allowed operations, without any *a priori* knowledge on implementation details. These operations are assumed to be constructive, or effective (in the sense that they can be performed algorithmically). Describing such algorithmic data types is an important part of the field of algorithm design.
  We call *computer variables*, for lack of a better term, entities that play the role of a temporary assignment between a name and a value, which is subject to change over time (for instance across multiple iterations of a loop). When a new assignment to a certain name is performed, the value previously assigned to it, if any, is lost. This is a simplified model of the memory of an actual computer during the execution of a program, designed to hide irrelevant technological details.
  **Examples**: A proof of the existence of the greatest common divisor of two integers, presented first by writing down Euclid's algorithm, then by proving its correctness, falls in this category. Other examples might include the description of list-sorting or list-searching algorithms.
- **The MA-object conception** (Table 2b) is similar to the AP-object conception in that it concerns problems about algorithms and their proofs. One possible difference in this case is that actual algorithms are manipulated explicitly, instead of potential or hypothetic algorithms.
  In this conception the description of the algorithm itself is considered part of the problem at hand; algorithmic description operators and representation systems are therefore not included in the table.
  **Example**: The analysis of the computational complexity (in terms of time or space) of a particular algorithm might fall into this category. The study of other properties may appear as well, for instance the stability of a given sorting algorithm.
- **The CA-tool conception** (Table 3a) refers to the activity of directly providing a computer program expected to solve a given problem, either numerically or symbolically, perhaps even in an approximate manner.
  Particular care might be taken in the validation and control of the proposed solution, which is considered good programming practice. The least required effort usually consists only in clear code documentation and sufficient testing, but other compelling arguments may be provided by other tools and procedures, such as syntactic analysis, type checking, certifiable code annotation, program verification, automated testing, etc.

**Example**: Computing the gcd of two integers using an implementation of Euclid's algorithm written in C or some other programming language falls into this conception.

- **The CA-object conception** (Table 3b) concerns questions asked about the properties of a given, explicit computer program.
  **Examples**: Writing manual or computer-assisted proofs about the termination or safety of programs relates to this conception.
  Another interesting and rather extreme example is the development of automated code analysis, interpretation or transformation tools themselves, for instance compilers, interpreters, profilers or debuggers for a given programming language, or automatic or semi-automatic verification software. In this case, the adopted representation system must be able to handle code reification: namely, the representation and manipulation of programs themselves.

## *4.4 Relationships Between Conceptions*

### 4.4.1 The Tool–Object Continuum

Let us first review the relationship between the tool-conception and object-conception of the same paradigm. We wish to make it clear that the boundary between these conceptions is not as clear-cut as our taxonomy might seem to indicate. Indeed, there is no wide gap between the two conceptions associated with a given paradigm, but rather a continuity according to the tool–object dialectic, or rather a transition from tool to object. This transition is accompanied by a move from specific, instantiated problems towards generic ones. However, there may in practice exist several intermediate problems which occur as one progressively extends and widens the set of instances of the problem at hand. One should also note that the control structure of the tool-conception plays a particular role in this transition from tool to object: the more it is present in a given activity, the closer one gets to the corresponding object-conception. The question of determining whether a given problem admits an algorithmic solution also plays a central role in this articulation.

Moreover, it should be remarked that when we mention the existence of this shift from tool to object, we do not mean to imply that algorithmic activity is necessarily linear or one-way. In fact, there are clearly numerous alternations, just like in mathematics, between tool and object. But it appears to us that this shift is globally directed from tool to object, in the sense that the use of any tool may naturally bring questions that make it a potential object of study. One can also see this as a movement which, in terms of the conceptions model, tends to change the status of control structures into operators in the context of new problems.

### 4.4.2 A Continuum Between Mathematics and Computer Science

Similarly, we have to point out that the distinction between paradigms AP, MA and CA is not absolute, but rather witnesses a fine gradation of conceptions and concerns between mathematics and computer science. This arbitrary split into three paradigms seems to offer a reasonable granularity for use as an analysis tool, to allow a sufficiently accurate representation of the various conceptions found in scientific literature and to support reflections about the production of this knowledge. Therefore, in addition to the tool–object continuity, there exists another continuity along the AP–CA axis, and it might be the case that certain activities found either in literature or in teaching may fall between two of our conceptions.

Still, let us remark that one may find some kind of "chronological" hierarchy (which one should absolutely not see in terms of value) between AP, MA and CA. For a given problem in $\mathscr{P}_a$, scientific study rather follows a chronological shift from AP to CA, via MA. One may decribe this as a transition from constructivity concerns (AP) to effectivity concerns (MA) and finally to implementation concerns (CA), accompanied by an increasing level of detail in the specification of algorithms.

This process obviously has exceptions, for instance in the case where no exact proof of a certain phenomenon exists but empirical observation through programming may still offer insights as to its validity. One should also be aware that many (indeed most) questions about algorithms and programs are of course *undecidable* in general. It is therefore useless to expect a single computer program to check *exactly*, given the text of any other program, whether its executions always terminate. Nevertheless, in some cases automated tools may be able to provide exact information about elementary properties of programs, or approximate answers to more difficult questions.

The continuity along the AP–CA axis is also accompanied by an increasingly clear separation of the aspects related to proof, syntax and semantics. Indeed, in the AP paradigm, all three concerns are intermixed. In MA, the solution is given as a "construction" or algorithm. It carries in some sense both syntax and semantics and is separated from its validation. Finally in CA, a strong accent is put on syntax (indeed, a computer executing a program only performs purely symbolic manipulations), while concerns of semantics and proof are left to the designer of the program (possibly with the help of computer-assisted tools). This interplay between validation, meaning and representation of algorithms, which varies between paradigms, is, for us, a central point to understand the role and place of proof in mathematics and computer science. We make the hypothesis that it has a strong potential for studying didactical issues about proof in mathematics, computer science and their interactions.

## 5 Perspectives

In this chapter, we have tried to provide evidence of the necessity and the relevance of studying the interactions between mathematics and computer science, with a particular focus on proof and taking into account the interplay between syntax, semantics and pragmatics. This opens up new avenues of research by helping identify issues in logic, mathematics and computer science that may be overlooked or remain implicit in the classroom. An example we are currently exploring is the representation and manipulation of polynomials. This topic illustrates several of the issues we discussed in this chapter:

- representation issues—polynomials represented as lists of coefficients or as arbitrary expressions via their syntax trees;
- algorithmic issues—computations in both representations (naive evaluation or using Horner's scheme, arithmetic operations on polynomials, formal derivation or integration, canonical forms, equivalence...);
- complexity analysis—comparison between representations;
- interplay between syntax and semantics—work on the structure of algebraic expressions;
- possible context for the introduction of inductive constructions and reasoning;
- classical classroom mathematical content—operator priorities and other algebraic rules, decomposition of algebraic formulae into calculation programs.

Due to the role of polynomials in calculus and analysis, we consider that developing didactical situations aiming to deal with these aspects will improve the knowledge of polynomials as objects, and as a consequence will foster students' skills in recognizing and using them as tools in relevant contexts.

**Acknowledgements** Research funded by the french *Agence Nationale pour la Recherche*, project number <ANR-16-CE38-0006-01>.

## References

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers, principles, techniques*. Addison Wesley.
Arora, S., & Barak, B. (2009). *Computational complexity: A modern approach*. Cambridge University Press.
Artaud, M. (1998). Introduction à l'approche écologique du didactique - L'écologie des organisations mathématiques et didactiques. *Actes de la IXème école d'été de didactique des mathématiques* (pp. 101–139). Caen: ARDM & IUFM.
Arzarello, F., Bussi, M. G. B., Leung, A. Y. L., Mariotti, M. A., & Stevenson, I. (2012). Experimental approaches to theoretical thinking: Artefacts and proofs. In *Proof and proving in mathematics education* (pp. 97–143). Springer.
Balacheff, N. (2013). *cK¢*, a model to reason on learners' conceptions. In *PME-NA 2013-psychology of mathematics education* (pp. 2–15). North American Chapter.
Baron, G. L., & Bruillard, É. (2011). L'informatique et son enseignement dans l'enseignement scolaire général français: enjeux de pouvoir et de savoirs. In: Recherches et expertises pour l'enseignement scientifique (Vol. 1, pp. 79–90). De Boeck Supérieur.
Basu, S., Pollack, R., & Roy, M. F. (2006). Algorithms in real algebraic geometry (2nd ed.). In *Algorithms and computation in mathematics* (Vol. 10 ). Berlin, New York: Springer.
Bérard, B. (2001). *Systems and software verification: Model checking techniques and tools*. Springer.
Borwein, J. M. (2012). Exploratory experimentation: Digitally-assisted discovery and proof. In *Proof and proving in mathematics education* (pp. 69–96). Springer.

Brousseau, G. (1997). *Theory of didactical situations in mathematics*. Kluwer Academic Publishers.
Chabert, J. L. (1999). *A history of algorithms from the pebble to the microchip*. Springer.
Colton, S. (2007). Computational discovery in pure mathematics. In *Computational discovery of scientific knowledge* (pp. 175–201). Springer.
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). The MIT Press.
Da Costa, N. C. A. (1997). *Logiques classiques et non classiques: essai sur les fondements de la logique*. Paris: Masson.
Douady, R. (1986). Jeux de cadres et dialectique outil-objet. *Recherches en didactique des mathématiques*, *7*(2), 5–31.
Durand-Guerrier, V. (2008). Truth versus validity in mathematical proof. *ZDM*, *40*(3), 373–384.
Durand-Guerrier, V., & Arsac, G. (2005). An epistemological and didactic study of a specific calculus reasoning rule. *Educational Studies in Mathematics*, *60*(2), 149–172.
Frege, G. (1882). Über die wissenschaftliche Berechtigung einer Begriffsschrift. *Zeitschrift für Philosophie und philosophische Kritik, 81*, 48–56. (English translation in *Conceptual notation and related articles*. Clarendon Press (1972)).
Godot, K., & Grenier, D. (2004). Research situations for teaching: A modelization proposal and examples. In *Proceedings of ICME 10*, IMFUFA, Roskilde University.
Gravier, S., Payan, C., & Colliard, M. N. (2008). Maths à modeler: Pavages par des dominos. *Grand N*, *82*, 53–68.
Grenier, D., & Payan, C. (1998). Spécificité de la preuve et de la modélisation en mathématiques discrètes. *Recherches en didactique des mathématiques*, *18*(2), 59–100.
Gribomont, P., Ribbens, D., & Wolper, P. (2000). Logique, automates, informatique. In F. Beets & E. Gillet (Eds.), *Logique en perspective: Mélanges offerts à Paul Gochet*, Ousia (pp. 545–577).
Gueudet, G., Bueno-Ravel, L., Modeste, S., & Trouche, L. (2017). Curriculum in France. A national frame in transition. In *International perspectives on mathematics curriculum, research issues in mathematics education series*. IAP.
Hart, E. W. (1998). Algorithmic problem solving in discrete mathematics. In L. J. Morrow & M. J. Kenney (Eds.), *The teaching and learning of algorithm in school mathematics, 1998 NCTM Yearbook* (pp. 251–267). Reston, VA: National Council of Teachers of Mathematics.
Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*(10), 576–580.
Hopcroft, J., Motwani, R., & Ullman, J. (2007). *Introduction to automata theory, languages, and computation* (3rd ed.). Addison-Wesley.
Howson, A. G., & Kahane, J. P. (Eds.). (1986). *The influence of computers and informatics on mathematics and its teaching, international commission on mathematical instruction edn*. ICMI Study Series. Cambridge University Press.
Kahane, J. P. (2002). *Enseignement des sciences mathématiques : Commission de réflexion sur l'enseignement des mathématiques : Rapport au ministre de l'éducation nationale* (cndp ed.). Paris: Odile Jacob.
Lovász L (2007) Trends in mathematics: How they could change education? In *Conférence européenne "The future of mathematics education in Europe"*, Lisbonne.
Meyer, A., & Modeste, S. (2018). Recherche binaire et méthode de dichotomie, comparaison et enjeux didactiques à l'interface mathématiques - informatique. In *Proceedings of EMF*, Paris, France (to appear).
Modeste, S. (2012). Enseigner l'algorithme pour quoi ? Quelles nouvelles questions pour les mathématiques ? Quels apports pour l'apprentissage de la preuve ? Ph.D. thesis, Université de Grenoble.
Modeste, S. (2013). Modelling algorithmic thinking: The fundamental notion of problem. In *Proceedings of CERME 8*, Antalya (Turkey).
Modeste, S. (2016). Impact of informatics on mathematics and its teaching. In F. Gadducci & M. Tavosanis (Eds.), *History and philosophy of computing* (pp. 243–255). Cham: Springer International Publishing.

Morris, C. W. (1938). Foundations of the theory of signs. In *International encyclopedia of unified science* (pp. 1–59), Chicago University Press.

Ouvrier-Buffet, C. (2014). Discrete mathematics teaching and learning. In *Encyclopedia of mathematics education* (pp. 181–186).

Perrin, D. (2007). L'expérimentation en mathématiques. *Petit x*, *73*, 6–34.

Quine, W. V. (1950). *Methods of logic*. Harvard University Press.

Reynolds, J. C. (1998). *Theories of programming languages*. Cambridge University Press.

Sedgewick, R., & Flajolet, P. (2013). *An introduction to the analysis of algorithms*. Pearson Education.

Sinaceur, H. (1991a). Corps Et Modèles: Essai Sur l'Histoire de l'Algèbre Réelle. Vrin.

Sinaceur, H. (1991b). Logique: mathématique ordinaire ou épistémologie effective? In: Hommage à Jean-Toussaint Desanti, Trans-Europ-Repress.

Straubing, H., & Weil, P. (2012). An introduction to finite automata and their connection to logic. In P. S. Deepak D'Souza (Ed.), *Modern applications of automata theory* (pp. 3–43). IISc Research Monographs: World Scientific.

Tarski, A. (1933). The concept of truth in the languages of the deductive sciences. Prace Towarzystwa Naukowego Warszawskiego, Wydzial III Nauk Matematyczno-Fizycznych 34(13), 172–198 (English translation in Tarski (1983)).

Tarski, A. (1936). On the concept of logical consequence. *Przegląd Filozoficzny, 39*, 58–68 (English translation in Tarski (1983), pp. 409–420)

Tarski, A. (1941). *Introduction to logic and to the methodology of the deductive sciences*. Oxford University Press (reedited in Tarski (1995))

Tarski, A. (1943). The semantic conception of truth and the foundations of semantics. *Philosophy and Phenomenological Research*, *4*(3), 341–376.

Tarski, A. (1954). Contributions to the theory of models, I–II. *Indagationes Mathematicae*, *16*, 572–588.

Tarski, A. (1955). Contributions to the theory of models, III. *Indagationes Mathematicae*, *17*, 56–64.

Tarski, A. (1983). *Logic, semantics, metamathematics: Papers from 1923 to 1938*. Hackett (J. H. Woodger, trans. Introduction: J. Corcoran).

Tarski, A. (1995). *Introduction to logic and to the methodology of deductive sciences*. New York: Dover Publications, INC. (unabridged Dover republication of the edition published by Oxford University Press, New York, 1946)

Thomas, W. (1997). Languages, automata, and logic. In *Handbook of formal languages* (pp. 389–455). Springer.

Vergnaud, G. (2009). The theory of conceptual fields. *Human Development*, *52*(2), 83–94.

Wittgenstein, L. (1921). Logisch-philosophische abhandlung. *Annalen der Naturphilosophie, 14* (English translations in C. K. Ogden trans. Routledge & Kegan Paul (1922) and D. F. Pears and B. F. McGuinnes, trans. Routledge (1961)).