Programming systems: in search for historical and philosophical foundations

Liesbeth de Mol, Giuseppe Primiero

Chapter 1
Programming systems: in search for historical and philosophical foundations

Liesbeth De Mol, Giuseppe Primiero

"Today we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes – build the whole thing, push it off the cliff, let it crash, and start over again." (Graham 1968)

1.1 Methodological Background

The purpose of this book is to engage with historical and philosophical issues underpinning, what we identify here, as 'programming systems', viz. large systems that have been programmed in order to control some process or set of processes.

In a recent paper published in the *Communications of the ACM*, we read the following assessment of the state of modern computing systems, [Neumann, 2017, p. 3]:

> *"Unfortunately, the trends for the future seem relatively bleak. Computer system trustworthiness and the implications of its absence are increasingly being questioned. Semi- and fully autonomous systems, the seemingly imminent Internet of Things, and artificial intelligence are providing further examples in which increasing complexity leads to obscure and unexplainable system behavior. The concept of trustworthiness seems to becoming supplanted with people falsely placing their trust in systems and people that are simply not trustworthy – without any strong cases being made for safety, security, or indeed assurance that might otherwise be found in regulated critical industries".*

L. De Mol
CNRS, UMR 8163 Savoirs, Textes, Langage
Université de Lille
e-mail: liesbeth.demol@univ-lille3.fr

G. Primiero
Department of Computer Science, Middlesex University London
e-mail: G.Primiero@mdx.ac.uk

Trustwortiness of large systems is just one of a growing number of serious problems related to computing, with the potential to affect millions of lives.[1] This is due not just to properties of the systems themselves but also their use, design and development by humans. On the one hand, these systems are ubiquitous, both in terms of usage and impact: almost everyone in large part of the developed world interacts constantly with a computing device; also, some of these systems have progressively evolved into cyber-physical entities, capable of acting upon and being affected by the external environment. On the other hand, though, there is an obvious mismatch between the complexity and ability of these systems to act in our world, and the level of knowledge required to interact with them intelligably and so critically. While this possibility of use without knowledge and understanding of these systems has been a major factor in their diffusion, it also has the important drawback that computing systems are nowadays used mostly by people who are unaware of the risks and consequences involved. Additionally, the increasing complexity and size of those systems, which is often rooted in a historically accumulated set of layers of abstraction and so-called bloated software systems [Wirth, 1995], has only deepened the issues of software design, development and maintenance as they came to be known in the 1960s (see Sec. 1.3). By consequence, it has become more difficult to prevent (potentially disastrous) errors. While this is principally a technical concern, it involves also political and commercial aspects underpinning the design, production and distribution of computing systems. From the point of view of the social and political implications, suffice here to mention issues of accountability in algorithm design and privacy of users.[2]

Given these circumstances, we are very much in need of a deeper reflection on the nature of computing systems. A methodological safe ground for such an investigation into the foundations of computing would require us to have a clear understanding of the field in itself, of the relations among its several sub-fields, and a solid grasp of how different approaches interact in the development of complex systems. There is, however, no such well-defined and clean foundation for the computing field in general: as it was argued in [Tedre, 2015], there is not even a clear and coherent identity. This is rooted, on the one hand, in the fact that computing has not yet reached its maturity as a discipline and, on the other, that it is both a science and a technology, with often different and sometimes conflicting interests. While science aims at stable, durable and solid results, technology is driven by the need of quick innovation and even quicker market returns. As Kalanick, former CEO of Uber, has recently remarked in the context of discussions on self-driving cars:

[1] This is a long standing issue in computing, touching on several areas. One of the early and most broad views on computing, risk and trust can be found in [MacKenzie, 2004]. Recently, the area of computational trust has grown sensibly in its impact and applications, from software packages distribution systems to vehicular networks, see e.g. [Primiero, Boender, 2017, Primiero et al., 2017] for some approach and overviews of the related literatures. For a high-level commentary on trust of digital technologies, see [Taddeo, 2017].

[2] The issue of algorithm accountability is gaining much traction, especially in view of current progress in AI. For a recent high-level analysis of the problem, see [Diakopoulos, Friedler, 2016]. For contributions concerning the debate on the ethical relevance of algorithms in terms of accountability and their public impact, see [Mittelstadt et al., 2016] and [Binns, 2017].

"*We are going commercial [...] This can't just be about science.*".[3]

The DHST/DLMPST Commission for the History and Philosophy of Computing (www.hapoc.org) was established in 2013 with the awareness that such a fundamental reflection on the computing field can only be possible through the interaction and dialogue among different expertises. The approach of the Commission is to create opportunities for collaborations and discussions within a pluri-disciplinary and pluri-methodological group of researchers, engaging with both the history, the philosophy and the formal and technical aspects of computing. We are strongly convinced that it is only by being embracive and tolerant with respect to different viewpoints, methods, and topics that it will be possible to develop a history and philosophy of computing which can account for both the scientific, social and technological aspects of the discipline.

Among others, one of the series of events organized by the Commission is the *Symposia on History and Philosophy of Programing* (HaPoP). The third in this series, HaPoP-3 was organized on June 25, 2016 at the *Conservatoire des Arts et des Métiers*, Paris by Liesbeth De Mol, Baptiste Mélès, Giuseppe Primiero and Raphaël Fournier-S'niehotta. Contrary to previous editions, this meeting focussed on one particular topic, namely on the nature, problems and impact of *operating systems*. The present volume collects contributions to HaPoP3.

Operating systems historically resulted from a broad set of general problems related to a large variety of aspects of computing (languages, memory, task complexity, to name a few) and so can be understood and contextualized as a (partial) answer to some of those problems. Moreover, both from the contemporary and historical perspective, it is hard to strictly isolate operating systems from others they are closely connected to, like networks and hardware systems. Accordingly, the editors have decided to shift the focus of the current volume to *programming systems*, to underline both the presence of historical aspects that precede and follow the birth of what qualifies as an operating system, the programming practices that underpin their design and development and the need to account for extensions of the concept of operating systems that we are witnessing today.

The general methodological approach of this book fits with the HaPoC philosophy. Accordingly, the current volume includes both papers which are motivated by conceptual issues or questions alive in the contemporary debate but which have roots in early episodes of the history of computingwhereas other more historical contributions bring to the fore fundamental problems which are still pressing today. In other words, the pluralistic approach of this book, allows and even necessitates to overstep boundaries between communities and it is our hope that this effort will engage researchers from the different communities to advance the very much needed foundations of computing.

[3] Quoted in [Chafkin, 2017]

1.2 Introducing Programming Systems

The term 'program' can have different meanings, and the historical context taken as a starting point for the origins of the activity of 'programming' largely affects the accepted definition. For the present purposes, we chose to start with ENIAC, since this is the historical context in which our contemporary use of 'program' originates.[4] ENIAC, one of the so-called first computers, was unveiled to the public in 1946. This machine, in its initial configuration, had two fundamental properties that no other computing device had at that time:

1. it was an electronic machine, and so computation was done at a very high, humanly impractical speed;
2. it was programmable, i.e. it could be set-up to compute any function within the material limits of the machine.

It was this combination of high-speed and programmability that required a deeper reflection on both the design of the machine and the 'art' of programming it – two strongly connected aspects for early ENIAC, where 'programming' meant physically rewiring the machine: there was a large gap between the time required to prepare and set-up a program and its execution time; unlike other contemporary machines like the Mark I, it was no longer possible to 'follow', and hence to fully control a computation; finally, it made no sense to provide 'code' through the mechanically and slowly punched cards. The answers to these issues were twofold: first of all, ENIAC was permamently rewired as a stored-program machine and a new design, known generically as the EDVAC or von Neumann design, was described; secondly, different approaches for controlling "*the automatic evolution of a meaning*" [Goldstine, 1947] were developed.[5]

The ENIAC was a one-of-a-kind machine and by the late 1940s-early 1950s the standard design had stabilized on the EDVAC design [Neumann, 1945][6] and the stored-program.[7] The latter is today considered as the stabilising technical and conceptual element from which 'programming systems' became possible: the stored-

[4] See [Grier, 1996]. Obviously, an ENIAC program is something quite different from a program expressed in a high-level language. See [Haigh, 2016b] for a different approach in which one starts from a generic definition of 'program' (as a 'sequencing of operations'). In that work then, a 'modern program' is rooted in the ENIAC machine and EDVAC design.

[5] More particularly, the approach taken by von Neumann was the identification of different steps in the preparation and set-up of a problem – a kind of division of labor – where the most prominent stage is that in which the 'dynamics' of a program is captured by means of a flowchart. The other is due to Curry, who focussed on the automation of the coding process and developed a logic for program compositionality. See [De Mol, 2015] for a partial comparison between the two approaches.

[6] It should be added that this is the standard narrative. Of course, there were many variants on the EDVAC design and also entirely different designs such as that for the Whirlwind which was not serial. See also [Backus, 1978] for a critical discussion of the von Neumann method.

[7] There are different understandings of the origins of the stored-program concept, its intellectual lineage and its historical implementation and understanding. See [Haigh et al., 2014] and [Copeland, Sommaruga, 2015] for two different interpretations.

program concept expresses the basic principle of computer science that programs and data are interchangeable and granting, ultimately, the possibility to "*[simplify] the circuits at the expense of the code*" [Turing, 1946]. As it will become clear from several contributions to this volume, contrary to what Goldstine and von Neumann believed, i.e. that:

> *"the problem of coding routines need not and should not be a dominant difficulty. [In] fact we have made a careful analysis of this question and we have concluded from it that the problem of coding can be dealt with in a very satisfactory way."* [Goldstine, 1946]

programming problems would not be resolved nor absolved by this basic principle, nor by initial symbolizations of the general flow of a program.

Once the design of computing machines had more or less stabilized, the construction of computing machinery moved away from the research labs at the university to industry, and so commercial interests started to play their role. However, for both scientific (e.g. SWAC) and business-oriented applications (e.g. in the context of LEO machines), and thus also for machines used in both contexts (IBM and Burroughs), there remained an important set of programming and physical problems to be resolved. Originally, computers were coded through machine instructions and so the semantic gap between "code" and hardware execution was quite small. However, this coding through stacks of punched cards or tape was a highly time-consuming and very error-prone process. Another associated issue was developing at the hardware level: the need of increasingly complex sets of instructions to execute meant the need for greater amounts of memory and the ability to centralize the different instruction control in one physical unit. In the 1950s, these two problems were tackled in a timely fashion and almost in parallel, as analysed in Part I of this volume.

First, one sees the development of techniques to optimize the coding process: while, at first, these were mainly developed and used in one particular practice and around a specific hardware system, there were clearly attempts at more systematic approaches. One well-known example is the programming book for the EDSAC [Wilkes, 1951] which is basically a 'library' of more or less standardized subroutines and, in its reprinted version, wanted to transcend the particularities of EDSAC to be a "*general introduction to programming for any computer of the stored-program type*". These approaches went hand-in-hand with approaches to improve on the hardware design to have more efficient and simpler coding with fewer errors or better error-handling.[8] The example of the LEO machines presented in Chapter **??** is illustrative of these initial more systematic attempts. More particularly, it focuses on how approaches are developed to identify and resolve errors in a fashion that anticipates the definition of principles of correctness inspiring modern research in program verification. These efforts were strongly characterised by the business nature of the Lyons company who developed the machine: for this reason, more formal

[8] See for instance the development of microprogramming which is basically anm approach of hardware programming [**?**]: "*This paper describes a method of designing the control circuits of a machine which is wholly logical and which enables alterations or additions to the order code to be made without ad hoc alterations to the circuits*" The paper [De Mol, 2017] discusses several machines, some of which fit into the microprogramming strategy, that stick close to the hardware and develop optimum coding techniques such as latency and underwater programming.

principles of valid program execution (like ensuring termination) are accompanied by heavily pragmatic choices (e.g. in the way program were designed and tests performed). Second, the improvement on hardware was essential not only in reducing the risks associated with component failures, but in particular in guaranteeing the possibility of accomodating a lot more memory. Complementary to these more systematic approaches that remain, basically, in the realm of the order, assembler and machine code, the first steps are being taken towards the development of higher-level languages and techniques of 'automatic coding' which were aimed at relieving the programmer from the tedious coding task and so to 'automate' the programmer.[9] So, for instance, Grace Hopper made the first steps for automating subroutining for the A0-language of UNIVAC I.[10] Chapter **??** is to be set against the background of automating aspects of the programming process. More particularly, it focuses on the period 1954-1964, the decade before the term 'operating system' has more or less stabilized. This chapter deconstructs a classic narrative from the history of computig, viz. that the operating system is basically the IBM vision of automating the operator and so is historically located at the transition from batch processing (where the operator was human) to time-sharing systems. It is argued that this narrative in fact hides a more complex history which is about automating different aspects of programming. It is shown that it took several years before one could start to differenitate more clearly between different kinds of systems, including the operating system. More particularly, a taxomomy of different types of systems from the late 50s and early 60s is offered. It is within that taxomomy that the steady development of 'operating system' is seen and so, it becomes clear that the automation of the operator is just one of a set of parallel developments that brought about the distinguishability of operating systems from other systems.

1.3 The Complexity of Programming Systems

In the early 1960s, it is no longer mainly the hardware that shapes the problems related to the design and development of large-scale programming systems, but rather the programming systems themselves that determine the problems at stake. Indeed, as summarized by the developers of the AOSP operating system for the Burroughs D825, [Anderson, 1962]:

> "computers do not run programs, [...] programs control computers."

This underlies the diffuse realization that the problem does not lie with the machines, but rather with the 'programs', the way they are written, the way they constitute a complex system ready for commercial and scientific use by a broad range of different 'users' with different aims, who want well-documented, error-free and

[9] See for instance [Daylight, 2015] which discusses the approach of hte so-called 'space cadets'. See also [Nofre, 2014] for a discussion of the use of the notion of language in this context.

[10] See [Hopper, 1980] for a personal account of that development.

efficient systems. What is hard, is the software-side of computing.[11] From realizing that the programming problem was not simply going to be resolved by faster systems with larger memory, (the legend of) the software crisis was born. As summarized by Dijkstra in his Turing award lecture [Dijkstra, 1972]:[12]

> "instead of finding ourselves in a state of eternal bliss with all programming problems solved, we found ourselves up to our neckys in the *software crisis* [m.i.]! [...] To put it quite bluntly, as long as there were no machines, progamming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.[...] To put it in another way: as the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between ends and means."

The 'software crisis' is thus closely tied with the development of progressively larger and more complex systems. Two approaches can be identified as answers to the problem:

1. the formal approach, closely associated with a logical understanding of the foundations of computing;
2. the modernist approach, associated with the development of 'grand designs' that would provide universal environments for the solution of multiple programming problems.

The contributions in Part II of this volume provide an understanding of the first of the two approaches above. A very direct case of treating a programming system as a formal system is the development of programming semantics: these were introduced precisely to deal with issues related to mapping specification with implementation on possibly different machine architectures. Chapter **??** goes back to a basic historical case to discuss and compare four different styles of formal semantics that were developed in that context. More particularly, the paper focuses on the origins and problems of formal semantics that were developed for the case of Algol 60. It was considered a good language to demonstrate the potential of a logical approach since it was supposed to scale well to realistic languages. More particularly, the paper discusses: the Vienna operational description; Vienna functional description; Oxford denotational description and the VDM denotational description. For each style, the authors discuss not just the historical context, but engage with particular stylistic, syntactic as well as modelling features. This uniform approach for analyzing each of the styles not only results in a historical study of the reasons and modalities of their origins, but it also allows a more critical review of each semantical description of the language.

The formal approaches did not come about in a straighforward manner, but are based on a critical study of the foundations of programming and computing. Amongst others, they require an analysis of what constitutes a full-fledged computational object, including whether some 'objects' are going to have fewer rights than

[11] See [Mahoney, 2008] for a historical take on this wordplay.

[12] See [Haigh, 2010] and [Ensmenger, 2010] for two different interpretations of the impact of the so-called software crisis.

others, thus introducing a principle of non-uniformity. Chapter **??** engages with this problem, in particular with the possibility of treating functions as so-called first-class citizens. This basic problem in the foundations of programming, first pointed out by Strachey in the Algol context,[13] opens up a discussion on the technical and conceptual consequences of a particular formalization of computational citizenship; but it also connects developments in programming with the foundational debate in mathematics from the late 19th and early 20th century. Within this setting, the nature of operating systems and their coming about is investigated at an even higher level of abstraction than language: the system becomes an environment in which functions can be treated as computational objects. This understanding of an operating system has the advantage of generating the conceptual space for a number of other associated (both theoretical and technical) topics fully developed in the modern understanding of systems: execution privileges, access privileges, and the possibility of delegating those in different environments. It is especially interesting how establishing these traits of operating systems as environments of function definition and execution makes possible the convergence of both formal and technical discourse, in a move highlighting the foundation of both theoretical and physical computing.

The two contributions in Part III of this volume approach the mentioned 'modernist' take on the problem of system complexity. They both center their analysis on what is often considered on of the most successful of the grand design approaches to operating systems, namely Unix. In Chapter **??**, Unix is approached through its relations with other designs, both those that were supposed to improve it (like Plan 9), and those that had different philosophies (like Smalltalk). This analysis highlights a number of important features that have emerged as unifying traits in the process of system design: the focus on programmability as the main core-business of the system; the creation of a meta-system providing a unified semantic description for different types of objects (e.g. programs, files, devices); and its flexible 'everything is a file' design, allowing any program to be used with any file as input and any device as output. It is argued that both Unix and Smalltalk, while usually interpreted as 'grand designs', can also be aligned with a more postmodern understanding of programming in which there is not just one ultimate language but many where each offers its own "viewpoint". The operating system then becomes the backbone to support that postmodernism.

In Chapter **??**, Unix is set against the background of one of its predecessors, the Multics project: this detailed analysis of the features and processes in its early instantiation PDP-7 Unix, shows the switch from a 'bigger is better' approach to a 'simple is better'[14] one [Raymond, 2003]:

> "Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface".

[13] More particularly, during a talk titled *Fundamental concepts of programming languages* given at the International Summer School in Computer Programming in Copenhagen, in August 1967 [Strachey, 1967]

[14] Which is basically the 'worse is better' philosophy. See also **??**

The process of creating the first Unix system started in 1969 and its several versions were developed until the 1980s, while the success of later instantiations like Linux and McOS are well-known to everyone.

1.4 Programming Systems in the Real World

The discussion whether a real software crisis has been overcome, or whether we are witnessing a new one, is still very alive today. Only, the stakes are now much higher: energy grids, banking systems, border controls, medical appliances, traffic control and automated vehicles, polling systems, any single important aspect of our everyday life is managed by and relies on a programming system. Part IV of this book engages with these more recent developments focusing on ethical, political and even aesthetical issues of large-scale systems.

In Chapter **??**, the problem of defininig ethical principles for operating systems within a safety-critical setting is analysed. Note how this project relies on the very same idea that motivates the formal approach to computing from the previous Section 1.3: first, the authors seek to connect processing in an ethical cognitive calculi to a successful, proof-based analysis and verification at the OS level; second, this formal analysis is implemented in a language to demonstrate feasibility in a self-driving system. The importance of logically grounded, verifiable and formally reliable systems is expanding from purely academic research, to projects developed in major private players in the computing industry (Amazon and Facebook are particularly significative examples).

In Chapter **??** the relation between operating systems and globalization is examined from the point of view of sovereign nations, which are reconfiguring themselves as properly cyber-physical entities whose control extends to the software and data domains. The integration of state-sponsored and private software and hardware components is aimed at increasing control and at infringing user privacy: this aspect becomes nowadays essential in understanding the novel functional configuration of operating systems. The complexity of systems is thus again at stake in updating their definition, although this time with an additional level of influence, extending throughout the whole digital chain.

Finally, the definition of aesthetical criteria for complex systems relies inevitably on a compositional approach, almost matching the complexity analysis suggested by [Fetzer, 1988]. In this respect, a first step is made in Chapter **??**, where the problem of defining elegance of simple programs is tackled: this notion is further analysed in terms of properties depending both on abstract and pragmatic criteria. These necessarily include the program's 'fitness for purpose', a criterium that (again) recalls the correctness principles mentioned at the very beginning of this volume.

The present volume is the first volume ever published that combines historical, philosophical and technical approaches to tackle issues of programming systems.

Whereas there are some studies focussing on one of these approaches,[15] a combined approach, that allows to see different issues from multiple perspectives, was still missing from the literature. We consider this volume as a way to open up a very much needed foundational debate requiring the perspectives from historians, philosophers and practitioners: the former provide the historical backbone for current issues and so, amongst others, help to identify the 'real' issues from the more contingent ones; philosophers help discerning conceptual trajectories and the evolution of ides, like those of correctness and computational citizenship; finally, the practitioners give the technical and problem context in which those ideas and issues originated, were technically tackled and evolved.

Acknowledgements We would like to thank Baptiste Mélès and Raphaël Fournier-S'niehotta for their help with setting up and chairing HaPoP-3. We are also very grateful to the participants to the symposium as well as the PC members for their help in selecting the accepted talks. Finally, this book volume would not have been possible without the careful and critical reading of the different reviewers who helped to improve the contributions.

References

Anderson, J.P.; Hoffman, S.H.; Shiman, J and Williams, R.J. (1962). The D-825, a multiple-computer system for command & control. *1962 Fall Joint Computer Conference (AFIPS)*, pp. 86-96.

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, vol. 21, nr. 8, pp. 613–641.

Berry, D.M. (2011). *The Philosophy of Software – Code and Meditation in the Digital Age*. Palgrave MacMillan.

Binns, R. (2017). Algorithmic Accountability and Public Reason. *Philosophy & Technology*. https://doi.org/10.1007/s13347-017-0263-5.

Brennecke, A. and Keil-Slawik, R. (1996). *History of Software Engineering*, August 26-30 1996, Dagstuhl seminar 9635, organized by W. Aspray, R. Keil-Slawik and D.L. Parnas

Chafkin, M. (2016). Uber's first self-driving fleet arrives in Pittsburgh this month. *Bloomberg Businesweek, August 18, 2016*. .

Copeland, B.J., Sommaruga, G. (2015). The Stored-Program Universal Computer: Did Zuse Anticipate Turing and von Neumann?, In G. Sommaruga, T. Strahm (eds.), *Turing's Revolution*, pp. 43–101, Springer International Publishing Switzerland.

Daylight, E.G. (2015) Towards a historical notion of 'Turing-the father of computer science'. *History and Philosophy of Logic*, vol. 36, nr.3, pp. 205–228.

[15] For historical works, see [Brennecke, 2002] and [Hashagen, 2002]. For computer science works, see [Tanenbaum, 2008] and [Silberschatz, 2011]; for a philosophical approach to software with some aspects related to systems, see [Berry, 2011].

De Mol, L.; Carlé, M.; Bullynck, M. (2015) Haskell before Haskell. An alternative lesson in practical logics of the ENIAC. *Journal of Logic and Computation*, vol. 25, nr.4, pp. 1011-1046, A version is available from: http://hal.univ-lille3.fr/hal-01396482/document

De Mol, L.; Bullynck, M.; Daylight. E. (2017). Less is more in the Fifties. Encounters between Logical minimalism and computer design during the 1950s. Available from: hal.univ-lille3.fr/hal-01345592v2/document

Diakopoulos, N., Friedler, S. (2016). How to Hold Algorithms Accountable. MIT Technology Review. https://www.technologyreview.com/s/602933/how-to-hold-algorithms-accountable/.

Dijkstra, E. W. (1972). The humble programmer, *Communications of the ACM*, **15**, 859–866.

Ensmenger, N. (2010). The computer boys take over, MIT Press.

Fetzer, J. H. (1988) Program Verification: The Very Idea, *Communications of the ACM*, 31(9):1048-1063.

Goldstine, H.H. and von Neumann, J. (1947) Planning and coding of problems for an electronic computing instrument Volume 2 of *Report on the mathematical and logical aspects of an electronic computing instrument*, part I,II and III, 1947-48. Report prepared for U. S. Army Ord. Dept. under Contract W-36-034-ORD-7481.

Goldstine, H.H. and von Neumann, J. (1946) On the principles of large-scale computing machines. in: Aspray, W. and Burks, A (eds.), *Papers of John von Neumann on computing and computer theory*, MIT Press, 1987, pp. 317–348.

Grier, D.A. (1996) The ENIAC, the verb 'to program' and the emergence of digital computers *IEEE Annals for the history of computing*, vol. 18, nr.1, pp. 51–55.

Haigh, T. (2010). Dijkstra's crisis: The end of Algol and the beginning of software engineering: 1968-72' in: *Workshop on the history of software, European styles*, Lorentz Center, University of Leiden.

Haigh, T., Priestley, M., Rope, C. (2014). Reconsidering the Stored-Program Concept. *IEEE Annals of the History of Computing*, vol. 36(1), pp. 4-17.

Haigh, T.; Priestley, M.; Rope, Cr. (2016). *Eniac in action. Making and remaking the modern computer*. MIT Press.

Haigh, T.; Priestley, M. (2016). Where Code Comes From: Architectures of Automatic Control from Babbage to Algol *Communications of the ACM*, vol. 59, nr. 1, pp. 39–44.

Hashagen, U; Keil-Slawik, R.; Norberg, A.L. (eds.) (2002) *History of Computing: Software Issues*. Springer.

Hopper, G. (1980). Keynote address Wexelblat, R.L. (ed.), *History of Programming Languages*, ACM Press, pp. 7–24.

MacKenzie, D. A. (2004). Mechanizing Proof – Computing, Risk, and Trust. MIT Press.

Mahoney, M. (2008). What makes the history of software hard? *IEEE Annals for the history of Computing*, vol. 30, nr.3, pp.8–18.

Mittelstadt, B.D., Allo, P., Taddeo, M., Wachter, S., Floridi, L. (2016). The ethics of algorithms: Mapping the debate. *Big Data & Society*, July-December 2016, pp. 1-21.

Neumann, P. G. (2017). Trustworthiness and Truthfulness are essential. *Communications of the ACM*, vol. 60, nr.6, pp. 1–3.

Nofre, D.; Priestley, M. and Alberts, G. (2014) When technology became language: the origins of the linguistic conception of computer programming, 1950-1960. *Technology and Culture*, vol. 55, nr. 1, pp. 40–75.

Primiero, G., Boender, J. (2017). Managing Software Uninstall with Negative Trust. In Jan-Philipp Steghöfer and Babak Esfandiari (eds.), *Trust Management XI - 11th IFIP WG 11.11 International Conference, IFIPTM 2017*, IFIP Advances in Information and Communication Technology, vol. 505, pp.79–93, Springer.

Primiero, G., Raimondi, F., Chen, T., Nagarajan, R. (2017). A Proof-Theoretic Trust and Reputation Model for VANET. In 2017 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2017, pp.146–152.

Raymond, E.S. (2003). *The art of Unix programming*. Addison-Wesley Professional.

Silberschatz, A.; Galvin, P.B.; Gagne, G. (2011). *Operating system concepts* Wiley and Sons.

Strachey, C. (1967). Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 2000, vol. 13, pp. 11-49.

Taddeo, M. (2017) Trusting Digital Technologies Correctly. *Minds & Machines*. https://doi.org/10.1007/s11023-017-9450-5.

Tanenbaum, A.S. (2008) Modern Operating Systems. Pearson International Edition, 3rd. edition.

Tedre, M. (2015). *The science of Computing. Shaping a discipline*. Boca Rato: CRC Press.

Turing, A. M. (1946) Lecture to the London Mathematical Society on 20 febru- ary 1947. 1947. in: Brian E. Carpenter and Robert W. Doran (eds.), *A.M. Turings ACE Report of 1946 and Other papers*, MIT Press, 1986, 106-124.

von Neumann, J. (1945) *First draft of a report on the EDVAC*, University of Pennsylvania, June 30, 1945. Available from: http://www.virtualtravelog.net/entries/2003-08-TheFirstDraft.pdf.

Wilkes, M.V.; Wheeler, D.J. and Gill, S. (1951) *The preparation of programs for an electronic computer* Addison-Wesley Second edition, 1967.

Wilkes, M. and Stringer, B. (1953) Micro-programmingand the design of the control circuits in an electronic digital computer *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 49, nr. 2, pp. 230–238.

Wirth, N. (1995) A plea for lean software *Computer*, vol. 28, nr. 2, pp. 64–68.