Enterprise Modelling and Information Systems Architectures
Vol. 15, No. 6 (2020). DOI:10.18417/emisa.15.6

Informatics as a Science

Wolfgang Reisig

Humboldt-Universität zu Berlin, Berlin, Germany

Abstract. *This contribution addresses the quest for a framework for a comprehensive science of "informatics" as a formal theory of discrete dynamic systems, in analogy to the model of natural sciences. A variety of examples show that this endeavor is promising indeed, and that (detached) parts of it exist already. In the long run, informatics may evolve as a self-contained science, more comprehensive than nowadays "Computer Science", by complementing its strong technological aspects with a consistent theoretical, mathematical basis, on an equal footing with natural sciences.*

Keywords. comprehensive science of informatics • formal foundations • modeling

Communicated by Peter Fettke. Received 2019-08-30. Accepted on 2020-01-20.

Introduction

Background

Computer Science is frequently told as a success story, driven by Moore's law: during the last six decades; computation-, information- and communication technology became exponentially cheaper, quicker and smaller (Brock 2006). Accordingly, new application areas evolved. Systemic malfunction of devices, failed projects, unmanageable behavior of computer embedded systems, violation of privacy etc., are accepted as unavoidable side effects and justified as the price to be paid for the rapid evolution of the area.

Whether matters might – or should – have evolved differently, is rarely discussed and hardly suspected. The quest for alternatives to the factual evolution of Computer Science is annoying in particular for young scientists, who eagerly learned facts, and want to build on this presumed solid and steadfast basis. They are squeezed in systems of promotion and reward, that don't support fundamental questions on the nature of informatics.[1]

In this situation it appears nevertheless interesting to pose some fundamental questions and to discuss aspects that could unify the diverging subdomains of Computer Science. As a basis for a science of informatics, this may render future developments quicker and better comprehensible.

Informatics as a science: historical development

The emerging computing technology of the 1950ies covered essentially two problem domains: numerical problems as they occur in classical engineering sciences, and searching-and-sorting problems in large data sets, as they occur in a population census. The two programming languages FORTRAN and COBOL had been tailored to those problems; the steps from a problem to an algorithmic idea and to a program were usually manageable. This changed in the 1960ies: increasing computer power made problems solvable that required more complex algorithmic ideas and more elaborate programs. This rendered programs

E-mail. reisig@informatik.hu-berlin.de
I owe many valuable comments to readers of a previous German version of this text: Dines Bjørner, Peter Fettke, Christoph Freytag, and Otthein Herzog.

[1] By "Computer Science" we refer to the traditional approach, whereas "informatics" denotes the to-be-developed science.

increasingly error prone and incomprehensible, and finally a "software crisis" had been identified at a famous conference in Garmisch-Partenkirchen (Naur and Randell 1969), to be tackled by the new discipline of "software engineering". This in turn caused new programming languages (PL/1, Pascal, Ada), new program paradigms (structured programming, object orientation) and later on new software architectures (components, service-orientation, micro-services). Furthermore, software design methodologies emerged, such as systematic refinement, software architectures, and the specification of interfaces.

Looking back, it is obvious that the Garmisch-Partenkirchen conference raised important questions and initiated constructive answers. However, on the long way from a problem via an algorithmic idea up to a software, only the last step, i. e. writing down programs, has been framed more convenient by the concepts suggested there.

This way of thinking about "Computer Science" as a science is visible at the memorial lecture for famous Edsger W. Dijkstra in 2010, where likewise famous computer scientist Tony Hoare emphasizes four criteria for sciences in general, and the issue of Computer Science in particular (Hoare 2010):

- *Description*: Science describes properties and behavior of systems; in Computer Science, such a "description of properties and behavior can serve as a specification, describing an appropriately precise interface between its purchaser and its supplier." Hoare suggests logic based description methods, with weakest preconditions as an example.
- *Analysis*: The central items and their conceptual relation of a science are detailed here. Central items of Computer Science are programs. They are analyzed with pre-/postconditions such as Microsoft Contracts.
- *Explanation*: The behavior of systems is substantiated. For Computer Science, the semantics of programming languages provides this substantiation.
- *Prediction*: Good science can predict the evolution of processes. In Computer Science, this means to predict the behavior of programs, i. e. to verify them.

Summing up, a program is conceived as a mathematical item; its decisive properties should be formulated in a logical language, its semantics should be formally captured, and its correctness should be proved. These are scientific concepts, indeed. In this spirit, also Gries (1981) conceives the activity of programming as a scientific activity.

This contribution

In 14 sections we formulate some thoughts for a comprehensive science of informatics as a formal scientific theory of discrete dynamic systems. The subject of this science is intended much broader than the above outlined traditional "Computer Science", as suggested by Dijkstra, Hoare, Gries, Knuth and others. Nevertheless, the new, broader theory should be conceptualized as a *formal* theory; not as classical Computer Science extended by informal aspects of social sciences. Examples include formal concepts of information processing and algorithmic behavior that are not intended to be implemented, as they occur in business informatics, in embedded systems, and generally in the nowadays much discussed "internet of everything".

We will pose questions and explain some mutually related concepts. Some of them are not fundamentally new, but have frequently been studied in isolation. Properly combined, they support the claim to contribute to a new science of informatics.

However, we don't present a fully-fledged science of informatics. Obviously, a number of additional ideas, concepts and insight is still missing. We just want to show that it is worth searching for a fundamental science of informatics, and to stimulate discussion.

1 Successful construction of scientific models

Good scientific theory is formulated in terms of *models*. A model is a system of notions and relations among them, intended to better understand reality. "Reality" is either given as natural phenomena, or – as in the case of informatics – constructed by man. Particularly impressive is the example of physics: For centuries, physicists searched a unifying model for all branches of physics, setting up an impressive body of scientific theory. Particularly impressive is the deep harmony between physics and mathematics (Livio 2009) with very abstract, yet exceedingly useful scientific concepts and models. A typical example is the notion of *energy*. This notion allows to mutually relate and quantitatively fix quite different phenomena that, for example, occur when a car is accelerated and crashed into a wall: In this process, an amount of energy is involved. It is initially contained in petrol, then in acceleration, and finally in reshaping metal. The notion of "energy" is useful because it provides a quantifying *invariance* for dynamic processes. This kind of invariants, usually denoted as "laws of nature", are the scaffold of science. In recent years, systems biology likewise is searching for such notions and invariants to better understand metabolic processes.

Informatics should learn from physics and other sciences how to establish a comprehensive theoretical framework for all of its aspects. Already in 1962, John McCarthy encouraged research into a mathematics-based "Science of Computation", evolving its central items and properties out of a few postulates (McCarthy 1962).

2 Models in informatics

As *modeling* is the center of scientific theory building: what are the models of informatics? Just as many (natural) sciences, informatics is about *dynamic* systems, however with a fundamental difference: Physics describes behavior usually in continuous terms, with functions over the real numbers. This allows to construct differential equations, integrals, etc. In contrast, informatics describes behavior in discrete steps. This allows to describe entirely different properties, and requires different analysis techniques.

Models in informatics are used for various different purposes:

- to describe domain-specific facts, for example a companies' structure of the book keeping and the accounting department (the "correctness" of this kind of descriptions remains inevitably informal);
- to describe algorithmic behavior, for example how to apply for a claim settlement with an insurance company;
- to describe a software's effect, either in terms of properties or of its operational behavior.

A model is a symbolic description. It may be executable on the symbolic level, in which case its behavior can be simulated on a computer. But it also may conceptually describe the behavior of a system in a concrete domain, that is not intended to be implemented. With a really useful science of informatics it will be worthwhile to discuss correctness on the model level and then to systematically derive correct software.

Seamless integration of computing technology with its technical or organizational environment is inevitable for many computer integrating systems; not the least the *internet of things*: such systems are manageable only with formal models that include computing technology as well as its environment. This includes organizational or technical components that are not intended to be implemented.

Various modeling techniques support this view, in particular the UML (Booch et al. 2005) and – primarily for business informatics – BPMN (Object Management Group 2011). These techniques suggest various graphical means to express particular, subtle, domain specific aspects. Both these techniques (and some more, e. g. Harel's statecharts) are widely applied, in deed. But they offer only rare specific means to prove properties of the modeled systems. Even if systems are modeled

by means of those techniques before implementation, correctness is usually studied (tested) on the software level.

Modeling is a central issue of formal methods such as, besides many others, ALLOY, B, Focus, Live Sequence Charts, RAISE, TLA, VDM and Z. The main concern of these methods is a systematic way to construct correct software (Bjørner and Havelund 2019). These methods, however, insist in implementability, at the price of convenience and adequacy of modeling the reality. This problem is, to some extent only, tempered by domain specific methods (Bjørner 2018). General modeling methods with their focus on the realm to be modeled, include ASM (Gurevich 2000) and Petri nets (Reisig 2013). This focus supports, for example, business informatics: there, behavior of components are modeled, that are not intended to be implemented (Bichler et al. 2016). The term "conceptual modeling" refers to models, mainly for business applications, including components that are formally described, but not necessarily implemented. A typical example is the "Open Models initiative" (Karagiannis et al. 2016). The quote "Computer Science is no more about computers than astronomy is about telescopes", attributed to E.W. Dijkstra, may apply here.

The idea of system models that focus application areas more than implementation, is rarely addressed. A typical example of this narrow view is the Dagstuhl seminar on the history of software engineering in 1996 (Brennecke and Keil-Slawik 1996).

From a scientific perspective, in the long run, modeling techniques with adequate more expressivity for application domains, combined with strong analysis techniques, are urgently needed. They may look quite different from what is available now.

3 Trustworthy models

In a general, systematic buildup of modeling principles, it should be possible to find a – formal – modeling technique that allows to describe a system trustworthy, comprehensible, and unambiguously. For a given instance we usually have an intuitively clear understanding of what an appropriate description of a system is about: It contains all aspects that the modeler considers relevant, and it does not enforce aspects that the modeler wants to ignore. More concretely formulated, a trustworthy model M of a discrete system S describes

- each elementary item of S as an elementary item of M,
- each elementary operation of S as an elementary operation of M,
- each composed item of S as a composed item of M,
- each composed operation of S as a composed operation of M,
- each – local – state of S as a state of M,
- each – local – step of S as a step of M.

Summing up: elementary and composed items and operations, as well as states and steps of S and M should correspond bijectively. Intuitive and formally represented behavior should conform as tightly as possible.

This rises the quest for a modeling technique that would meet these requirements, at least for a large and interesting class of systems. Obviously, such items, operations, states and steps do not fit into the classical scheme of computability theory. This has been observed and discussed many times, beginning perhaps with Donald Knuth's fundamental *The Art of Computer Programming* (Knuth 1973), where the notion of computational methods (nowadays: *transition systems*) is suggested as a general framework for the notion of algorithms. A computational method consists of a set S of states and a next-state function F on S, where *"F might involve operations that mortal man can't always perform."* (Knuth 1973, p. 9). Knuth denotes a computational method effective, in case F is a computable function. Robin Milner in his EATCS award lecture (Milner 2005) declared: *"... we should have achieved a mathematical model of computation, perhaps highly abstract in contrast*

with the concrete nature of paper and register machines, but such that programming languages are merely executable fragments of the theory . . . ". It remains open, which kind of non-executable fragments Milner has in mind.

The classical framework of computation with variables and operations over strings of symbols, as well as programs with sequences of assignment statements, alternatives, and conditional loops has frequently been generalized to cover freely chosen mathematical objects and operations, as in Tucker and Zucker (2000) and Dershowitz (2012). Shepherdson (1995) devises a similar idea, based on Turing Machines. In a slightly different style, and from a purely logical perspective, Gurevich (2000) suggests Abstract State Machines as programs over a signature (a sorted alphabet). The user of this formalism may freely chose his signature as well as its interpretation (a structure), and systematically manipulate the structure by help of terms generated from the signature.

Summing up, a good model employs symbols that in the modeled realm are interpreted as items or operations. This fundamentally differs from programming: A programming language fixes the interpretation of the employed symbols.

4 Invariants in informatics

As outlined in Sec. 1, invariance is a pillar of scientific theories. This solicits the quest for notions of invariance in informatics. The best known such notion is certainly Hoare's invariant calculus (Hoare 1969) with the concept of loop invariants for classical programs. According to Furia et al. (2012), loop invariance is one of the fundamental ideas of software design. This may be true if the notion of correctness of software design is bound to its very end, i. e. the coding in a classical programming language. A science of informatics should however focus the correctness of models. In fact, many modeling techniques employ particular versions of invariants: for example, Tel (2000) suggests special invariants for distributed processes, distributed algorithms, and communication protocols. The invariant calculus for Petri nets is exceedingly expressive (because Petri net transitions are reversible) (Reisig 2013).

Each such invariant declares a relationship among the variables of the model that holds in each reachable state. Those notions stick close to the items of the given model. As shown by the example of the notion of energy in Sec. 1, physics knows much deeper, less obvious but nevertheless (or therefore) useful invariants. Informatics should strive at comparatively deep invariants. Such invariants might make precise what remains invariant in the case of

- a bank client, withdrawing cash at a cash machine;
- a life insurance, transferring a client's policy to a new hardware;
- a car insurance, regulating a damage;
- a computing center, simulating tomorrow's weather;
- an operating system, executing garbage collection;
- a compiler, translating a program;
- a travel agent, booking a journey.

There are presently no modeling techniques with such invariants. Nevertheless, it is useful to systematically search such techniques that, compared to so far existing concepts, evolve much more abstract and less obvious, yet useful notions of invariants.

5 A fundamental notion of "information"

The above example of the notion of "energy" shows that invariance can be based on a very abstract, yet intuitively conceivable notion. Are there similar such notions for informatics? In the sequel I try to outline a notion of "information", with the central invariant of information preservation in local steps: in a given state, a system contains a distinguished amount of "information". As long as the system does not communicate with its environment, this information can be converted in different ways; it can be newly combined; something can be derived (computed) from it; parts of

it can be made inaccessible, etc. But the amount of information as a whole remains invariant. A computation, i. e. a sequence of steps, then models a *strictly organized flow of information*.

A constructive definition of such a notion of information is not in sight. Nevertheless, such a notion would be quite useful; for example to construct invariants according to Sec. 4. With such a notion it might be possible to formulate precisely what changes and what remains when documents are copied, deleted, or combined. Protection of data and privacy as well as related notions might gain a much more precise meaning.

A further interesting aspect of such a notion are *information preserving operations*: Such an operation, f, retains all information when applied to an argument a. Hence, a can be re-computed from the result $f(a)$. Examples of such operations include the negation if propositional logic, and the positive square root of positive real numbers. Reversible functions occur frequently in the context of electric circuits (Vitányi 2005). The consequent application of reversible computing may decisively boost IT security: An attacker can retrospectively be identified.

It might become realistic to define particular notions of "information" in terms of operations that are feasible or acceptable in specific contexts.

6 Interactive components

Classical theoretical Computer Science abstracts information-technological processes in terms of functions over symbol chains. With deep results on complexity theory and relations between logic and automata theory, this framework has been established as theoretical basis of Computer Science. In this context, Turing machines are the best-known model. A Turing machine, including a store and a processor, can be conceived as an adequate abstraction of computing technology of the 1960ies. A multiprocessor architecture, using more than one processor to increase computing speed, may more or less adequately be abstracted to an architecture with a single processor. In fact, a system consisting of many cooperating components where communication is the central issue, may be simulated on a single processor. This, however, would spoil the purport of the system.

Systems consisting of many cooperating components have been suggested in the 1980ies; for instance the *Actor* formalism of Agha and Hewitt (see Agha (1986)). Similar ideas are the basis of languages such as LINDA (Carriero et al. 1994) and the Chemical Abstract Machine (*CHAM*) (Berry and Boudol 1990). In this Metaphor, active elements are conceived as a kind of molecules, that are "swimming" in a – metaphorical – chemical solution that react with each other whenever two of them meet. A further example is Broy's FOCUS formalism (Broy and Stølen 2001) with components that realize stream processing functions. Finally, also Petri nets (Reisig 2013) belong to this kind of modeling techniques: consider each transition as an elementary active unit. These and many other similar modeling techniques and programming primitives rise the question for a theory that is a proper basis and abstraction for such systems; in analogy to the computable functions, that are a proper abstraction for any kind of sequential input/output algorithms.

In a series of contributions (among them Wegner 1997 and Wegner and Eberbach 2004), Wegner generalizes Turing machines for this purpose. Cockshott and Michaelson (2007) challenges the correctness of Wegner's arguments.

My point here is not the limitations of capabilities of computers, but a quest for system modeling: Which techniques are adequate to model and analyze systems that proceed in discrete steps and interact with their environment?

7 Independent steps

The concept of *discrete steps* is based on *states*: A step starts at a state and ends at a state. The classical framework of system descriptions assumes *global* states: Each step updates a global state. A single behavior, a run, is then a sequence $s_0 - t_1 - s_1 - t_2 - s_2 \ldots$ of states s_i and steps

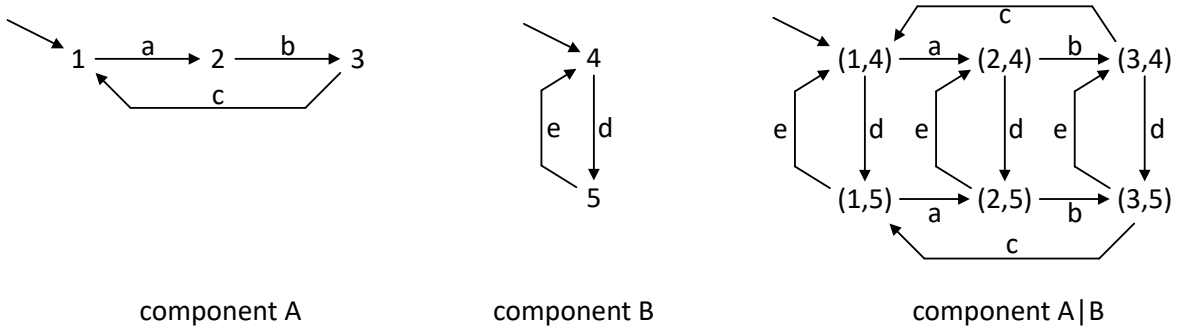

Figure 1: Cartesian product $A \mid B$ of two components A and B

$s_{i-1} - t_i - s_i$ ($i = 1,2,\ldots$). The state space of a system C, composed from components $C_1, \ldots, C_n$, is usually constructed as the Cartesian product $S_1 \times \ldots \times S_n$ of the state spaces S_k of the components C_k. Each step occurrence of a component C_k thus implies many step occurrences of the composed system C: independent steps in different components are *interleaved*, i. e. represented in arbitrary order, thus yielding a non deterministic system model.

Fig. 1 shows a small example: A and B are two independent components with three (resp. two) states. The steps of each component form a cycle. Each of the two components has exactly one infinite behavior (run). The composition $A \mid B$ of A and B is a component with six states. In three of them, two steps start, yielding an infinite number of infinite runs. This perception of "behavior" is intuitively plausible, and is employed in many analysis techniques. In particular, it is the base of model checking composed systems.

However, this perception comes with disadvantages: In the above example, the two steps starting in a state of $A \mid B$ look like alternatives; but in fact, they occur independently. The aspect of alternative refers to an alternative temporal sequence as measured on clocks outside $A \mid B$. Lamport (1978) discusses details of this kind of assumptions on temporal orders of events; to do so he requires clocks with perfect precision. This kind of assumptions can be avoided, taking advantage of the observation that independent steps start and stop in *disjoint local states*. As a consequence, independence of steps is identifiable, and can be covered by representing the two steps without any order. A single run then is no longer a sequence of steps with global states, but a partial order of steps with local states. Order then no longer represents progress of time, but the (causal) "before-after" relation. With this perception, the system $A \mid B$ of Fig. 1 has only one (infinite) run, joining the order of A and B. More illustrating may be the behavior of $A \mid B$ with the additional requirement that B never executed more local steps than A. This results in just one run, as shown in Fig. 2. Petri nets support this proposal with the concept of "distributed runs" (Reisig 2013).

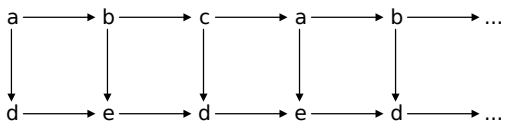

Figure 2: Distributed run of $A \mid B$, with the requirement that B never has executed more local steps than A

Lamport's example of an hour-clock, moving to the next state each full hour, illustrates a further disadvantage of the perception of a single run of a system as a sequence of steps: One would expect that an hour-and-minute-clock is an hour-clock as well. An hour-and-minute-clock, however, executes not one, but 60 steps each hour! Hence,

it is not suitable as an hour clock! To overcome this problem, Lamport (2002) suggests a "stuttering" logic that conceptually equates a single step with "any number of steps".

Both examples show poor consequences of perceiving a single run as a sequence of global steps, whenever systems are composed or refined. It is apparently useful to take independent events, i. e. local causes and local effects, seriously and to distinguish them from nondeterminism. This applies in particular to big systems, because they are usually composed or refined from smaller systems. Küster-Filipe (2000) suggests a specific logic for such systems.

In a more general perspective, here we suggest to take concurrency as a fundamental phenomenon of the real world, to be respected and represented in models. Abramsky (2006) contributes valuable insight into this question. This contrasts the view of concurrency as an implementation issue, assuming "sequential thinking" as the basis of Computer Science, as pleaded in Rajsbaum and Raynal (2019).

8 Limited expressivity of assignment statements

Not only programming languages, but also modelling languages describe steps by help of assignment statements. This is adequate, or at least acceptable, in many cases. But it also leads to less convincing models. An example is the "pebble game" that Dijkstra describes in a video of an ambitious series of videos of Stanford University (Dijkstra 1990): assume an urn, containing finitely many black and white pebbles. A *step* removes two pebbles a and b out of the urn and returns a pebble, c, according to the following rule: c is white in case a and b are colored differently; c is black, otherwise (in case of two white pebbles, one of them is colored black). In a sequence of such steps, all pebbles disappear until only one remains.

Fig. 3 shows Dijkstra's model: a nondeterministic program with arithmetic operations on four integer variables. Fig. 4 models the game as a

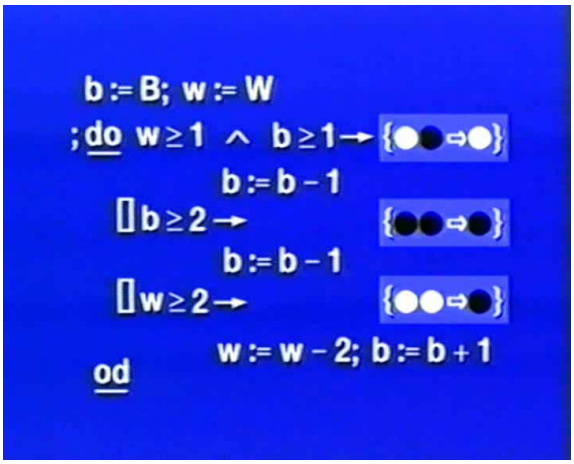

Figure 3: Dijkstra's representation (Dijkstra 1990) of the Pebble game

Petri net: *PEBBLE* is a constant symbol, to be initialized by a (multi)set of for sets of black and white pebbles. Each Transition shows a step, with two pebbles removed and one pebble returned. Arc inscriptions show the color of the involved pebbles. This model represents removal and return of pebbles straightly. It avoids Dijkstra's detour of counting and computing the number of involved pebbles. By help of an invariant, Dijkstra shows that the remaining pebble is white if and only if the initial white pebbles are odd-numbered. A corresponding invariant exists likewise for Petri nets (Reisig 2013).

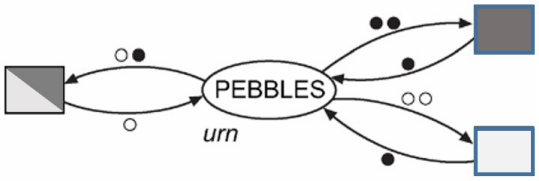

Figure 4: Petri net representation of the pebble game

Variables and assignment statements are also less favorable to model distributed systems such as communication protocols, etc. An example is the TLA model of an asynchronous interface as in Lamport (2002). Ultimately, a scheduler is assumed, regulating access of many components to the variables they share. The components

are distributedly implementable only under far reaching assumptions.

A proper science of informatics will eventually describe system steps not only by help of assignment statements, but also by a variety of other, possibly more abstract concepts. Petri nets are an example: The semantics of a step, i. e. a transition, is given by the updates of the marking of the places in its local vicinity. Broy (1998) suggests a different approach refraining from assignment statements.

9 The metaphor of the living organism

New systems can be constructed by refining and composing given systems. Are there further methods to systematically construct new systems from given ones? The few proposals include the metaphor of a "3D printer", as well as the "living organism" metaphor (Dershowitz and Falkovich 2014). According to this metaphor, a set of "living cells" may create autonomous "creatures" with fundamentally new properties. More generally, this rises the question for the principal limits of such constructs, in analogy to the limits of computability (viz. symbol manipulation) in classical computation.

10 Correctness, and verification

Scientific theories live from models that bring about interesting consequences. A model is beneficial only it yields interesting, non-trivial insights. The purely descriptive character of UML, BPMN, ASM and other modeling techniques without specific analysis techniques considerably limits their usage. A really good model of a system is trustworthy (cf. Sec. 3) and can be analyzed, in particular by help of non-trivial invariants (cf. Sec. 4).

Many properties of systems are reducible to properties of single states and runs. Temporal logic has reached a dominant position to describe such properties, with model checking and abstract interpretation as efficient analysis techniques. The abstract distinction of liveness- and safety properties (Alpern and Schneider 1985) is very useful in this context. Nevertheless, a science of informatics should offer means to represent and to prove much deeper properties too, possibly based on non-trivial invariants (Sec. 4) and a specific notion of information flow (Sec. 5).

A user of a large system requires a modified notion of "correctness": He is usually not interested in the correctness of the entire system (many big systems are – at extreme situations – not correct anyway). His only interest is the system's correct functioning for his specific use case. Additionally, he would love a plausible, intelligible, convincing argument why he can trust the result. Classical verification misses both requirements: a flawed system may be useful in some cases, and the hint at a formally verified property, formulated in terms of temporal logic, does not necessarily support trust in the intended outcome of a single application. First ideas to overcome this include *certifying algorithms* (McConnell et al. 2011) and runtime verification (Bartocci et al. 2018). A comprehensive science of informatics must include this flexible kind of correctness.

11 Time, causality, observation, etc.

For computer controlled real time systems, e. g. airbag control, classical real time models are adequate. Many modeling techniques utilize a naive notion of time, as if actual time was available in any degree of precision and without additional effort. Lamport (1978) wakens this view to some extent, without withdrawing it entirely. Some modeling techniques such as ESTEL, ESTEREL and statecharts employ the hypothesis of "infinitely quick" digital systems, because such systems work much quicker than the systems in their environment, e. g. their human users.

In fact, the notion of temporal "before – after" is frequently specified, where a "cause – effect" relationship was more adequate. The relationship of (discrete) time, causality and observation is fundamental for models in informatics, but not fully understood. A challenging example is a formal model for Stein's *apple sort algorithm* (Stein 2006): apples role down a sloped plank with

increasingly larger holes. Each apple passes the first hole with a diameter greater than the apple's diameter. With n such holes, this algorithm sorts the apples into n size classes.

12 Refinement and composition

Large systems are in general refined from more abstract specifications, or composed from smaller systems. There is a multitude of general methods, principles and formalisms (such as Back and Wright (1998)) to systematically refine a system from a specification. A fundamental principle for logical specifications is refinement-implication: the specification of the refined system implies the specification of the given system. Corresponding methods, principles and formalisms for the composition of logical specifications have been suggested for the language Z (Spivey 1989), Lamport's TLA (Lamport 2002) and Broy's stream-based FOCUS (Broy and Stølen 2001), together with the far-reaching idea of perceiving composition as logical conjunction. On an operational level, Reisig (2019) suggests a very general composition operator that is associative and does not require any assumptions about the inner of the involved components. All these methods, principles, and formalisms address the right questions. But none of them prevails.

13 Computability

For quite a while, the theory of computable functions has been conceived as the fundament of a science of informatics. All attempts to refute the Church/Turing thesis failed. However, this thesis has frequently been stretched beyond recognition. In fact, it just describes the limitations of systematically manipulating symbols sequences. Informatics, and particularly a comprehensive science of informatics, includes more fundamental aspects, to be covered by formal means. A science of informatics must include the interpretation of symbols in the real world. All this has clearly been addressed in Cleland (1993). Further interesting aspects of this topic are discussed in van Leeuwen and Wiedermann (2012), van Leeuwen and Wiedermann (2013), Shepherdson (1995) and Dershowitz (2012).

In a comprehensive science of informatics, the computable functions certainly will play a crucial role – besides some other concepts. This likewise applies to formal logic.

14 Informatics as an engineering discipline

Each typical engineering disciplines such as electrical engineering or chemical process engineering, is based on a science (such as physics and chemistry). Engineering makes scientific insight useful for man's interest. Software, however, is no such science. Software is the result of activities in the framework of software engineering. So, what is the scientific base of software engineering? One may try with "algorithms"; however, "algorithms" is usually perceived in a far too narrow sense. Most adequate would be a comprehensive science of informatics, as a basis for several engineering disciplines, one of which is "software engineering".

Conclusion

This text is intended to solicit interest in the aim of a comprehensive science of informatics. Physics is a paradigm for such a theory. I outlined a series of ideas for formal concepts that span far beyond computer technology and programming, reach far and should nevertheless start out with a nucleus of basics, thus contributing to a science of informatics. This text provides some suggestions as how a theory might be started. There exists already a lot of insight that would belong to this theory. But a comprehensive view remains to be developed.

The envisaged theories would not render so-far principles of informatics obsolete; they rather should be better and mutually related, together with forthcoming engineering concepts of informatics.

References

Abramsky S. (2006) What are the Fundamental Structures of Concurrency? In: Electronic Notes in Theoretical Computer Science 162(1), pp. 37–41

Agha G. (1986) Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press

Alpern B., Schneider F. (1985) Defining liveness. In: Information Processing Letters 21(4), pp. 181–185

Back R., Wright J. (1998) Refinement Calculus: A Systematic Introduction. Springer

Bartocci E., Falcone Y., Francalanza A., Reger G. (2018) Introduction to Runtime Verification. In: Lectures on Runtime Verification: Introductory and Advanced Topics, Lecture Notes in Computer Science. Vol. 10457 Springer, pp. 1–33

Berry G., Boudol G. (1990) The Chemical Abstract Machine. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Association for Computing Machinery, pp. 81–94

Bichler M., Frank U., Avison D., Malaurent J., Fettke P., Hovorka D., Krämer J., Schnurr D., Müller B., Suhl L., Thalheim B. (2016) Theories in Business and Information Systems Engineering. In: Business & Information Systems Engineering 58(4), pp. 291–319

Bjørner D. (2018) Domain Analysis & Description. A Philosophy Basis. Unpublished manuscript, Technical University of Denmark

Bjørner D., Havelund K. (2019) 45 years of Formal Methods – Challenges and Trends. Unpublished manuscript

Booch G., Rumbaugh J., Jacobson I. (2005) Unified Modeling Language User Guide, The 2nd Edition. Addison-Wesley

Brennecke A., Keil-Slawik R. (1996) Position Papers for Dagstuhl Seminar 9635 on History of Software Engineering. Schloss Dagstuhl

Brock D. C. (2006) Understanding Moore's Law—Four Decades of Innovation. Chemical Heritage Foundation

Broy M. (1998) A Logical Basis for Modular Software and Systems Engineering. In: SOFSEM '98: Theory and Practice of Informatics. Lecture Notes in Computer Science. Vol. 1521. Springer, pp. 19–35

Broy M., Stølen K. (2001) Specification and development of interactive systems. Focus on streams, interfaces, and refinement. Springer

Carriero N., Gelernter D., Mattson T., Sherman A. (1994) The Linda Alternative to Message-Passing Systems. In: Parallel Computing 20(4), pp. 633–655

Cleland C. E. (1993) Is the Church-Turing thesis true? In: Minds and Machines 3, pp. 283–312

Cockshott P., Michaelson G. (2007) Are There New Models of Computation? Reply to Wegner and Eberbach. In: The Computer Journal 50(2), pp. 232–247

Dershowitz N. (2012) The Generic Model of Computation. In: Electronic Proceedings in Theoretical Computer Science 88(1), pp. 59–71

Dershowitz N., Falkovich E. (2014) Generic Parallel Algorithms. In: CiE 2014. Lecture Notes in Computer Science. Vol. 8493. Springer, pp. 133–142

Dijkstra E. W. (1990) Reasoning about Programs. University Video Communications, Stanford. The Distinguished Lecture Series, Academic Leaders in Computer Science and Electrical Engineering, vol. III

Furia C. A., Meyer B., Velder S. (2012) Loop Invariants: Analysis, Classification, and Examples. In: ACM Computing Surveys 46(3), Article No. 34

Gries D. (1981) The Science of Programming. Springer

Gurevich Y. (2000) Sequential Abstract-State Machines Capture Sequential Algorithms. In: ACM Transactions on Computational Logic 1(1), pp. 77–111

Hoare C. A. R. (1969) An Axiomatic Basis for Computer Programming. In: Communications of the ACM 12(10), pp. 576–580

Hoare C. A. R. (2010) What can we learn from Edsger W. Dijkstra? In: Edsger W. Dijkstra Memorial Lecture, Austin Texas

Karagiannis D., Mayr H. C., Mylopoulos J. (2016) Domain-Specific Conceptual Modeling: Concepts, Methods and Tools. Springer

Knuth D. E. (1973) The Art of Computer Programming, Volume I. Addison-Wesley

Küster-Filipe J. (2000) Fundamentals of Module Logic for Distributed Object Systems. In: Journal of Functional and Logic Programming 2000(3), Article No. 3

Lamport L. (1978) Time, Clocks, and the Ordering of Events in a Distributed System. In: Communications of the ACM 21(7), pp. 558–565

Lamport L. (2002) Specifying Systems. Addison-Wesley

Livio M. (2009) Is God a Mathematician? Simon & Schuster

McCarthy J. (1962) Towards a Mathematical Science of Computation. In: Proceedings of IFIP Congress 62. North-Holland Publishing Company, pp. 21–28

McConnell R. M., Mehlhorn K., Näher S., Schweitzer P. (2011) Survey: Certifying algorithms. In: Computer Science Review 5(2), pp. 119–161

Milner A. R. J. (2005) Software Science: From Virtual to Reality. In: EATCS Award Lecture. Vol. 87. Bulletin of the EATCS, pp. 12–16

Naur P., Randell B. (1969) Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO

Object Management Group (2011) Business Process Model And Notation https://www.omg.org/spec/BPMN/2.0/. Last Access: 2019-08-30

Rajsbaum S., Raynal M. (2019) Mastering concurrent computing through sequential thinking. In: Communications of the ACM 63(1), pp. 78–87

Reisig W. (2013) Understanding Petri Nets. Springer

Reisig W. (2019) Associative composition of components with double-sided interfaces. In: Acta Informatica 56(3), pp. 229–253

Shepherdson J. C. (1995) Mechanism for computing over arbitrary structures. In: The universal Turing machine: a half-century survey. Springer, pp. 537–555

Spivey J. M. (1989) Introduction to Z and formal specifications. In: Software Engineering Journal 4(1), pp. 40–50

Stein L. A. (2006) Interaction, Computation, and Education. In: Interactive Computation: The New Paradigm. Springer, pp. 463–484

Tel G. (2000) Introduction to Distributed Algorithms, 2nd Edition. Cambridge Univ. Press

Tucker J. V., Zucker J. I. (2000) Computable Functions and Semicomputable Sets on Many-Sorted Algebras In: Handbook of Logic in Computer Science: Volume 5: Logic and Algebraic Methods. Oxford University Press, pp. 397–525

van Leeuwen J., Wiedermann J. (2012) Computation as an unbounded process. In: Theoretical Computer Science 429, pp. 202–212

van Leeuwen J., Wiedermann J. (2013) Rethinking computations. In: What is computation - Proc. 6th AISB Symp on Computing and Philosophy, AISB Convention 2013, pp. 6–10

Vitányi P. M. B. (2005) Time, Space, and Energy in Reversible Computing. In: CF '05: Proceedings of the 2nd conference on Computing frontiers. Association for Computing Machinery, pp. 435–444

Wegner P. (1997) Why Interaction Is More Powerful Than Algorithms. In: Communications of the ACM 40(5), pp. 80–91

Wegner P., Eberbach E. (2004) New Models of Computation. In: The Computer Journal 47(1), pp. 4–9