Prof. Dr. W. Reisig
Humboldt-Universität zu Berlin
D-10099 Berlin, Germany

In Praise of a Theoretical Basis for New Software Paradigms

Wolfgang Reisig
Humboldt-Universität zu Berlin

In the development of computing paradigms and software architectures, the role of theory underwent a substantial decline. We suggest clarifying the potential benefits of a theoretical basis for new software paradigms.

1. How matters developed

During the early days of informatics, i.e. in the 1950ies, the then available technology harmonized with the then abstract view on computing: The von-Neumann hardware architecture was abstractly reflected by Turing machines. Hardware was mainly used for numerical computation, adequately abstracted as computable functions.
Theory supported simplicity *(e.g. context free grammars for compilers*) and structured the area *(e.g. with the O(n) notation of complexity theory).* Teaching of informatics started with much of theory (at least in Germany).

In the late 1960ies and early 1970ies, the “Software Crisis” solicited engineering principles to systematically develop software. Theoretical contributions included theories for the semantics of programming languages and the verification of programs. Though broadly taught in many university courses, both contributions did not catch on in industrial software design.

A number of modelling- and software architecture paradigms attracted attention in the 1970ies and early 1980ies, including Algebraic Specifications, Petri Nets, Process Algebras, Statecharts, and Temporal Logic. Each paradigm came with its own theoretical standards. Their teaching fell apart, with each Computer Science department to select its own preferences. For decades now those paradigms enjoy limited but stable attendance in the software industry, with software tools supporting their use and the - more or less systematic - generation of software from models. Two theory based approaches gained broad attention: Object orientation (based on Abstract Data types and Algebraic Specifications) and Model Checking (based on Temporal Logic).

Middleware and internet software have been the central topics of software architectures in the 1990ies. Theoretical considerations are sparse in both fields, with the exception of arguments on performance and complexity. Teaching concentrated on concrete middleware languages such as Corba, with little emphasis on general principles.

Recent years brought hype for the UML: A collection of graphical standards, borrowing from statecharts, MSC, Petri Nets, etc. The success of UML may hopefully boost interest in systematic, analyzable, theory based modelling techniques (viz. not UML)

Actual buzzwords and challenges for new software paradigms include *model driven software design, service orientation, cloud computing* and *the internet of things.*

Actual text books cope with the above mentioned software paradigms usually in plain English, augmented by informal graphical depictions, essentially boxes and arrows (e.g. UML). Data structures are occasionally represented in XML and behaviour is described as program code. General considerations on properties of representation techniques and representations are usually missing, such as the meaning and logical structure of constructs, the expressive power of specification techniques, proof of relevant properties, substitutability of one construct by a provably equivalent one, etc..

This short survey demonstrates the declining theoretical background of newly emerging software architectures. This decline occurred as conventional theoretical notions do not fit the needs, and new notions did not arise. The educated reader may point to numerous proposals intended to overcome this problem. But over all he might agree with this analysis.

2. What we need

New software- and system paradigms could better be developed, described, and made use of, if they could be presented on a decent theoretical background, providing some useful general insight and principles. This should be conceived a central issue of “modelling” in our field. The construction of such a theory is far from trivial. In the sequel I just discuss three aspects, where this theory would substantially deviate from the classical approach of computable functions.

1. Runs of a computing device are frequently not intended to terminate. This was visible since the very first operating systems in the 1960ies already: An operating system is essentially a “do forever”-loop. Non-terminating runs became more relevant ever since; they are the standard for embedded systems software as well as for “always on” software, e.g. for service oriented architectures. Formal models for reactive systems, such as process algebras and temporal logic, cope with this aspect. It nevertheless deserves much more attention.

2. The modelling of a system frequently must cope with real world items. For example, a model of a lift control system must include properties of physical doors, locks, motors, etc, in combination with software components. The idea to just model such items symbolically is frequently too superficial: Decisive properties of such items must be considered, too. Even very elementary mathematical algorithms can not trivially be represented in a simple, symbolic manner: Already the initial states may be not representable, as their number exceeds the countable number of symbolic representations. System models and algorithms must be carefully separated from aspects of implementation.
3. A system in general involves more than one agent. Interaction of agents is a fundamental issue of any new theory of computation. Many-agent-systems therefore should not be conceived as a generalization of one-agent systems, but as a constituting aspect of systems. As an example, specify the problem solved by a *MUTEX* algorithm, and prove the algorithm is correct.

A number of further requirements may be stated: The intended theory should comply with the basic laws of physics (Turing machines don’t); it should characterize the border line between “the implementable” and “the non-implementable” (which is definitely not the border line between the computable and the non-computable functions). Ideally, it provides decent links to “soft” topics such as the psychology of design principles and interfaces, etc.

In science, sometimes theory is before practice (e.g. in modern physics) and sometimes behind. In software it was always behind (except the early 1950ies and in the case of OO). Now, it is far behind.

A good starting point for such a theory may now be the paradigm of Service-oriented architectures (SOA). As influential software engineers claim: SOA presently is “THE most relevant emerging paradigm of software architectures”, “a substantial change of view as it happens at most once a decade”, “the next fundamental software revolution after OO”, “much more than just another type of software”.

3. Résumé

A comprehensive *theory of software* may deepen our understanding of what we are doing, simplify our products, sharpen our profile in the public, and improve our teaching.