Simplicity Considered Fundamental to Design for Predictability

Wolfgang A. Halang

Fachbereich Elektrotechnik und Informationstechnik
FernUniversität Hagen, Germany

Abstract. Complexity is the core problem of contemporary information technology, as the "artificial complicatedness" of its artefacts is exploding. Intellectually easy and economically feasible predictability can be achieved by selecting simplicity as fundamental design principle. Predictability of system behaviour is identified as the central concept for the design of real-time and embedded systems, since it essentially implies the other requirements timeliness and dependability holding for them. Practically all dynamic and "virtual" features aiming to enhance the average performance of computing systems as well as the traditional categories and optimality criteria are found inadequate and are, thus, considered harmful. In mainstream research on scheduling the gap between academic research and reality has grown so wide that research results are doomed to irrelevance. Instead, useful scheduling research ought to employ utmost simplicity as optimality criterion, and strive to minimise software size and complexity. Computing should embrace other disciplines' notions and technologies of time. Programming and verification methods for safety-related applications are identified on the basis of their simplicity and ergonomic aptitude. It is advocated to utilise the permanent advances in microelectronics to solve long untackled problems and to foster simplicity and predictability by hardware support.

Keywords: Design for predictability, simplicity, dependability, safety, real-time and embedded systems, design concepts.

Complexity is the core problem of contemporary information technology, as the complexity — or better: the *artificial complicatedness* — of its artefacts is exploding. Thus, for instance, the standard document DIN 19245 of the fieldbus system Profibus consists of 750 pages, and a telephone exchange, which burned down in Reutlingen, had an installed software base of 12 millions lines of code. The problem was already addressed in a 1989 speech by Dijkstra, in which he said:

> Computing's core challenge is how not to make a mess of it.
>
> Prevention is better than cure, in particular if the illness is unmastered complexity, for which no cure exists.

> ... we better learn how not to introduce complexity in the first place.
>
> It is only too easy to design resource sharing systems with such intertwined allocation strategies that no amount of applied queueing theory will prevent most unpleasant performance surprises from emerging. The designer that counts performance predictability among his responsibilities tends to come up with designs that need no queueing theory at all.
>
> ... both the final product and the design process (must) reflect a theory that suffices to prevent a combinatorial explosion of complexity from creeping in.
>
> *It is time to unmask the computing community as a Secret Society for the Creation and Preservation of Artificial Complexity.*

These views were complemented by Biedenkopf in a 1994 paper:

> Complexity is a bureaucratic instrument of dominance (power).
>
> The stock of complicated solutions needs to be overcome.
>
> Simple solutions are the most difficult ones: they require high innovation and complete intellectual penetration of issues.
>
> *Progress is the road from the primitive via the complicated to the simple.*

Therefore, the ethics of IT professionals (Endres, 2003) include to take care that programs and systems remain intellectually tractable, to prefer simple structures and solutions for their clarity, to fight complexity, and to resist the many fashions, i.e., in other words, to obey the rule *Keep It Small and Simple!*

Stankovic and Ramamritham (1990) defined predictability by the possibility to show, demonstrate, or prove that requirements are met subject to any assumptions made, e.g., concerning failures and workloads. As all digital computers work fully deterministically, this notion of predictability needs to be formulated more precisely by taking the economical and intellectual *effort* necessary to show the fulfillment of the requirements into consideration. Intellectually easy and economically feasible predictability can then be achieved by using simplicity as fundamental design principle, which also yields dependability.

According to DIN 44 300 (1972 resp. 1985), real-time operation is the operating mode of a computer system in which the programs for the processing of data arriving from the outside are permanently ready, so that their results will be available within predetermined periods of time. The arrival times of the data can be randomly distributed or be already a priori determined depending on different applications. Thus, real-time systems must keep pace with their environments, in which they are *embedded*, resulting in the fundamental requirements *timeliness*, i.e., no tardiness is permitted, *simultaneity*, *predictability*, and *dependability* to hold under worst-case conditions. Real-time systems are (almost) always safety-related, and there is no functional correctness without correct timing.

Preconditions for dependable behaviour are *functional correctness*, *robustness*, also with respect to inappropriate handling, and *permanent readiness.* The latter is implied by the definition of real-time operation and calls for fault-tolerance and *graceful degradation*, i.e., predictable reduction, of performance in cases of error. It is a *misconception* to conclude from random distributions of data arrivals that real-time systems should behave non-deterministically. On the contrary, all computer reactions must the precisely planned and predictable in every detail. This holds, in particular, for situations of conflict and error. Only systems behaving deterministically are safety-licensable. Thus, *predictability of system behaviour is the central concept for the design of real-time systems*, since it essentially implies the other three requirements.

Sources of unpredictable delays and unbounded contention in contemporary computers are the use of synchronisation methods without a notion of time, interrupt inhibition, operating system overhead, direct memory access, caching, pipelining, dynamic storage allocation, virtual storage, garbage collection, multitasking, static priority scheduling, probabilistic arbitration and communication protocols etc., i.e., practically all dynamic and "virtual" features aiming to enhance the *average* performance of non-real-time systems which are, therefore, *considered harmful.* As, in general, program execution and system response times are completely unpredictable and totally unknown, the state-of-the-art in verifying the performance of real-time systems is still *Hope and Pray* (Stankovic, 1990).

Inappropriate categories and optimality criteria widely employed in systems design are probabilistic and statistical terms, fairness in task processing, and minimisation of average reaction time. In contrast to this, the view adequate for real-time systems can be characterised by observation of hard timing constraints and worst cases, prevention of deadlocks, prevention of features taking arbitrarily long to execute, static analysis, and recognition of the constraints imposed by the real, i.e., physical, world.

In mainstream research on scheduling *maximum processor utilisation is a category of the 1940s which refuses to die.* The original raison d'être of scheduling was cost minimisation and, indeed, in the 1940s and 1950s processors were the most expensive components of computer systems. In the meantime, however, cost relations have turned around totally. Thus, the gap between academic research and reality has grown so wide that research results on unrealistic problems with (subliminal) load and resource utilisation models are doomed to irrelevance, in particular for embedded real-time systems, on which people depend. Consequently, it is nonsense to regard this kind of scheduling as a viable research topic because, for instance, it is actually detrimental for system dependability to allocate a single CPU to several control loops. A realistic, holistic assessment must include safety aspects, value of the environment, and reliability and predictability of system behaviour. Then, hardware costs are negligible in first approximation: one hour production stoppage of a medium-size plant costs, e.g., some €50,000 in contrast to some €150 for a processor board, €5 for a powerful TMS320LF240X signal processor, or even only €0.99 for an ST7lite microcontroller. Since, on the other hand, one hour work of a software engineer, which is equivalent to just a single line of tested and documented code, costs €100 or more, software is the area where considerable savings can be achieved. If tradeoffs have to be made, suboptimal processor utilisation is a cheap price to be paid for simplicity, predictability and dependability.

Based on the above, useful scheduling research ought to employ *utmost simplicity as optimality criterion.* New algorithms should support low complexity task execution schedules featuring minimisation of context-switches, inherent deadlock-prevention with implicit resource-access synchronisation, incorporation of inter-task and network communication, methods to determine required processor capacity, and cost minimisation according to the "bottom-line approach", i.e., taking all cost factors into account. The most pressing *economical* question in the optimisation of real-time computing systems with the largest potential for cost reductions and, at the same time, predictability gains is *minimising software size and complexity.* The current interest in minimising power consumption is a stray of hope that scheduling research is turning to feasible problems.

In process control, the duration between sampling and resulting actuation is generally unpredictable. The problem is exercarbated by handling multiple control loops with multi-tasking. Such delays are *not* negligible for reasons of control quality, stability, and robustness. There was no attempt, yet, to solve this problem with the help of suitable computer architectures.

The notion of time is suggested in a natural way by the flow of occurrences in the world surrounding us. As the 4th dimension of our (Euclidian) space of experience, time is already a model defined by *law* and technically represented by Universal Time Co-ordinated (UTC). Time is an absolute measure and a practical tool allowing to easily and predictably plan processes and future events with their mutual interactions requiring no further synchronisation. This is contrasted by the conceptual primitivity of computing, whose central notion algorithm is time-independent, where time is reduced to predecessor-successor relations and is abstracted away even in parallel systems, where no absolute time specifications are possible, where the timing of actions is left implicit in real-time systems, where there are no time-based synchronisation schemes, and where "Time (is even) Considered Irrelevant for Real-Time Systems" (Turski). As a result, the poor state of the "art" is characterised by computers using interval timers and software clocks with low (and in operation decreasing) accuracy, which are much more primitive than wrist watches. Moreover, meeting temporal conditions cannot be guaranteed, timer interrupts may be lost, every interrupt causes overhead, and clock synchronisation in distributed systems is still assumed to be a serious problem, although radio receivers for official date and time signals, as already available for some 100 years and widely used for many purposes, providing the precise and worldwide only legal time UTC could easily and cheaply be provided in each node.

Society increasingly depends on computerised systems for control and automation functions in safety-critical applications. For economical reasons, it is desirable to replace hardwired logic by programmable electronic systems in safety-related automation. Hardware has already reached a far higher degree of dependability than software (Hatton, 1995):

> We are now faced with a society in which the amount of software is doubling about every 18 months in consumer electronic devices, and in which software defect density is more or less unchanged in the last 20 years.

Hence, the software dependability problem needs to be solved by reducing complexity. Software must be valid and correct. It is correct if it fulfills its problem specification. For the validity of specifications there is no more authority of control — except the developers' wishes, or more or less vaguely formulated requests. In principle, automatic verification is possible. Validation, on the other hand, is inherently hard, because it involves the human element to a high extent.

Software is intellectually difficult and complicated to develop. It contains only design errors always being present, and needs correctness proofs, as tests cannot show the absence of errors. Safety licensing of systems whose behaviour is largely program-controlled is still an unsolved problem, whose severity is increased by the legal requirement that verification must be based on object code. The still too big a semantic gap between specifications and the too low a level programming constructs available can be coped with by *the-other-way-around approach*, viz., to select programming and verification methods of utmost simplicity and, hence, highest trustworthiness, and custom-tailor execution platforms for them.

Descartes (1641) pointed out the very nature of verification, which is neither a scientific nor a technical, but a *cognitive process*:

> *Verum est quod valde clare et distincte percipio.*

Verification is also a *social process*, since mathematical proofs rely on consensus between the members of the mathematical community. To verify safety-related computerised systems, this consensus ought to be as wide as possible. Furthermore, verification has a legal quality as well, in particular for embedded systems whose malfunctioning can result in liability suits. Simplicity can be used as the fundamental design principle to fight complexity and to create confidence. Based on simplicity, easy understandability of software verification methods — preferably also for non-experts — is the most important pre-condition to prove software correctness.

Design-integrated verification with the quality of mathematical rigour and oriented at the *comprehension capabilities of non-experts* ought to replace testing to facilitate safety-licensing. It should be characterised by simple, inherently safe programming — better specification, re-use of already licensed application-oriented modules, graphics instead of text, and rigorous — but not necessarily formal — verification methods understandable for non-experts such as judges. The more *safety-critical* a function is, the more *simple* the related software and its verification ought to be. Based on their simplicity, clarity and ergonomic aptitude, programming and verification methods for the higher Safety Integrity Levels according to IEC 61508-1 should be selected as follows:

SIL	**Programming Method**	**Verification Method**
4	Rule base tables	Inspection
3	Function block diagrams with verified libraries	Diverse back translation

Microelectronics can now place some 1 billion transistors on a single chip. According to the classical mainstream approach one could use them to preserve the von Neumann dogma and to create more (artificial) complexity resulting in unsurmountable problems for verification, packaging, pinning etc. It would be better, however, to utilise these hardware capabilities to solve long untackled problems by removing obsolete auxiliary artefacts, replacing virtual features by real ones, providing resource adequacy (Lawson, 1992), enabling "thin" interfaces, providing secure computing environments, allowing for optimum design decisions, and fostering simplicity and predictability. Hardware-support is a feasible means to achieve predictability, as was already shown by the following systems implemented in form of prototypes:

- parallel event processor for a real-time operating system kernel,
- elaborate timing unit with permanent UTC synchronisation,
- peripherals for jitter-free handling of time-tagged I/O data,
- dedicated processors for execution of SIL 3/4 software,
- registerless memory-integrated processor relinquishing caches,
- high-level-language oriented embedded system on a chip,
- DMA without degradation of processor performance,
- fieldbus without data repetitions in case of error,
- non-intrusive testbed for distributed real-time systems, and
- security mechanisms preventing malware.

In 1991, the author said in a keynote speech:

> Future real-time control systems will only be able to meet the demands of society if they will be dependable. Therefore, they must be *simple, behaviourally predictable,* and *safety-licensable.*

Obviously, the last 13 years were wasted to achieve this goal, as academia continues to solve non-problems, and industry follows fashions.