Introduction to the Formal Design of Real-time Systems

by David Gray
Applied Computing
Springer-Verlag London Limited, 1999, XIII p. + 461 p.
ISBN 3-540-76140-3

The target is customary: software engineering, in one of its most complex and often approached subdomains: concurrent and real-time systems. The weapon – felt as hard to learn and cumbersome to handle – is rather infrequent: formal design methods. The conceptual distance between the disciplined, closely controlled and well-established world of mathematics and the effervescent and ever-increasing jungle of real-time application development is huge. No wonder, the low hit rate – so far. This book may possibly change the overall picture.

The volume is divided into **four main Parts** (here "divided" is meant in its purely technical sense; in fact, the book strongly binds those parts together).

The **first Part** has two chapters, introducing and defining the problems associated with the correct design of the systems focused upon. The **first Chapter**, *Scene Set*, brings in the key notions of model, abstraction, atomicity, algebra, transition, concurrency, real-time, etc., like in a well-written play, where each happening is motivated by previous ones; moreover, it "stimulates awareness of the pitfalls of concurrent systems design". The **second Chapter**, *Concurrency and communication*, after identifying the problems, explores several paths of modifying existing sequential programming languages to address them – both in design and implementation; the solution spectrum ranges from simple critical section guards, through to monitors. Though, in its final subsection, a genuine "Review and Rethink", we find the necessary warning: "these programming solutions, while of use as methods by which to navigate, are of marginal use to designers attempting to satisfy a proof obligation".

The **second Part** "introduces the process algebra SCSS in a multipass fashion". The informal overview of the asynchronous algebra *CCS* (Calculus of Communicating Systems), in the **third Chapter**, *Message Passing*, is based on the idea that "with formal models for the sequential bits we need only add a model of message passing to create the design method we seek"; one of the conclusions is that "central to the success of this framework was deciding how the sequential bits should communicate". This overview is followed in the **fourth Chapter** by a formal and comprehensive description of such algebra's synchronous counterpart, expressed in the chapter's heading: *Synchronous Calculus of Communicating Systems*, capturing the issues of synchrony and timing, by means of a concept of global (discrete) time. It is pointed out that the "SCCS design strategy is a constructional one" (the most relevant example: software interrupts). The same framework is used for the specification and the design supposed to meet it (both system behaviours are captured in terms of labelled transition systems). Proving their sameness is handed over to the **third Part**, consisting of the **fifth Chapter**, *Equivalence*. In spite of its succinctness (only 32 pages) this part is very dense: after clearing up the meaning of equality, equivalence, and congruence it explores the role of observation ("two systems are equivalent when no observation can distinguish them"). To check if two agents behave the same if they engage in the same actions, they are compared by bisimulation (each agent simulates the other). The bisimulation relation over expressions is proved to be also an equivalence one.

The **fourth and final Part**, "the one which the students like best", consists of **two Chapters**. The sixth, *Automating SCCS*, brings into focus the concurrency workbench, "a tool to support the automatic verification of finite-state processes". The **last Chapter**, *Proving Things Correct*, launches "a more flexible way of writing specifications", based on modal logic (specifically, Hennessy-Milner Logic), adding to process algebras, formulas over agent properties (describing their behaviours).

At the end, before the substantial *References* and *Index*, two *Appendices*: the first recapitulates "some of the more useful SCCS propositions", the second looks at the "notation used throughout the book".

Hopefully, this condensed passing through the content can sustain the claim that, without being a "philosopher's stone", "formal methods [...] rather than replace traditional methods, they complement them".

Thus, beyond its obvious utility as a course book – the intended audience includes undergraduate and graduate students – the book is a premeditated bridge over the deep – and old – gap between software engineers, considering that rigorous techniques, as the author puts it, "were thought foolish and unnecessary" and those few computer scientists believing that "if you cannot write down a mathematical behaviour of the system you're designing, you don't understand it". Whether the engineers will massively follow the proposed path, is yet to prove. Perhaps other concurrency workbenches, more user-friendly than the exposed one, "which, unfortunately, employs a notation which, constrained by the lowest common denominator of character-oriented interfaces, is rather idiosyncratic" will strengthen the impact of such endeavours. Nevertheless, undoubtedly, in this regard, the book has an almost ecumenical role to play. And it succeeds, due to a most outstanding feature: its fascinating educational eminence. Among several facets able to substantiate such a hazardous assertion, just three convincing aspects:

- the pictures are so many, so diverse, so close to the turn of phrase, so taken on board by the text itself, that they have neither number, nor caption. And, for sure, they don't need it, because they are simply another way of writing: the whole becomes a kind of multimedia on paper.
- the (sometimes, forgotten) art of choosing the best examples – both simple, relevant and "isomorphic" to the particular topic – is breathing and animates the entire book. For instance, the process of purchasing a stamp from a machine is modelled initially on page 15, but afterwards the same example is given many times throughout the book, in a host of reworkings, marking the headway of the very undertaking.
- like in any well-written tutorial, complexity is added stepwise. However, here the steps are so small that you hardly notice them. It resembles a crescendo in a symphony, or – to choose a comparison closer to the subject we talk about – the growth of a silicon monocrystal. An example of this "loud speaking the thoughts": the six successively improved solutions of the mutual-exclusion problem, ending with the Dekker algorithm, are nonetheless followed by two others, proposed as simplifications.

Thus, only the system complexity is perceived, the cognitive one is fading away. In this regard, the generous redundancy is quite helpful (some repetitions are aimed at those not "reading the text linearly"). Therefore, it is no surprise that "all of the students who followed the original modules took to the formalism like ducks to water". Moreover, it can be expected that "the premise that most practising software writers are more craftsmen than they are engineers" will vanish or, at least, become obsolete.

The compelling intelligibility remains unaffected by the few technical "taskbuilding" errors; some examples, to be put right in a next edition: *predicate* instead of *proposition* (logical evaluations, pp. 253 -254), *associative* swapped over with *commutative* (the laws of Abelian monoids, p. 319), minor programming errors (mistaken variable types, p. 67, p. 90).

To summarise, the book seems to be successful due to the author's approach; indeed, "the involvement of nasty things like discrete mathematics" becomes palatable, even nice, because, instead of trying to "digitalise" designers, Gray took the other way round: he "humanised formulas". And it worked.

Boldur Barbat