AUTOMATED SOFTWARE ENGINEERING: SUPPORTING UNDERSTANDING

DRAFT OF 19TH JULY 2008

Michael Jackson

The Open University
jacksonma@acm.org

Each issue of the ASE Journal, on its inside back cover, reminds us of the breadth of its aims and scope. They include "... study of the foundations, techniques, tools and applications of automated software engineering ... techniques for constructing, understanding, adapting , and modelling software artifacts and processes ... specification and design representation schemes, both formal and informal; descriptions and models of the development process; tools and environments to support software development; cognition in software development, including studies of specifiers, designers and implementers, and cognitive properties of representation schemes, programming and programming languages; ... ." This is a broad field indeed, and almost everything is included that could plausibly be claimed as a part of software engineering. But the breadth is justified. Software engineering is indeed a process, to be performed by human beings. Software artifacts include both formal and informal representations, and their cognitive properties are of fundamental importance.

Essentially, software development is concerned with *descriptions*. A *program* describes the behaviour of a computer executing it. A use-case *scenario* describes an exemplary instance of a class of co-operative interaction between a human user and the computer. A *requirement* describes an invariant property that the system must guarantee—for example, the conservation of money in an electronic purse system, or the avoidance of skidding in a car's braking system. A *database schema* describes the structure and invariant properties of an assemblage of persistent data. A *domain description* describes some given property of a part of the problem world—for example, the road layout of an area covered by a satellite navigation system for use by motorists, or the sequence of operations by which a lift car can be made to rise from one floor to the next.

We make and manipulate such descriptions to help us in mastering size and complexity: size, because the development of a realistic system must deal with more concerns and more detail than any human head can hold; complexity, because the problem, its context and its solution all exhibit manifold heterogeneous interactions that are difficult to comprehend. Each description, then, must possess suitable cognitive properties: it must record and convey an intelligible mental model that both writer and reader can easily relate to an explicitly selected aspect of the model's subject matter. A mental model, of course, must be capable of being kept in mind and examined there, without placing excessive demands on the human memory. Different aspects of the same subject will often demand more than one description. Together, the descriptions must enable us to understand the emergent properties implicit in the conjunction of the different aspects of each individual subject, and in the combinations and interactions of the different subjects as they arise in the design and operation of the complete system. The role of our software engineering tools is to help us to construct, understand, analyse, manipulate, transform and present these descriptions in the most effective way possible.

The title 'automated software engineering' is to some extent a misnomer. It suggests an automated process that proceeds independently and, once set in motion, acts without further reference to the source of its original stimulus. But automation in this sense, in the context of software engineering, can be a characteristic only of individual mechanised processes—such as compiling a program text or checking that a model possesses a specified property—that occur as constituent steps in a purposeful software engineering activity. In the earliest days of electronic computers programs were written in pure machine code, typically in octal numbers. Then simple assemblers became available, capable of translating mnemonic alphabetic operation codes into octal, assigning storage locations to local variables, and translating variable references into machine addresses. The innovation was hailed by some optimistic enthusiasts as the arrival of 'automatic programming'. It was, of course, nothing of the kind. It mechanised a troublesome encoding task, relieving the programmer of a modest burden and performing the task speedily and far more reliably. The encoding

task, troublesome though it may have been, was only an unwelcome addition to the real work of programming, not its essence. The work of the programmer, now writing in assembler language rather than machine language, was enlarged and deepened. When the distractions of machine code were set aside it became clear that program structure and design were more significant than details of coding, and addressing those aspects of programming—especially as program size increased as hardware became more powerful and more capacious—became a more central concern for all programmers.

In the same way we recognise that the role of automation in software engineering is to provide support by mechanising some of the development tasks that are potentially troublesome—sometimes even to the point of impossibility—for a human engineer to perform unaided. A model checker searches for counterexamples to a postulated property, like a conscientious programmer desk-checking a program: but unlike the programmer it does so exhaustively, at high speed, and with perfect reliability. Relieved of some of our troublesome burdens, we become free to enlarge and deepen the practice of our discipline, moving closer to its essential core and strengthening our capacity to address its essential concerns. Now that we have a model checker, we can focus on the question: what properties should we check?

The heterodox word ‘description’ seems preferable to the more commonly used word ‘model’. One reason is that it allows us to preserve the word ‘model’ for what Ackoff calls *analogic models*, in which one physical reality models another by analogy. A classic example of an analogic model is a system of hydraulic pipes constituting a model of an electrical circuit: the pipes are analogous to the wires, and the water flow to the flow of electricity. Analogic models abound in software engineering in information problems and in information subproblems of larger systems: a typical database is an analogic model of the reality that furnishes its subject matter. By contrast, an *analytic model* is a formal description of its subject matter, supporting some kind of reasoning or calculation about it.

Another reason for preferring the word ‘description’ is that because of its natural usage in everyday language it forces on us more insistently the question ‘What is described here’? In software development it is easy, for example, to vacillate between viewing a precondition in a Z operation schema on the one hand as an assertion that a certain event in the problem world is possible only in certain circumstances, and on the other as restricting the specification domain of an executable software procedure. Similarly, a database schema can be viewed either as describing its subject matter in the problem world or as describing the reified data structure held within the computer system. If the two were perfectly isomorphic the confusion would be harmless, but they are rarely if ever isomorphic. These confusions can gives rise to many difficulties, including those concerning the interpretation of *null* values in relational databases.

A third reason, perhaps the most important, is that we need the salutary reminder that in the engineering of software-intensive systems the meaning of most—arguably, all—of our descriptions is ultimately grounded in the physical problem world, or in the physical world of the software execution platform. The semantic domain that matters is not merely the formal and abstract semantic domain in terms of which the formal meaning of the description language sentences are defined, but the more messy reality in which our descriptions are to be interpreted in practice. This physical grounding means that our descriptions can not capture perfect mathematical truth about their subject matter, because at the granularity of interest the subject matter conforms to no perfect mathematical truth: our descriptions are only approximations to the reality they describe. It means also that each part of the reality may demand more than one description: strong typing is an excellent invention for programming purposes, but nothing in the physical world is strongly typed in the programming sense. An important part of engineering software-intensive systems is therefore the composition or reconciliation of disparate descriptions of the same parts and aspects of the physical problem world.

The raw materials of descriptions are languages. Some descriptions are made from programming languages; some, such as statements of the system’s general purpose, may be made from natural language; some may be made from a formal specification language, or temporal or modal logic, or Horn clauses, or any useful mathematical notation. Diagrammatic descriptions, expressed as statecharts or action diagrams or class diagrams, message sequence charts or state machine graphs, are made from linguistic raw material mined from the deposits of human graphical invention. A description for which no language is available cannot be written—or cannot be written in a useful form. How can a developer restricted to UML describe the grammar of an input stream to be parsed

by a program? Or a reachability property of a state machine? Free-form annotations are not material for automated software engineering tools.

There is an inviting analogy between the raw materials used for the parts of engineering products, such as motor cars or bridges, and the languages used for the descriptions produced in software development. Building everything from just one material is the mark of a craftsman or artist—a potter, for example, or a stonemason—not of an engineer. Motor car manufacturers know that they must use the right material for each part: the windows must be made from glass, not from steel or rubber; the gear wheels in the differential must be made from steel, not from glass or plastic; a body structure built on a space frame can be made from aluminium or fibreglass or plastic, but a unitary body must be made from sheet steel. In the same way, a software developer must use an appropriate language for each description. As a general rule, these languages should themselves be simple, each being well-adapted to the purposes for which it is used. The notion that the value of a language is increased by elaborating its syntax and semantics to embrace more detail and to combine more properties of the subject matter is surely a mistake. The ability to form complex descriptions is less fruitful than the ability to analyse and understand the interactions among multiple simpler descriptions expressed in simple languages.

The manipulations for which we need tools are an unbounded set. At an early stage, programming was the only recognised constituent activity of software development, so a heavy emphasis on compilation as the fundamental operation to be mechanised is unsurprising. And, indeed, some of the tools we need are essentially compilers: their ultimate purpose is to generate an executable program from a more convenient program representation in a language that may be called ‘higher-level’ or ‘problem-oriented’ or ‘platform-independent’. More than thirty years ago proposals were advanced, and languages designed, for automatically generating programs by combining and transforming their formal specifications [Burstall 79], and program generation from problem statements remains an area of active work today. This kind of work, perhaps, can properly be called ‘automated software engineering’. However, there are many other kinds of manipulation, less ambitious but extremely useful, for which software engineers need supporting tools.

In the development of a realistic software-intensive system there is a large gap between the initially posed problem and its eventual solution. Only a human engineer can identify a relevant part of the world, examine it, and capture its properties in a formal descriptions. Only a human engineer can determine what the sponsor, or customer, or stakeholders of a system require, and capture enough of their needs and desires in formal descriptions. Only a human engineer can discern, or invent, a problem structure that brings the complex requirements of a feature-rich system under some adequate kind of control. Only a human engineer can determine what properties of a machine behaviour demand model-checking, or what properties of a problem world domain the system designer must rely on. The gap between problem and solution must therefore be chiefly bridged by the human activities of describing, reasoning, designing, communicating, analysing, and inventing, and software tools are needed to support those human activities and, by doing so, to amplify our human powers. The *sine qua non* of those activities is human understanding. In large measure, then, enabling and supporting human understanding must be the goal of the software engineering tools.

Supporting human understanding means, above all, reducing perceived complexity. In software development, complexity is the mother of error. As C A R Hoare famously said in his Turing Award lecture [Hoare81]: “...there are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.” Here we are talking about complexity in the non-formal sense of a particular kind of obstacle to human understanding. It is intellectual complexity, the complexity that manifests itself as difficulty in writing and understanding software and its network of associated and contributory descriptions—in developing a system and achieving confidence in its dependability. A central role for automated software engineering is to help human software engineers to master that kind of complexity: to ensure that we do not introduce gratuitous complexity where none need exist, and to reduce the residual inescapable complexities of our subject matter to their most understandable forms. Essentially, that means helping us to form a coherent mental model of the subject matter in hand, small and simple enough to be held in the mind and powerful enough to support whatever checking, analysis or reasoning the work demands.

A well-known, but still cogent, illustration of reducing perceived complexity is the widespread acceptance of structured programming in the late 1960s. In the earliest years of computer programming, and for many years afterwards, programs were usually represented by flowcharts. A flowchart is a directed graph, paths along the arcs denoting possible execution paths. The nodes of the graph correspond to executable operations, including test operations: for a conditional branch the corresponding node has more than one outgoing arc. The possible traversals of the flowchart, starting from the fixed initial node, represent the possible executions of the program. As the earliest programmers quickly discovered, it is discouragingly difficult to grasp all possible program executions from a flowchart. Even small programs can be complex and hard to understand, because a flowchart offers no help in understanding how the values of the variables evolve during program execution. Any abstraction or structuring of the execution flow must be devised, and carefully held in mind, by the writer or reader of the flowchart.

In his famous letter [Dijkstra68] to the editor of CACM, Dijkstra explained that "we can interpret the value of a variable only with respect to the progress of the process" and that flowcharts cannot provide useful "coordinates in which to describe the progress of the process." In structured programming, execution is described as a nested set of sequence, conditional and repetition clauses, in the form that is now universally familiar. Progress of the process can be neatly characterised by a stack of textual indices and repetition counts, showing "where execution has reached in the text" and "how many times has each enclosing repetition been executed". These "coordinates" provide a frame of reference in which the evolving values of the program variables can be far more readily grasped, and arguments about program correctness can be far more easily formulated and checked.

Another way in which we sometimes place our capacity for understanding under unnecessary strain is by the production and use of descriptions in a *fragmented*, rather than a *whole*, form. Human beings are ill-equipped to understand a whole description presented in fragments if a relationship among the fragments is an important aspect of what is to be understood. This is why books and papers in which state machines are discussed present the description of the machine in a diagrammatic form rather than in the typographically less demanding form of a transition table. It explains why noted formalists in computer science who usually disdain diagrams nonetheless present their structured programs in texts indented to show the levels of the structure. It is also why the London Underground takes care to present passengers with the famous map rather than an equivalent tabulation of lines and adjacent station pairs on those lines. The map is more useful because in considering a journey we must understand the reachability and path properties of the system, not only the individual elements of the adjacency relations between stations by which they are induced. We therefore need a description that presents those properties directly, and allows us to understand them easily in the sense of forming a reliable and useful mental model for the purpose in hand. To extract the same information from the table alone would be a toilsome, memory-intensive, and highly error-prone process.

The London Underground has recognised that fragmented descriptions are troublesome and error prone, and takes care to present us with the whole description. In software engineering we have not always been so perceptive, as two common, closely related, examples show. The first example is event-driven programming. The usual pattern for event-driven program design is to conceive the program as the recipient—perhaps from keyboard, mouse or interrupt—of a collection of externally initiated events. The meaning of each event depends, of course, partly on its class: a left-click means one thing, a right-click means another, and a keystroke means something else again. The event class can be determined for each event independently, from its source in the environment. But the meaning of each event will often depend also on the global context: a left-click at one point in the whole event sequence means something different from a left-click at another point. The usual form of event-handler design therefore sets and interrogates global state variables to handle the events correctly. There is ample opportunity here for error: the event handler programs form a collection of fragments, but the whole process to which they are intended to be equivalent may never be identified and is certainly never clearly presented.

A second example is the treatment of use-cases in some developments. The operations and events for the US Army device called a 'plugger' (Precision Lightweight GPS Receiver) seem to have been originally designed in a fragmented fashion. The device is used to direct missiles against enemy targets. It has explicit user operations to *initialise*, *set target* and *fire*, and a warning light indicating

that the battery is low and must be changed. In a famous incident [Loeb02] the user set the target, responded to the warning light by changing the battery, and fired. The implementation of *change battery* included the code for *initialise*, for which the device's response was to set its sole location register to the location of the device itself. The missile was therefore directed at the plugger, and killed the user and three of his comrades. Evidently, the sequence of events that occurred in this case had never been considered by the designers of the device. Instead of considering the whole process—the complete history of events in the lifetime of the device—they were merely considering each event in fragmented isolation.

The cure for this kind of fragmentation-driven error is to recognise that fragmentation is necessitated only by the exigencies of an implementation environment, and can be avoided by a suitable treatment of the locus of initiative of each event. A program designed to handle the whole sequential stream of events, or a system designed to respond to the whole sequential succession of use-case instances, would not suffer from the fragmentation difficulty. The programming environment, however, does not allow the program to be written in this whole way but then executed piecemeal as each event or use-case instance occurs. A suitable software engineering tool could allow the program to be designed whole and then statically transformed into the desired fragmented event handler or use-case handler. The equivalence between the collection of fragments and the whole process can be viewed abstractly—for example in a process algebra—or in the most concrete programming terms, as it is in the program inversion [Jackson75] technique (for which a transformation tool was provided in the unpromising environment of COBOL programming of thirty years ago). For determined adherents of fragmentation, another useful tool would reassemble their fragmentary designs, and make directly visible the whole that emerges from them.

Fragmentation is only one obstacle to understanding, and there are many others. For example, developments that are faithfully wedded to object-oriented programming both as their implementation and as their guide to problem analysis and solution may suffer from a lack of abstraction in the large. Global properties are notoriously hard to discern in object-oriented designs, and tools that can support and compose abstractions at many levels are of great value. The equivalent of program slicing tools designed to display projections of more general composite descriptions would support understanding and analysis in many contexts. Tools that manage, manipulate and transform descriptions in this kind of way form an important class of automated—or mechanised—software engineering tools. Their purpose is to help us to understand. They generate no code, but are explicitly directed to extending and strengthening our intellectual reach.

REFERENCES

[Burstall 79] Rod M Burstall and Joseph A Goguen; *The Semantics of CLEAR, A Specification Language*; in Proceedings of the Abstract Software Specifications Winter School, Copenhagen 1979; Springer LNCS 86, pages 292-332, 1979.

[Dijkstra68] E W Dijkstra; *A Case against the GO TO Statement*; EWD215, published as a letter (*Go To Statement Considered Harmful*) to the editor of Communications of the ACM Volume 11 Number 3 pages 147-148, March 1968.

[Hoare81] C A R Hoare; *The Emperor's Old Clothes* (Turing Award Lecture); CACM 24,2, February 1981; reprinted in C A R Hoare and C B Jones, Essays in Computing Science, Prentice-Hall 1989.

[Jackson75] M A Jackson; *Principles of Program Design*; Academic Press, 1975.

[Loeb02] V Loeb; 'Friendly Fire' Deaths Traced To Dead Battery: Taliban Targeted, but US Forces Killed; Washington Post, page 21, 24 March 2002.

Note: A later version of this paper was published in Automated Software Engineering November 2008. That publication is available at www.springerlink.com.