On Constructing Large Computerized Systems (a position paper)

Jean-Raymond Abrial

ETHZ Zurich, Switzerland jabrial@inf.ethz.ch

The subject mentioned in the title of this short article does not seem, at first glance, to be a genuine research subject. Although there are, from time to time, some famous breakdowns of large computerized systems (as, for instance, recently at SBB in Zurich), it seems nevertheless that these systems are working nowadays in a satisfactory fashion. As a consequence, their construction process must have been mastered otherwise such disasters would have occurred more frequently. This was clearly the case at the beginning of last century with an emerging technology such as avionics. There were lots of crashes due to the fact that people did not know how to construct good airplanes. The main reason for this was that they did not understood yet the theory of flight mechanics, which was in its infancy.

In our case, however, the situation is a bit different from that of early avionics in that there is no clear theory yet related to large computerized systems. The most obvious indication supporting this fact is that, most of the time, experts cannot clearly explain why such systems indeed work correctly. When a serious breakdown occurs, the corresponding superficial reason is normally found after some time (not always however), and it is usually repaired in a very ad-hoc fashion. But people are never sure that another breakdown will not occur some time later, precisely because they do not know the more profound reason for that earlier breakdown. It is my belief that such a state of the art is not satisfactory.

From almost the beginning of Informatics, the main tools used to develop computer systems were a High Level Programming Language and a corresponding Compiler. There has been many of them (far too many in fact) always proposing new features whose pretensions are to eventually solve the problem of constructing better programs than the previous generation of High Level Programming Languages and Compilers did. One can even regularly see in the literature the term “next generation of programming languages” being used. Unfortunately, it does not seem to solve the problem since, over the years, the famous software crisis is still with us.

An interesting research is then to investigate whether there could exist some other intellectual means and tools that could be used instead of High Level Programming Languages and Compilers. It does not mean of course that we believe that we can replace final computer programs by something else. But, we would like to investigate whether we could invent some other ways to obtain such final programs.

When High Level Programming Languages were invented in the sixties (i.e. Algol), the idea was to make abstract programming patterns such as conditionals, loops, procedure calls, and the like be first class citizens in the programming methodology. In the more restricted realm of Assembly Code, such programming patterns did not exist explicitly but were all implemented by means of a single feature, namely the conditional goto instruction. The High Level Programming Language was then an abstraction of the more concrete Assembly Code. And the Compiler was the tool that allows us to move from this abstraction to a concrete implementation.

An interesting statistics that can be obtained from high level source programs is the ratio of the number of lines of code devoted to pure algorithms over the total number of lines of code excluding comments and the like. Of course, the figures differ from one application to the other but it is usually far less than 1/2. This means that such abstract relational features as components, classes, methods, inheritance, visibility, assertions and the like describing

the objects, their properties, and their relationships are becoming far more important than pure algorithmic features such as conditionals, loops, and more generally computations. It is my belief that such abstract relational features are not well handled in a High Level Programming Languages, whereas pure algorithmic calculations are in my opinion still well handled in such languages.

In other engineering disciplines (i.e. mechanical construction) people do not hesitate to use languages when they are clearly needed and other means when they are not needed. For instance, they use the language Mathematica to define the formal computations related to their usage of the Calculus. But they do not use languages to describe the complex relationships between the components of their future system, their properties, their links, etc. In fact they store the various components of their product in one form or another and express the relationships that hold between them. We can consider that they thus build a Database of their future system. The engineering process is supported by the contents of this Database, its modification, and the tools that are disposed around it.

In our discipline, there is a frequent confusion between the two terms "assertion" and specification", even if both of them are written using a mathematical notation. An assertion, is a *local predicate* that must always be true at some point in the *execution of a program*: it can be either checked dynamically while the program is running or better statically proved. But in no ways can such assertions represent the specifications of a large computerized system. For the simple reason that the specifications of a large computerized system essentially consist in the definition of a number of *global properties* by which it will be possible to state that the final system comprising software and external equipment (including users) works in a correct fashion. Clearly, when writing such properties the software part of the computerized system does not exist yet and even sometimes also the external equipment. In fact, such global properties are not associated with specific pieces of code in the final software, they are rather supposed to be globally maintained by the software in question together with its environment, which, most of the time, are both supposed to run for ever. Moreover, the specifications, as just described, are the point of departure of the *design* which has also to be defined first globally to end up eventually, *by architectural decomposition*, in some more local properties: the assertions then appear to correspond to the *final stage* of a long design process. It is quite clear that High Level Programming Languages (event “modern” ones) are not at all suited to be the place for writing such specifications.

What is wrong is to have the semantics of the High Level Programming Language being the medium defining the properties of the offered features. This is far more easily handled and modifiable as the invariant properties of a *System Construction Database*. It might still be useful to have some pretty printing of the contents of the Database. This would resemble a high level program but would be produced as an additional output of the construction process rather than as its input. In first approximation, the contents of the System Construction Data Base is made of the *various components* of the system in construction together with their *relationships*. These components are surrounded by a number of *tools* that can be used to *develop* them. The System Construction Database should not be confused however with what High Level Programming Language technology calls a “library”. In fact, it is far more general as explained below. The System Construction Database approach offers quite a number of advantages over High Level Programming Languages. Here are a few of them:

1. The System Construction Database can be used not only to store future software components but also, more importantly, their various *abstract, and later refined, mathematical models*. And here the tools that replace the compiler and even the computer executing the final program are a *proof obligation generator* and a *prover*. Specification, and design, and corresponding tools, are put together with implementation and corresponding tools. In this respect, the System Construction Database contains the on-going design history of the software construction. It is important to note that the specification of a large system

is not a monolithic text but rather a succession of more and more precise mathematical models taking account gradually of the requirements of the future system. High Level Programming Languages are not at all appropriate to handle this task: they suffer from their initial purpose, namely that of instructing a computer on the way to perform its computations. Specifications and design have nothing to do with instructing a computer, they rather record the thoughts and reasoning of the engineers.

2. The System Construction Database approach will also induce a rather more appropriate way of elaborating the final product than that given by the usage of a High Level Programming Languages and Compilers. Unless it is very small, you shall never write a program and subsequently submit it to the compiler. This sequential approach to construction will be replaced by a more reactive approach, which corresponds to the way engineers work. You rather interact with the Database by entering modeling elements, their properties, and their relationships. Such an interaction is permanently supported in the background by tools working in a *differential fashion* without being explicitly even invoked by the user.

3. The System Construction Database approach will also allow us to store and update components which can be quite different in nature from computer programs, namely models of pieces of equipments which might interact with the future software components. Such models will be able to be refined as other future software models are. This will allow us to construct embedded systems by specifying and designing their software parts in strong relationships with some modeling of their environments.

4. Besides the formal tools we have already mentioned above in 1 (proof obligation generators and provers), the System Construction Database may contain other tools as well, being able to be applied to the various models, namely model checkers, informal modelling (UML) to corresponding formal modelling tools, model animators, abstract interpretation tools, even testing tools, etc. The reason for incorporating such tools is that clearly there is *no universal panacea*. The engineers need to have a large palette of possibilities at their disposal in order to construct their computerized systems in the most effective way. The System Construction Database offers the possibility to have all such tools working in an integrated fashion on all the models that are recorded in the Database.

5. Besides the components and their mathematical models (be they future software or environment components), it will be possible to also store in the System Construction Database a document related to the *requirements* of the future system. Such a document will take the form of natural language fragments intermixed with slightly more structured texts containing the concise and precise requirements of the system in construction. A useful analogy is that of a book of mathematics where definitions and theorems are labeled, numbered, and written using a different font from that used in the rest of the text corresponding to explanations and proofs: this will make the definitions and theorems immediately separable from the rest of the text. By structuring in this way the requirement document, the traceability of the requirements will be handled in an integrated fashion within the System Construction Database. This will be done by connecting each structured requirement to some parts of the abstract models dealing with that requirement. This traceability can then be pursued during the design phase and the final software and environment construction. It must be noted here that such requirement documents are usually very poor: either inexistent or far too verbose. As a consequence, the designers have often lots of difficulties in extracting from them the precise requirements. Experience shows that the famous, and said to be inevitable, syndrome of "specification changes during construction", appears to magically disappear when such a special attention is payed initially to writing and structuring the requirement document. Every large project must have an important initial phase devoted to this task: the System Construction Database will then be the natural repository for such requirement documents.

6. The System Construction Database could be spread over several sites so that the communication between people working in different places on the same project could be handled

far more comfortably than with high level programs sent over the network. But of course, the contents of parts of a Database could also be copied under the most appropriate form (XML), which will be far more convenient than, again, sending source files.

In conclusion, we question the present usage of High Level Programming Languages for constructing large computerized systems. We propose instead to partially replace it by defining and using a System Construction Database which will be far more appropriate as an engineering medium than the actual programming languages. As a matter of fact, this proposal is not really new: Eclipse is, among others, a proposal that has been made for a number of years and that goes clearly in that direction.