School of Computing Science,
University of Newcastle upon Tyne

Extending the Horizons of DSE (GC6)

Tony Hoare, Cliff Jones and Brian Randell

Technical Report Series

CS-TR-853

July 2004

Copyright©2004 University of Newcastle upon Tyne
Published by the University of Newcastle upon Tyne,
School of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK.

Extending the Horizons of DSE (GC6)

Tony Hoare
Microsoft Research, Cambridge

Cliff Jones and Brian Randell
School of Computing Science, University of Newcastle upon Tyne

5 February 2004

The aim of this short note is to complement the technical discussion in the documents "Dependable systems evolution: A grand challenge for computer science" and the subsequent "DSE-net: A Network for the Grand Challenge on Dependable Systems Evolution". Its preparation was prompted by our discussions at and following the DSE workshop held last November at Queen Mary College. We suggest how the proposed research could be extended so as to fit with, and both take advantage of and contribute to, ideas on system structuring from the software architecture and other dependability-related communities. While the full set of objectives and research methods of these communities is wider than those of the DSE project, we recommend continued close collaboration with these communities in pursuit of the objectives of DSE.

To emphasise the obvious, the basic idea underlying all techniques aimed at achieving high dependability, is that of *consistency-checking of useful redundancy*. A verifying compiler (if it existed) would aim to check the consistency of a program with its specification once and for all, as early as possible, giving grounds for confidence that error will always be avoided at run time. In contrast, the dependability technique termed fault tolerance delays the checking until just before it is needed, and shows how to reduce the effects of any errors that are so detected to a tolerable level of inconvenience, or possibly even to mask the effects entirely. This technique is used in connection with various types of fault, e.g. hardware, software or operator, but the error checking involved will have been derived from the specification. Our case for collaboration between the verification and fault tolerance communities in particular is made on the basis that the same kinds of redundancy are exploited, no matter when the consistency is checked.

Ideally one has a specification that is fully complete and accurate. In practice, both programs and specifications can harbour faults – but given their different characteristics there is a good chance that these faults will not have identical consequences, so that one can gain considerable confidence from evidence of their consistency, something that will not be the case if the program is synthesised completely automatically from a specification.

Program specifications, especially partial specifications, are often given in the form of assertions; assertions are a form of redundancy whose consistency with the program might be checked before running or during execution of a program. Similarly verifying that code is free of, for example, buffer overflow errors, involves consistency checking, though in this case of the code against application-independent “assertions”. In general, checking the redundancy before execution is an attempt to make sure that whole classes of erroneous behaviour have been avoided whereas checking assertions at run-time might detect errors arising from residual faults early and limit their propagation (and perhaps even facilitate some cleaning up). There are classes of assertions that can be checked by a compiler – these are often thought of as (extended) types. In general, a compiler cannot check arbitrary assertions but it is possible to statically check “proof carrying code”.

The basic idea of consistency checking of useful redundancy in fact underlies *all* forms of program validation and verification, from formal proof to debugging. Code inspections are a form of consistency checking using human redundancy. Perhaps the most extreme example of the use of redundancy is to be found in fault-tolerant systems that are based on the use of “design diversity”, e.g. in systems incorporating multiple separately designed modules with common specifications, running on processors from different manufacturers, and having their results continuously compared. Though there are studies that draw attention to the limits of diversity achieved in supposedly independent designs, significant dependability improvements can be achieved, and in safety-critical systems the use of design diversity is relatively common. (The idea of consistency checking of useful redundancy is equally relevant in hardware, in bureaucracies, and in socio-technical systems involving computers and human beings. However here we will, like the DSE proposal, confine ourselves to software.)

Equally fundamental and closely-related, of course, is the use of system (in particular program) structuring techniques. This is often motivated by considerations of complexity reduction (i.e. understandability), and of code re-use. However structuring – as long as it is retained in the operational system – also provides an important means of error detection and of limiting error propagation. Perhaps the simplest and most powerful form of structuring is that of the use of interpretation – it is for example relatively easy to ensure that the interpreter of a language cannot be harmed by any errors in any programs that it interprets. (Compilation is in essence an act of structure destruction – one gains run-time efficiency, and early detection of many kinds of program fault, but can lose a degree of both flexibility, and of error containment.) The use of middleware, and workflow languages, are other examples of approaches to system construction that involve the retention of dependability-enhancing structures at run time – as indeed is much work on software architecture.

With this preamble, we would like to suggest that the research, and the tool-building, to be undertaken in DSE should, in addition to concentrating on the problems of producing verifiably fault-free individual programs, include significant effort aimed at employing and exploiting high-level redundancy and structuring facilities, not just in the design and verification of complex software systems, but also during their deployment, and their adaptation in response to changing requirements. The rest of this short note aims at explaining and justifying this suggestion.

Exception Handling

Exception handling constructs, both in conventional programming languages, and – equally importantly – at higher system levels (e.g. in workflow, with such constructs as the “compensation” scheme proposed for BPEL4WS, and in configuration languages), are a form of retained structuring that enables one to distinguish the provisions for various relatively unlikely situations. Aside from the benefits – with regard to the design and validation task – of this form of “separation of concerns”, the use of such constructs for handling various types of errors (in particular invalid input data and mistaken operator commands), can aid the provision of coherent methods of error recovery, and the production of systems which can “degrade gracefully” in situations in which the intended full functionality cannot for some reason be achieved. (For brevity, a simplified example could be that of a banking system whose main aim is to service all transactions on all accounts correctly, but which has a sequence of possible fallback behaviours, e.g., first to one in which some accounts’ transactions are not processed, and second to one in which some accounts become totally inaccessible, before failing completely if this is unavoidable.)

The idea of exception handling is an old one, though the development of forms suitable for use in the presence of complex concurrency is relatively new. In such situations a coherent error recovery strategy is essential, given the possibility of multiple simultaneous errors, as well as of further errors occurring while error recovery is still in progress, and the realities of concurrent activity in the world outside the computer system. The provision of formally-defined constructs, and of good tool support, for sophisticated exception handling is therefore crucial.

Software Architecture

Much work in the software architecture research community is presently concentrated on “design patterns”, and in particular techniques for constructing systems out of components and stylized connectors. This form of structuring facilitates not just the system design and evolution, but also run-time error detection and confinement. Generic connectors can be designed to provide various forms of application-independent fault tolerance (including forms that involve the use of design diversity). And (application-specific) connectors can be used to effect so-called “wrapping” of components, especially pre-existing components, in order to try to cope with mismatches between components. Again this whole area is one in which rigour and tool support are both very important.

Levels of Abstraction

Multi-level architectures involve the use of multiple representations of a system, at successively lower levels of abstraction. Low levels concentrate on computer-related issues - higher levels can be couched in terms of application-level concepts, indeed can be expressed in application-oriented languages. This can make it much easier for application domain specialists to contribute significantly to the task of determining whether a system will meet its requirements, indeed to the whole validation and verification activity, and to do so from the early design stages of a project.

Ideally, such levels of abstraction are employed not just at design time, but instead are retained during operation as well, something that increases in hardware performance is making ever more practicable. (Amongst the technologies that are used to provide such levels of abstraction are interpreters, middleware, and reflection.)

Multi-level architectures aid system adaptation, and the incorporation of pre-existing (sub)systems. However, they also enable the use of powerful forms of consistency checking *at* each level, *and* between levels – checking that we would argue can and should be performable at run time as well as prior to run time, in as far as this is possible.

Summary and Conclusions

A principal aim of this document has been to provide an argument for incorporating exploitation of various different redundancy and structure-preserving approaches into the planned programme of work in DSE-net on formally-based software design and verification techniques. This will involve introducing and checking redundancy at various levels of granularity and abstraction, particularly at interfaces between components of a structure. Such additional emphasis on redundancy and structure will, we believe, increase the effectiveness of DSE's verification-oriented approach, its domain of applicability, and its practicability.

The verification research described in the original DSE proposal aims to do as many checks as possible before the program is put into service, some perhaps even at design time. Potential benefits of this strategy include the reduced cost of detecting errors at the earliest possible occasion, and the extra efficiency resulting from a compilation approach that removes the redundancy and the structure from the executable code. In contrast, fault tolerance research explores the benefits of retaining redundancy and structure for checking and exploitation while running a program. The potential benefits include the ability to provided continued service – albeit possibly of a degraded character – despite the occurrence of hardware or residual software errors, in-service software upgrades, and the extra confidence obtained by making checks at the very last minute before they are needed. In fact the two approaches are complementary, and can and should be usefully combined.

In current software engineering approaches, it is often impossible to decide which forms of structure and redundancy will be retained at run time. A framework of tools for dependability, of the type envisaged in the original proposal, but designed to support and exploit rich redundancy and structural information, would both facilitate software evolution and allow these decisions to be delayed, and perhaps even changed during such evolution.

As with the original DSE proposal, we assume that a major feature of the project would be experimental work including examination of past development projects and of existing large codes of current interest. An equally important feature is that the results of this experimental research, and of the underpinning theoretical research, will be incorporated in tools that are of practical use to software engineers.

One final remark: both the original DSE proposal and this document concentrate solely on solutions that can be found by the known hard scientific methods of modelling and deduction, ones that can be wholly represented and solved within the silicon package of a computer. However, unless we maintain links with the research communities concerned with overall system dependability, formalists will ignore all the equally important, frequent, and more intractable problems of the interface between computers and the real world, and particularly human beings. These problems need tackling by a far wider range of research methodologies, including historical investigations, sociology, HCI, management science, etc., which all have a contribution to make to dependability in the more important senses of that word. Any formal project needs to keep contact with the real world, if only to avoid missing opportunities to extend the technology of formal research towards tackling a wider range of real problems.