Issues in Software Safety (for Dependable Computing)

Kohei Honda
Queen Mary, London

February 2, 2004

1 Preamble

This short note presents one possible issue regarding software safety and discusses how we may tackle it, written in the context of Dependable Computing Project [5].

2 Background

In 1996, Necula and Lee published a paper on what they christened proof-carrying code [3]. This initial publication on the topic is a good read with lucid presentation of a technical inquiry which starts from a remarkable conceptual framework to its thorough-going technical implementation, with a well-thought discussion on the context of their technology. When I first read this paper, I was reminded of a talk in the Japan-UK workshop in early 90s on what Burstall and McKinna called "deliverables". There is a similarity, in that both offer a framework in which the responsibility of providing a correctness proof lies in the code producers' part, and a type theory is used for handling a correctness proof (though in quite different ways). There are also major differences: among others, a program considered in Necula and Lee's framework is executable code, rather than a term in the constructive logic, as in the case of Burstall-McKinna's framework. Hence its use of Hoare logic for specifying safety. One of the key technical contributions by Necula and Lee is to find a concise way to represent and verify proofs extracted from annotated code using a type theory. Together with concurrent activities by Gregory Morrisett and his group on typed assembly code, their work marked a starting point of the ongoing wide-spread interests in the guarantee of the safety of software at the level of binaries.

Necula and Lee have chosen, in [3], an interesting setting to present their technical ideas, the so-called extensible operating systems. An operating system is *extensible* when one can incorporate a new module, hence new functionality, incrementally, without the need of recompilation (such a module may arrive as, say, mobile code). While the same framework can be found in languages such as Java, the safety issue arises in the most vivid form in an operating system, since a new module is supposed to run in the protected mode alongside other kernel modules. Thus the lack of safety can literally be disastrous (for example

assume a new module wrongly writes on that part of memory which is being used by a process scheduler). Thus the correctness concern is crucial in this setting: and one of the most telling arguments in the paper by Necula and Lee is that their framework does not incur any run-time overhead while ensuring basic code safety, which is important in the setting the authors have chosen.

The aspect I mentioned above was widely discussed and has been followed up by many researchers, so no further words are necessary. Here I wish to discuss another aspect of their chosen setting, the extensible operating systems, which may touch the very nature of "software safety". An operating system is a piece of software which, as any sizable software nowadays would, consists of multiple pieces of modules which cooperate with each other. For a new piece of code to join them and to function properly, it may as well rely on the functionalities of the existing modules (consider for example adding a new security manager into the kernel for the purpose of an updated standard of digital right management: it should cooperate with existing security managers, a memory manager, a file manager, protocol stacks, etc.). But how can we specify and prove correctness in such a setting? What does it mean for a piece of software to run correctly when its core functionality depends on those of existing pieces of software?

The answer is simple, at least conceptually: the correctness of a piece of software in such a setting can be specified and verified only under the assumption of the correctness of existing pieces of software. And this aspect is not limited to operating systems: in fact, almost **all** kinds of software nowadays explicitly or implicitly rely on the functionalities of other software. A Java program relies on its APIs, these APIs are implemented using native code and OS libraries, an OS library would use kernel calls, and even a kernel relies on, for example, the proper working of the network for carrying out its essential functionalities (how can a TCP stack work properly if the TCP itself, which involves another end of communication as well as, in effect, all routers in-between, does not?).

Thus the issue of software dependent on other software is an omnipresent one: methodologies for the safety and dependability of modern-day software should address this point, since if not we cannot ensure the dependability of the whole software. But can we indeed provide such a framework? Fortunately, the past studies in theories of computing have already provided two basic, and closely related, foundations for answering this problem: compositional program logics and types.

3 Program Logics and Types

A program logic is *compositional* when an assertion for a big program in that logic can be derived from those of its sub programs. For this purpose, an assertion needs to associate a program with information about how it can be composed with other asserted programs. For example, if we have an asserted program $\{A\}P\{B\}$ in Hoare's logic (which is by the way the original compositional logic), then A controls what can be composed to P as its predecessor; and B controls what can be composed to P as its successor. So if we have an asserted program $\{B'\}Q\{C\}$ such that B entails B', then $P;Q$ guarantees $\{A\}(P;Q)\{C\}$, that is it always gives us a state specified by C starting from any state satisfying A (possibly under some assumption such as termination); but if B does not entail B', then we cannot guarantee this property

for a composite program. This can be extended to other constructs, including procedures and concurrency, always with essentially the same framework: an assertion specifies how a program can be composed with other programs relying on their certain properties, as well as a property guaranteed by the composite program when a proper composition has been carried out. Thus the core idea of compositional program logics, which is also inherited in a comprehensive specification/refinement framework such as Z, is the very framework which we would need for capturing the correctness of dependent software.

Interestingly, the above reading of Hoare's logic (which Hoare himself suggested in his recent treatise [1]) is in close connection with the use of types in modern programming language, even though the main context of the study of types has so far been in λ-calculi rather than in the standard procedural languages. For example, if a program is given a type $\mathbb{N} \Rightarrow \mathbb{N}$, it ensures to return a natural number *provided* it is fed with a natural number as an argument. It is precisely because the abstraction given by types offers information on both what it expects from the composable programs and what a composite program can guarantee, that the safety property provided by a type discipline is in general compositional. Extensive studies of type disciplines are done in the context of ML and other functional languages; as we already mentioned, Morrisett and others extended this framework to assemblers, where type-based abstraction, inherited from the source code through compilation, gives a way to type-check code written in an assembler, ensuring, among others, basic safety.

4 A Challenge for Dependability: Statement and Assessment

Both compositional program logics and type disciplines offer a fundamental framework for compositional safety specifications. A difference between them is that a program logic generally offers a method for specifying (essentially) arbitrary observable properties in a chosen universe of discourse, while a type discipline usually centres on a specific, and often easily checkable, compositional safety property. Since structuring principles and description for computation are intricately related, types and logics also interact at deep levels.

In both program logics and type disciplines, one crucial aspect is missing from our present perspective, which is that neither of them treat correctness when multiple programming languages are involved in a single piece of software. We need to treat multiple languages since, in reality, the pieces of software on which a program depends for its proper function may as well involve those written in other programming languages. For example, if we wish to ensure the quality of a piece of software which is to run under a certain operating system, how can we do so without relying on, hence specifying, the properties of the OS functionalities it relies upon? But if we are to compositionally reason about multiple pieces of software written in different languages, we cannot rely on a program logic for a single programming language: properties of programs in individual programming languages, specified in the corresponding program logics, should be embeddable into a more general framework in which we can integrate specifications for software behaviour written in different languages. Here we are arguing for the need of a unifying theory of program logics for specifying

and verifying general software behaviour, encompassing, and independent from, each specific language.

We may assess the merits and issues involved in this problem domain from three different viewpoints. Firstly, from a scientific viewpoint, having a framework for specifying and verifying properties of software independent from specific details of individual programming languages may be a worthy target, apart from its feasibility: for our understanding of program logics is reaching certain maturity (which now covers a wide range of high-level programming language constructs, extending to even low-level ones), and a scientific aspiration would always aim for the next level of generality.

Secondly, from an engineering viewpoint, we have many problems to tackle for reaching a feasible engineering solution, which are also problems against which the generality and applicability of a candidate basic theory can be tested. As some of examples: How can we specify naturally in such a setting? Can we incorporate the theory into such a general specification framework as Z? How can we develop a mapping for an individual language? What would be a notion of, and calculi for, refinement in this setting? How can such a framework be used for testing software? While there are many issues to be solved, one thing we can be sure about is that we shall never run out of examples by which we can test the feasibility of the idea, since, as already mentioned, almost no software in these days functions without depending on other pieces of software.

Thirdly, from a social viewpoint, such a framework would offer a firm technical footing which enables us to discuss, assess and concretely verify software quality *vis-a-vis* its expected functionalities. If our society has a standard which decrees, in a feasible, realistic way, when we may regard a piece of software as serving its basic functionality, equipped with a suitable way to judge conformance of a program *vis-a-vis* such a standard, this can result in a fundamental boost for raising the quality of software. In fact, without such a framework, we may not even be able to talk about the notion of responsibility in software products, since it is seldom the case that shipped code does not use functionalities of software made by others. Here the theory would help us to delineate the extension of responsibility, as any general engineering theory should.

In terms of concrete activities, we expect that the development of the proposed framework will have many positive interactions with various research threads discussed in the draft proposal [5]. In fact, the nature of the proposed specification/verification framework means possible development in this problem domain is orthogonal to, hence complements, other theories and techniques for software safety, among others those mentioned in the proposal. It is our intention that a produced theory, on which a basic study is being carried out, should work well with theories/notations/technologies which underlie existing tool sets, including program analyses, program verification, program specification and refinement, and safety guarantee by typing; that the framework can be used both for the analysis and specifications; and that it can be used, in effect, as part of a tool set for strong software engineering which the dependable computing project is aiming to produce, especially in a later stage of development. Towards such integration, we believe that the network of activities currently being planned in the draft proposal[5] would offer an essential nurturing ground for the development of the proposed research, as well as possibly broadening the applicability of various existing and emerging methods for dependability.

5 Conclusion

To meet the fundamental challenges towards realisation of dependability of software, we should tackle many scientific and technical issues, as discussed in the draft proposal for Dependable Computing. I argued, in a brief discussion above, that the lack of language-independent framework for specifying/verifying properties of software may be regarded as another issue which may deserve consideration. To try to obtain a new level of abstraction in the theory of specification/verification of program correctness may look too daunting. Perhaps we may not be able to have any such feasible framework, at least in near future.

In a beautiful book [4] on the Z specification method which I read recently, I found an anecdote on a route map of London Underground: the original diagram for the routes of London Tube faithfully modelled all geographic details. This was difficult to read and maintain, so one person proposed to render a diagram so that it abstracts away all these details. This was met with opposition from the manager initially, since it looks too revolutionary. In the current diagram the proposal for high-level abstraction has been adopted, which has indeed become much more readable and easier to maintain than before. The anecdote says that a suitably chosen abstraction worked, not that arbitrary high-level abstractions would work. Yet it does say abstraction is sometimes important, and the abstraction discussed in the present note is, as far as I can see, one of the vital ones for achieving safe, dependable software. As we discussed in the previous section, this abstraction, when materialised, will have fruitful interplay with many of the research threads discussed in the draft proposal, eventually leading to integrated technologies for dependability in general computing environments.

References

[1] Hoare, C.A.R., *Unifying Theories of Programming.* 1994.

[2] Morrisett, G. Walker, D., Crary, K, and Glew, W. From System F to Typed Assembly Language. *ACM Transaction on Programming Language and Systems*, 21(3):528-569, 1999.

[3] Necula, G. and Lee, P. Safe Kernel Extensions Without Run-Time Checking. *2nd Symposium on Operating Systems Design and Implementation* (OSDI '96), 1996.

[4] Woodcock, J. and Davies, J. *Using Z: specification, refinement and proof.* Prentice-Hall International, 1996.

[5] Woodcock, J., O'Hearn, P. and McDermid, J. DSE-Net: *A Network for Grand Challenge on Dependable Systems Evolution.* January 2, 2004.