Modularity Matters Most

Karl Crary* Robert Harper† Peter Lee‡ Frank Pfenning§
Carnegie Mellon University
Pittsburgh, PA 15213

November 1, 2001

Abstract

We contend that modularity is the key to improving software quality. We advocate a view of modularity that emphasizes not the mere assembling of software systems from component parts, but rather the specification of interfaces between components, verification that components meet their specifications, and the assembling only of components with compatible specifications.

Key to this methodology is the use of types to specify and automatically to verify adherence to interfaces. We claim that this methodology makes a higher degree of software correctness possible than has often been achieved heretofore, and moreover, that it may be achieved in a practical manner. To reach this goal will require the development of sophisticated new type system, will require new techniques for modularizing certain correctness properties, and will require a delicate balance between concise code and automated checking.

*E-mail: `crary@cs.cmu.edu` (Corresponding author.)
†E-mail: `rwh@cs.cmu.edu`.
‡E-mail: `petel@cs.cmu.edu`
§E-mail: `fp@cs.cmu.edu`

1 The Importance of Modularity

Over the past decade the software industry has experienced enormous increases in the *quantity* of software production. Unfortunately, we have not seen a similar increase in the *quality* of software or the *productivity* of programmers. As a result, society finds itself increasingly dependent on an ever-growing body of expensive software that is becoming increasingly unreliable, and this situation has occurred despite many concentrated research efforts in various areas such as software engineering, programming languages, and logic. Software today simply is not fundamentally more reliable than it was a decade ago.

Our software systems are becoming increasingly complex and inter-reliant and the techniques and tools provided by the academic community are used only sparsely. In part, this can be attributed to the many barriers to technology transfer. However, we can also recognize that in a number of ways the methods provided by the research community fail to be applicable to the problems faced by developers or maintainers of large-scale, long-lived systems.

To address the quality and productivity problem, software producers have moved to a component-oriented world of programming. A component is a body of code that is deliverable, independently deployable, and ready for integration in larger systems. In an ideal component world, a number of producers would create a pool of interchangeable items. A programmer could then build systems by choosing components from several pools, adapting them, and connecting them as desired.

Building systems from components reduces the construction time and increases quality of software. The reduction of cost is obvious. Even the use of plain libraries, the simplest form of components, reduces the cost of building software systems. The improvement of code is due to the existence of pools of interchangeable components. If one component fails, the programmer substitutes it with a less faulty one. The substitutability of components places pressure on software manufacturers to produce higher quality software.

One important aspect of such component-based software development and maintenance is the need to understand properties of complete systems, their individual components, and how they interact. There is a wide range of properties of interest, some concerned only with the input/output behavior of functions, others concerned with concurrency or real-time requirements of processes. Upon examining the techniques for formally specifying, understanding, and verifying program behavior available today, one notices that they are almost bi-polar. On the one extreme we find work on proving the correctness of programs, on the other we find type systems for programming languages. Both of these have clear shortcomings: program proving is very expensive, time-consuming, and often infeasible, while present type systems support only minimal consistency properties of programs.

We recommend a program of research to improve the quality of software components for heterogeneous systems using programming language technology. We believe that a concentrated effort, involving a balance of fundamental and applied research, can bridge the gap between program proving and type systems, in part by designing and implementing more refined type systems that allow rich classes of program properties to be expressed, yet still be automatically or semi-automatically verified. Through careful, logically motivated design, we believe that the best ideas from abstract interpretation, automated program analysis, type theory, and verification can be combined.

2 The Central Role of Types

The key requirement for the development of robust, maintainable, and composable software modules is a mechanism for specifying modules' invariants and abstractions, and for ensuring that

those invariants and abstractions are respected. Without such a mechanism, individual software components—even correct software components—cannot be assembled into a working software system. Moreover, the stronger the invariants and abstractions that can be specified and enforced, the more robust the resulting system can be.

Central to the research program we recommend is the methodology of types as the key means of providing this mechanism. The key advantage of types over all other means of addressing this need is that types can (and should) be made intrinsic to the programming language in which components are implemented. Thus, properties provided by types, unlike those provided by extrinsic tools, are an essential part of the code: they are created as part of the code, and, since they are automatically verified whenever code is compiled, they cannot be lost or drift out-of-date as the code evolves.

Even simple type systems can guarantee useful properties for software systems, but past research has resulted in more sophisticated type systems capable of specifying and enforcing quite strong and complex properties. The facility to make strong assertions regarding component behavior (thereby ruling out large classes of components that do not abide by those assertions), is the source of expressiveness in a type system. For example, parametric types enforce data abstractions [6, 7], and refinement types [3, 10] enforce value range properties and other data invariants.

We suggest a focus on the use of types to specify and enforce *behavioral contracts* that provide assertions concerning the correctness of a system. We identify three levels of software correctness that will be of interest:

1. The first and weakest level of correctness is *type safety.* A type-safe programming language guarantees the integrity of data for all programs during execution. In particular, a program in a type-safe programming language will not produce an erroneous answer, due to a type-misinterpretation of bits, and will not access data outside of its scope. Despite the weak nature of safety guarantees, little software actually provides this level of correctness, as testified by the commonplace need of operating system mechanisms to protect applications from one another.

2. The second level of correctness is the preservation of a program's key invariants and the integrity of its data. This level of correctness rules out the more severe software failures that occur when data structures become logically corrupted.

3. The third level of correctness is complete conformance of software to its intended behavior.

The third level of correctness is obviously the most difficult to achieve, not only because of the inherent difficulty in developing bug-free software, but also because ensuring software's complete correctness requires a complete specification of its intended behavior, which is usually impractical and often fundamentally impossible. Moreover, this level of correctness is also substantially less important than the other two. In most cases, one can work around a program's bugs, provided its key internal invariants are preserved. However, if a program's data becomes corrupted or if it interferes with other program's data, it is nearly always impossible to make any further use of it.

3 Directions for Research

Due to these considerations, we recommend a focus on the first two levels of correctness with a two-pronged attack. First, we recommend investigation of richer type systems, with the end of providing stronger properties in the second correctness category. Second, we recommend research focusing on the particular problem of verified component integration. We discuss each of these two elements below:

3.1 Richer Type Systems

Most type systems in use in programming languages today provide (at most) the first level of software correctness, type safety. This level of correctness is important, and indeed is a prerequisite for the achievement of stronger levels of correctness. However, the state-of-the-art in type systems research has this level well in hand. We recommend that type systems research now move to ensuring the second level of correctness, protecting invariants and data integrity.

There are a variety of avenues for this research we recommend. In some avenues, research has made progress but there is still far to go; in others, research is only just beginning:

- One category of invariants for type systems to specify and enforce are simple properties of data structures such as value ranges (*e.g.,* for integers: even, positive, or within $\{0, \ldots, 127\}$, for strings: non-emptiness or bounds on length.). Other properties that are somewhat more complicated, but are often amenable to similar treatment, are basic data structure invariants such as the red-black invariants on red-black binary search trees.

  Such properties as these are relatively elementary, but can be extremely important. In fact, the Computer Emergency Response Team (CERT) has observed that a majority of all computer security flaws are due to failures of this sort of property. A variety of *type refinement* systems have been proposed for ensuring properties such as these [3, 10].

- Another category of integrity properties are ones regarding the evolution of state in communicating or reactive software systems [9]. A particularly important example of such properties are those that state that a system's communications adhere to some protocol.

- An important category of properties are bounds on the resources (*e.g.,* time, space, or network bandwidth) that a software component may consume [2]. Despite the obvious importance of such properties (a system that uses too many resources does not work at all), they are often ignored outside of embedded or real-time applications.

- Information flow properties are particularly important for some applications [5, 4, 1, 8]. These properties state that confidential information (or data derived from confidential information), although it may be processed by an application, is never passed on to a subcomponent or external agent that is not authorized to receive that information.

3.2 Verified Component Integration

The research program we recommend into richer type systems can be expected to make progress toward more correct software in *homogeneous* systems, systems that are built from components, all of which were implemented in a single rich type system (typically a single programming language). However, real world systems often do not have the luxury of homogeneity. It is often reasonable to assemble a software system from components verified in entirely different type systems, or even verified using a means other than types.

Therefore, the second element of our recommended research program is to develop means for integrating diverse, verified components into a single verified whole. We suggest an architecture based on three parts:

- A semantically rich interface language for composing heterogeneous software components. Although several interface languages already exist, those that now exist focus merely on data formats (think of this as category one correctness), and do not provide richer specifications component behaviors.

- A "semantic linking" tool, that combines software components, ensuring that the requirements specified in the interface language are adhered to. The semantic linker also would translate data between the representations of the various components in a manner specified by the interface.

- Finally, for some code it may be prohibitively difficult to verify the properties specified by the interface. In the worst case, these desired properties may not even hold. Therefore, our suggested architecture contains an "interface coercion" tool, which automatically transforms code to make the desired properties easy to verify, but at the cost of run-time overhead. This tool would be critical for the utilization of legacy code.

  For example, when a component is required to return an integer within a specified range, but it is impractical to verify adherence to this requirement, an interface coercer might insert wrapper code that checks the component's output to ensure it is within the specified range and signal an exceptional condition when it does not. This wrapper code makes the property easy to verify at the cost of run-time overhead.

4 Some Obstacles to Success

In our recommended research program, there are some obstacles to success that can be anticipated from the beginning. First, the program depends on the composition of individual verified component into larger verified wholes, as a means of improving software quality (*i.e.,* correctness). However, it is clear that not all important properties of programs are easily modularizable. For example:

- Although the resource usage of a software system can be obtained by aggregating the resource usages of individual components, it can be very difficult to specify the resource usages of such components in isolation [2], since they can depend dramatically on how that component is used. Developments in modular resource bound verification will likely need to come in lock-step with developments in specification of data value ranges or even state evolution (discussed above).

- The correctness of floating point goes can be particularly difficult to establish in a modular fashion. Typically the verification of a software component's output is done relative to that component's inputs. In doing so, one implicitly assumes that one's inputs are correct. However, in floating point code, a component's inputs very rarely are exact. This complicates modularization of floating point code, because component that work correctly in isolation (depending on exact inputs) often cannot be composed correctly; rather, each component must deal with possible inexactness.

The second obstacle is as follows: the recommended research program depends on automated checking, to make sure that the properties specified by a component's types are adhered to in its initial implementation and as it evolves. However, it is not currently clear how far mechanical checking can go. As more important (and complicated) properties are attempted, specifications may become as complicated as the code itself.

This difficulty is exacerbated by the tension between checkability and conciseness. To make a rich type system tractable for automated checking, one often requires additional annotations in the code. These additional annotations may be burdensome for the programmer; but, lifting those annotations may make checking take a long time, which is also burdensome for the programmer. A practical type system requires a delicate balance between these two factors.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 184–198, Boston, January 2000.

[3] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[4] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[5] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 1999.

[6] John C. Reynolds. Towards a theory of type structure. In *Colloq. sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[7] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. Elsevier Science Publishers B. V., 1983.

[8] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[9] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4), July 2000. An earlier version appeared in the 1999 Symposium on Principles of Programming Languages.

[10] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of the 26th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 214–227, January 1999.