
Abstract
RS: Radical solution
A radical solution to the problem of <producing an understandable, reliable system implementation> is described, based on <using a new language specifically tailored for representing high-level interconnections between modules>. In comparison with <existing practices based on using usual low-level code for gluing together modules> it offers <improved code transparency and automated analysis>, which have been demonstrated in preliminary tests, but it leaves a number of side effects to be addressed including <the potential for a significantly more complex compiler>. Strategies are suggested for addressing these side effects.

Why RS?
The “module interconnection language” (MIL) approach proposed in this article seems radical for its time (1975), given the absence of a “Related Work” section, the paucity of references to previous MIL-like work (except tangentially, perhaps, the first two reference), and my own ignorance of any work prior to 1975 in this area.

Question - [Method/means of development]
<problem definition>
What is a better way of providing an understandable, reliable system implementation?

Results - [Procedure / technique]
<existing normal solutions>
The old way of doing development was to use the same low-level language for gluing together modules, even though they correspond to high-level concepts.

<solution strategy>
The authors suggest that although low-level code suffices for low-level concepts, programmers would benefit from a higher-level language for connecting modules. Such a language would codify constraints/preconditions/postconditions revealed by the specification and clearly document how modules combine in a structural fashion to satisfy the specification’s high-level requirements.

Validation – [Example]
<advantages>
The new way offers improved code transparency and understandability, with concomitantly improved reliability and maintainability. They attempt to demonstrate this by documenting a theorem proving program’s high-level structure. Qualitatively, their approach does seem to succeed in making the program’s structure fairly obvious, though the authors do not provide a “counter-positive” or “control” for the reader to use as a comparison. Additional validation in a later paper might prove valuable.

<list of side effects>
The authors correctly note that this may result in a more complex compiler. He notes, however, that this is a tradeoff, since the compiler will be able to provide more complete compile-time warnings at a high conceptual level, and that the improved speed and reliability of programmers will more than justify the increased tool complexity.

It is noteworthy the authors claim a main benefit of their approach is to facilitate increased reliability, partially through additional automated code analysis (and comparison to specification). Ironically, this may constitute one significant source of compiler complexity, which is the main “loose end” left to be resolved in future work.