Lightning Talk: Supporting Software Sustainability with Lightweight Specifications

Mistral Contrastin*, Matthew Danish*, Dominic Orchard*†, Andrew Rice*
*Computer Laboratory, University of Cambridge, UK
firstname.lastname@cl.cam.ac.uk
†School of Computing, University of Kent, UK
D.A.Orchard@kent.ac.uk

***Abstract*—Lightweight specifications can aid software maintenance by providing a way to verify that certain properties of a program are preserved. We give two examples of lightweight specifications for numerical code: *units-of-measure types* which specify the physical units of numerical quantities in a program; and *stencil specifications* which describe the pattern of data access used in array computations. Not only can we automatically verify that a program correctly implements these requirements but specifications provide documentation for future developers. Specifications can also be inferred and generated automatically in some cases, further reducing programmer effort. We finish by identifying future potential specification techniques to ease the maintenance and comprehension of scientific code.**

I. Introduction

Being able to comprehend and easily extend a code base is central to software sustainability; inflexible and inscrutable code is difficult to maintain, adapt, and debug in the future. Frequently the intention of the original programmer is not clear from the code alone. There may be an underlying mathematical model from which the code is derived (*e.g.*, in numerical computations in science), but the relationship between the implementation and the model is rarely documented clearly. Programmers often attempt to communicate their original intention by commenting their code, providing informal specifications and descriptions of the program. This approach is often less than perfect: comments must be kept up-to-date with the code they describe and an appropriate level of abstraction must be used to provide effective information (rather than, say, describing each operation line by line).

This informal and manual commenting approach contrasts with full program specification in which a formal and precise mathematical description is provided for a program [5]. This is the precursor to automatic verification, where a verification tool checks that a program behaves correctly with respect to its specification. Full specification of scientific programs is however challenging: specification languages are very different to programming languages (requiring an additional skill set) and writing specifications often requires an understanding of the verification process; full specification requires significant effort. This approach is simply not feasible for the much of the scientific community. Furthermore, it is currently unknown how to effectively specify and verify many high-level numerical properties of programs, such as *convergence* (some work in this direction is by Boldo *et al.* [1]).

We believe *lightweight specification and verification* provides an intermediate solution. Lightweight specifications describe the behaviour of *some* aspects of a program, rather than the whole. This reduces the burden on the programmer whilst still aiding comprehension of the program by others. The specification language can be designed to target a higher level of abstraction than the code itself thereby producing useful specifications which are both human- and machine-readable. We advocate for including such lightweight specifications as inline comments in the code so that the usual tool-chain (compilers, IDEs, version control) is unaffected. The usual verification benefits are provided: a program can be checked for conformance to its lightweight specifications, and this also ensures specifications are up-to-date with the code.

II. Example

We give an example of two such lightweight specification and verification techniques provided by our tool, CamFort, for Fortran code base verification. Figure 1 shows an extract of a Navier-Stokes fluid simulation (based on [4]). The code snippet is constrained by two kinds of specification:

1) The `unit` specifications (lines 1, 4, 8, 11, 14) specify the *units-of-measure* of numerical quantities in the program. This ensures that the units of `f`, `delx`, `dely`, `del_t`, and `rhs` are used consistently. This rules out a common source of bugs from mismatched units.
2) The `stencil` specifications (lines 17, 18) describe the shape of the array access in the approximation computed on line 22. They describe that, at each index `(i, j)`, the arrays `f` and `g` are accessed "backwards" to a depth of 1 in the first and second dimensions respectively. This kind of specification is especially useful when more complicated access patterns are used, and frequently corresponds to choices made when deriving a discrete approximation from a continuos mathematical model.

CamFort has four modes of interaction with specifications:

1) **checking**: CamFort checks that the code conforms to all specifications. If the code does not conform, then information is provided to help identify the source of the program error.
2) **inference**: CamFort can infer specifications automatically, providing useful information and reducing programmer effort. For units-of-measure, a programmer

```
1   != unit(m) :: xlength, ylength
2   real, parameter :: xlength = 22.0
3   real, parameter :: ylength = 4.1
4   != unit(m) :: delx, dely
5   real, parameter :: delx = xlength/imax
6   real, parameter :: dely = ylength/jmax
7
8   != unit(1/s**2) :: rhs
9   real rhs(0:imax+1, 0:jmax+1)
10
11  != unit(m/s)   :: f, g
12  real f(0:imax+1,0:jmax+1), g(0:imax+1,0:jmax+1)
13
14  != unit(s)     :: del_t
15  real, intent(in) :: del_t
16
17  != stencil readOnce, backward(depth=1, dim=1) *
                      ↪  reflexive(dim=2) :: f
18  != stencil readOnce, backward(depth=1, dim=2) *
                      ↪  reflexive(dim=1) :: g
19  do i = 1, imax
20    do j = 1, jmax
21      if (iand(flag(i,j), cf))
22        rhs(i,j) = ((f(i,j) - f(i-1,j)) / delx +
                      ↪   (g(i,j) - g(i,j-1)) / dely)
                      ↪   / del_t
23    end do
24  end do
```

Fig. 1. Fragment of a Navier-Stokes fluid simulation with lightweight specifications added.

need not specify the units for each variable. For example, the unit specification for `rhs` on line 8 need not be given. In *infer* mode, CamFort infers and reports the units of all variables (whether they have been given an explicit specification in the code of not). For stencil specifications, CamFort can infer specifications of the shape of a large class of regular array access patterns.

3) **synthesis**: based on the above inference, CamFort can further reduce programmer effort by inserting automatically inferred specifications into the code where relevant. For example, the specifications on lines 17, 18 can be inferred and synthesised by CamFort entirely automatically without any programmer effort.
4) **suggestion**: (just for units-of-measure), CamFort can suggest a subset of program variables which if given a specification manually by the programmer provides enough information to CamFort to infer the units-of-measure for all other variables.

The inference, synthesis, and suggestion features of CamFort described above further supports the lightweight nature of the specifications. In a previous study, we sought to measure how much CamFort reduces programmer effort via the inference of units-of-measure specifications [7]. We calculated the proportion of variable declarations in a program which required a user-given specification in order to infer the units-of-measure specification of the rest, as reported by the *suggest* mode. On a corpus of forty small programs taken from a computational physics textbook, we found a median of only 18% of variables needed a user-given specification [7, Figure 6], thus an 82% saving in effort.

For the full Navier-Stokes code of which Figure 1 above gave an excerpt, CamFort suggests that only 79 of the 262 variable declarations actually require a user-given specification to infer and synthesise units-of-measure for the rest of the variables: a 70% saving in effort compared with manually specifying the units of every variable.

III. Future directions

There are a variety of directions to explore for future specifications. We give three examples that we are exploring. **1) Software contracts** such as pre- and post-conditions, assertions, and loop invariants can be added to check expected ranges of values and program behaviour. Techniques for inferring contracts are available [6] which would ease the burden on the programmer. **2) Test generation**, *e.g.* QuickCheck [2], provides a way to generate program tests from user-supplied properties of functions and methods. Test inputs are automatically generated and applied, exposing counter examples. **3) Dependency specification** track how a piece of data is used within a program. Long-lived (*e.g.*, global) data is common and specifications which restrict how the data is used throughout the program would allow programmers to make changes and be confident of their scope and influence.

IV. Summary

Lightweight specifications aid software maintainability and reuse by providing high-level information to other developers about the intention of the code. Automatically verifying their correctness ensures that the code remains up-to-date with the specification and can provide confidence in changes made by new developers. Our open-source tool CamFort[1] provides implementations of stencil and units-of-measure specifications (see [3] for more). The synthesis and inference techniques it provides show how tool support can further reduce the burden of using specifications and verification systems.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EP/M026124/1).

References

[1] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: a comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4):423–456, 2013.

[2] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 46(4):53–64, 2011.

[3] Mistral Contrastin, Andrew Rice, Matthew Danish, and Dominic Orchard. Units-of-Measure Correctness in Fortran Programs. *Computing in Science & Engineering*, 18(1):102–107, 2016.

[4] M. Griebel, T. Dornsheifer, and T. Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction*, volume 3. Society for Industrial Mathematics, 1997.

[5] Konrad Hinsen. Writing software specifications. *Computing in Science & Engineering*, 17(3):54–61, 2015.

[6] Francesco Logozzo. Technology for Inferring Contracts from Code. In *Proceedings of SigADA High Integrity Language Technology (HILT 2013)*. ACM, November 2013.

[7] Dominic Orchard, Andrew Rice, and Oleg Oshmyan. Evolving Fortran types with inferred units-of-measure. *Journal of Computational Science*, 9:156–162, 2015.

[1] https://github.com/camfort/camfort