Invited Keynote Talk

Invariant Based Programming

Ralph-Johan Back
Abo Akademi University
Joukohaistenkatu 3-5, 20520 Turku, Finland
www.abo.fi/~backrj
backrj@abo.fi

Abstract

There are a few standard approaches to constructing verified programs. The original approach, by Floyd, Naur and Hoare, assumes that the program code is given, together with an informal description of what the program is supposed to do. Program verification amounts to expressing the requirements as precise pre- and postconditions, finding the appropriate loop invariants, constructing the verification conditions and proving them correct. This is known as *a posteriori verification*. Dijkstra popularized an alternative approach, *correct-by-construction*, where we also start by formulating precise pre- and postconditions. Program code and loop invariants are then derived at the same time, hand in hand, and verification conditions are proved as they arise. The third possibility, *invariant based programming* (Reynolds, van Emden, Back, see [1]), moves the construction of program code to an even later stage. Pre- and postconditions are formulated first, as in the other approaches. The next step is then to formulate the loop invariants, before any code is written. The code is constructed last, as transitions between the different situations (precondition, postcondition, loop invariants) that can occur during program execution. The verification conditions corresponding to these transitions are verified as they arise.

Traditionally, program structure is based on flow of control (conditional statements, while loops, procedures), complemented with data encapsulation (abstract data types, classes). This does not work for invariant based programming, because the transitions that describe the flow of control are introduced only late in the programming process. *Nested invariant diagrams* provide a graphical notation for invariant based programs where the program structure is determined by the information that we have in the different situations. The situations are expressed as sets of states, and program code as transitions between these sets. Nesting of invariants provides an extension hierarchy that allows us to express the program invariants in a very compact manner.

We have studied the feasibility of formulating specifications and loop invariants before any code is written, in a number of case studies. We propose that a systematic use of figures in the initial stages of program construction, as a way of fixing our ideas of how the algorithm is supposed to work, makes it rather straightforward to formulate the invariants needed in the program. We discuss the issues involved in building a program starting from the invariants, what is difficult and what is not, based on our experiences from informal sessions where these methods have been applied in practice. Our experiences indicate that a lot of the difficulties attributed to program verification may in fact not be part of program verification itself, but rather comes from the difficulties in formalizing the basic concepts in new application domains and from lack of experience with using logical reasoning in a stringent way.

Invariant based programming, as any verification approach, requires a lot of verification lemmas to be proved. Most of these lemmas are rather trivial, but there are a lot of them. A few are difficult, and often originate from the central algorithmic idea on which the solution is based. Machine support for proving verification conditions should therefore be very useful. We have built a prototype tool, Socos, which provides an integrated environment for drawing nested invariant

diagrams and for verifying verification conditions automatically, using Simplify. Those lemmas that cannot be verified automatically are then proved interactively with PVS.

We conclude with a discussion on how the proposed approach could be used to teach a formal approach to software construction at different levels of educations – high school, polytechnic and first year university courses. We propose that teaching logic as part of the ordinary mathematics curriculum and a course on invariant based programming could serve as a simple basis for teaching students to write programs whose correctness they can check for themselves with rigorous mathematical reasoning. We are currently evaluating this approach in two freshman courses taught at Abo Akademi University, the first one concentrating on teaching mathematics using structured derivations (an extension of Dijkstra's calculational derivations), and the second one teaching invariant based programming.

REFERENCES

- [1] Ralph-Johan Back: Invariant based programming. In S. Donatelli, P. S. Thiagarajan (Eds.): *Petri Nets and Other Models of Concurrency - ICATPN 2006, 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, Turku, Finland, June 26-30, 2006. Lecture Notes in Computer Science 4024 Springer 2006, ISBN 3-540-34699-6