Preprocessing for Invariant Validation

E. Pascal Gribomont
University of Liège (Belgium)

Topics: formal verification of concurrent systems, theorem proving systems

Abstract. Hoare's logic and Dijkstra's predicate transformer calculus have proved adequate for reducing the correctness problem for programs to the validity problem for logical formulas. However, the size of the logical formulas to be validated grows faster than the size of the program, and, even in the propositional case, the validation problem is NP-complete and becomes practically intractable for large programs. We introduce a strategy for dealing with this problem. The principle is to write the formulas in the form \((h_1 \land \cdots \land h_n) \Rightarrow c\), and to use efficiently computable criteria to select a small subset \(I \subseteq \{1, \ldots, n\}\) such that \(c\) remains a logical consequence of \(H_I = \{h_i : i \in I\}\). These criteria are motivated and the efficiency of the method is investigated.

1 Introduction

A classical method for proving that some concurrent system \(S\) is correct with respect to some safety property \(J\), when initial condition \(A\) holds, is the invariant method. In this case, an invariant is a formula \(I\) satisfying three conditions:

1. \(A \Rightarrow I\);
2. \(\{I\} \tau \{I\}\), for all atomic actions \(\tau\) of \(S\);
3. \(I \Rightarrow J\).

The notation \(\sigma \models B\) means that formula (or assertion) \(B\) is true in state \(\sigma\), and \(\models B\) means that \(B\) is true in every program state.

Condition 1 expresses that \(I\) is true initially, condition 2 means that \(I\) is respected by each transition \(\tau\) of the program, that is, if state \(\sigma\) satisfies \(I\) and if transition \(\tau\) leads from state \(\sigma\) to state \(\rho\), then state \(\rho\) also satisfies \(I\). When these three conditions hold, it is clear that all states of all computations (whose initial state satisfies \(A\)) do satisfy the safety requirement \(J\); otherwise stated, \(J\) holds throughout the computation.

This method is reputed to have three drawbacks:

1. Only safety properties are within the scope of the method.
2. Invariant design is often a subtle task.

1 Correspondence to:
P. Gribomont
Institut Montefiore
ULg, Sart-Tilman, B28
B-4000 Liège (Belgium)
3. Invariant validation becomes intractable for large programs.

The first point is of little practical significance, since the liveness properties of a program (the most relevant kind of non-safety properties) usually are consequences of previously established safety properties, joined with a fairness assumption. (This has been demonstrated many times in the literature, for instance in [7].) The second drawback is more real, but the discovery of adequate invariants is made easier by using an incremental methodology (more about this later). The third drawback is more serious. Most invariant design methodologies, including the incremental one just mentioned, are in fact trial-and-error methods: "promising candidate invariants" are rather easily produced, but have to be checked for invariance. Even for programs of moderate size, a complete verification is an awfully long and tedious work. The purpose of this paper is to contribute to solve this problem.

It should be noted first that the verification conditions associated with a program and an invariant are not "difficult" formulas, in so far only elementary mathematical results about data models (integers, arrays, and so on) are needed to validate them. The only problem is the size of the formulas. A straightforward approach is to use an automatic theorem prover. Many successful experiments with theorem provers, and especially with the well-known Boyer-Moore prover, have been reported in the literature (see e.g. [13, 25, 26]). In fact, this prover can deal with far deeper mathematical formulas than verification conditions, but it is not able to deal quickly with very long formulas, for an obvious reason. Even in the pure propositional case (and we are seldom in this case) the validity problem is NP-complete, so no general algorithm for validity checking can be practically efficient in all cases.\footnote{It is well-known that some NP-complete problems allow algorithms which behave efficiently for most instances occurring in practice but, until now, this is not the case for the propositional validity problem, although rather recent techniques like connection-based methods and the use of ordered binary decision diagrams have led to substantial improvement.} We intend to take advantage of the very specific structure of invariant verification conditions to produce a practically efficient validation system; for now, only "nearly finite-state" systems are considered, i.e., systems for which nearly all variables have finite range [17].

The sequel of this paper is as follows. Section 2 recalls briefly the invariant method with an example; the specific structure of verification conditions is described. Our algorithm is introduced and investigated in Section 3. Section 4 is devoted to a worked-out example and Section 5 contains comparison with related work and the conclusion.

2 Structure of invariant verification conditions

2.1 Construction of the invariant and verification conditions

The procedure is best recalled on a short example (see e.g. [16] for more details) of a simple algorithm for mutual exclusion.