Toward Verifiably Correct Control Implementations

Bugs may be introduced into control applications at all levels, starting from the high-level mathematical control laws to the actual machine code, complete with device drivers and multitasking. An important scientific and technical challenge for the controls and real-time software communities is to design analysis methods at these various levels of abstraction, along with verified compilation and synthesis tools.

State of the Art

- The CompCert compiler from INRIA and University of Rennes-I compiles C to a variety of popular targets (PowerPC, ARM, x86). The compiler has been proven correct mathematically with a machine-checkable proof. The assembly programs produced thus provably preserve the semantics of the source C code.
- The Astrée analyzer can verify many control system implementations in C if they are fairly static—excluding parallelism, dynamic scheduling, dynamic data structures, virtual methods, etc.

Safer, More Powerful Compilation

Compilers are software and as such may contain bugs. A bug in a compiler may result in the introduction of bugs in the object code the compiler generates, and thus in the program as it is executed in the embedded systems. Such bugs may be difficult to find, and thus for certain safety-critical systems, object code must be matched to source code for inspection, ruling out code optimization. However, disabling optimization leads to inefficient object code, requiring higher CPU performance or limitations in functionality.

Although progress has been made in safe compilation for programs written in C, an outstanding challenge remains for compilers for high-level specifications such as Simulink—a preferred formalism for many control systems—or complex languages such as C++.

In turn, this implies that the high-level specification language should have reasonable and unambiguous semantics.

Enlarging the Scope of Static Program Analysis

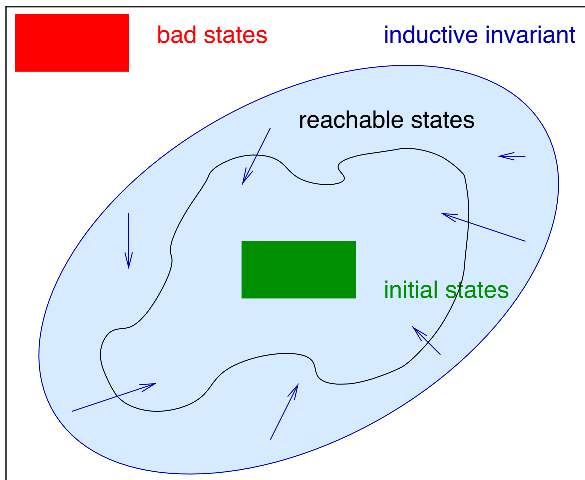
Static program analysis refers to the automated analysis of computer programs without actually executing the programs. Despite the recent availability of industrial-strength program analysis tools, considerable challenges remain.

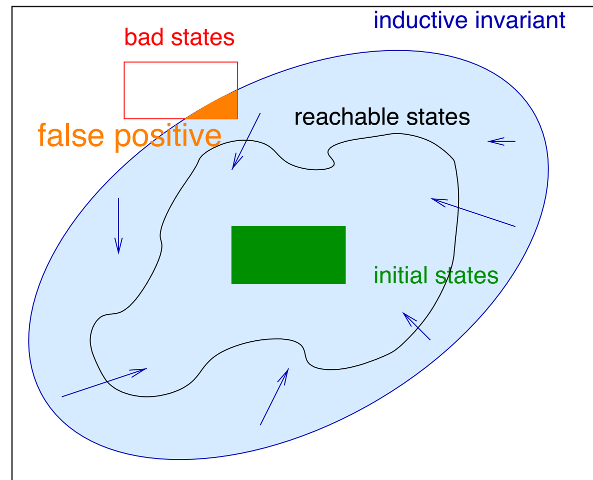
- The spectrum of applications needs to be increased. Fewer restrictions on programming styles and technologies should be imposed while keeping the likelihood of false alarms low.
- As with compilers, analysis tool implementations should be formally proven correct with machine-checkable proofs. This is especially important if, for critical systems, some testing is replaced by static analysis.
- Speed and automation need to be enhanced. Tools should be able to prove desired properties with minimal user intervention and to provide counterexamples in case properties are not verified.

During the Ariane 5 rocket's maiden flight in 1996, flight control software malfunctioned and the rocket had to be destroyed by remote command early in its trajectory. The malfunction was the result of improper reuse of Ariane 4 software. Hardware redundancy was no help since both computers ran the same incorrect software.

Contributor: David Monniaux, VERIMAG, CNRS, France

From: *The Impact of Control Technology*, 2nd ed., T. Samad and A.M. Annaswamy (eds.), 2014. Available at www.ieeecss.org.





A static software analyzer finds an inductive invariant, which includes but may overapproximate the initial states and all possible reachable states. If this invariant excludes bad states (as in the left graphic), the analyzer proves the absence of errors in any possible execution of the software. If the bad states intersect the inductive invariant but not the reachable states (right graphic), a false positive results.

Toward a Trusted Development Chain

High-level specification typically considers idealized mathematical computations. In reality, differential equations are discretized—for example, real numbers are implemented using a floating-point or fixed-point arithmetic; multiple clock domains may be used; mathematical functions may be approximated; and programs are split among different tasks or machines, which may not be in perfect synchronization.

Some of these transformations are automated, but many are still performed by hand, most of the time with no mathematical proof of their correctness. Tools are needed that automate these transformations or at least provide meaningful feedback to implementers.

Control applications increasingly run on multicore processors, including for critical embedded systems. Manual programming for parallel systems is notoriously error-prone. Shared memory implementations require careful placement of locking mechanisms—too few of them and *data races* may occur, but too many of them and *deadlocks* may freeze the system. Automated synthesis or verification of the parallel or distributed implementation, possibly with a formal proof of correctness, becomes increasingly desirable.

Communication protocols, especially on modern buses, are hard to get right; this is even truer when security properties are involved (e.g., resistance to eavesdropping or intrusion). Implementations of such protocols should be based on reusable, well-tested, or even formally proven libraries, and nonreusable parts should be synthesized from specifications. Doing this effectively and safely remains a research challenge.

```
int main() {
 int x = 0;
 int y = 0;

 while (1) {
  /* invariant:
  102 + -y + -x >= 0
  -y + x >= 0
  y >= 0
  */
  if (x <= 50) y++;
  else y--;

  if (y < 0) break;
  x++;
 }
}
```

Static analyzers, in this case the experimental tool Pagal, may display loop and function invariants. This helps developers understand what is going on in their software so it can be debugged more efficiently.

For more information, see the companion flyer on "Verification of Control System Software" in the Success Stories section of this volume.