Developing Programming Abstractions for Cyberphysical Systems

Sayan Mitra
mitras@illinois.edu
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Instruction Set Architectures (ISAs) define a generic interface of hardware for program-
ming computers. The execution of an ADD instruction, for example, hides the complexities
of transistor physics, with the expected semantics that the sum of the operands will be
stored in the target register, and that nothing else would change. This interface is vital for
the blossoming of the ecosystem of programming abstractions, microarchitectures, software
libraries, run-time systems, and applications.

Can we find a generic interface for programming cyberphysical systems? Unlike packaged
ICs, cyberphysical systems are open and exposed to the environment. The torque generated
by a “control instruction” depends not only on the the physics of the motor—which can be
considered to be part of the machine—but also on the wildly varying drag. Consequently,
every control instruction (controller) is specifically designed for the underlying hardware,
the anticipated environment, and the higher-level objectives. Control programs cannot be
safely ported from this year’s car model to next year’s, let alone across different cars or
vehicles. This specificity also stymies innovation in higher-level programming abstractions.

Consider programming an advanced alerting system that estimates the space occupied
by potentially offending vehicles based on information received from other vehicles as well
as motion estimates computed from telemetry and cameras [9, 11]. Here, the notion of the
occupied space should not just be a variable that is updated only when new messages are
received, but as the communication links and on-board computers can fail, this variable
should be updated continuously even in the absence of messages. This calls for a new
programming abstractions—perhaps, one with continuous, hybrid, or time-triggered variable
updates [6, 7]. Distributed and asynchronous nature of vehicle-to-vehicle and vehicle-to-
infrastructure systems bring to mind other kinds of useful programming abstractions [8, 5].

Benefits of a good programming abstraction go beyond improving productivity and
portability. As the static and dynamic analysis techniques for cyberphysical systems come
of age (see recent developments, for example in [4, 10, 3, 1, 2]), the common language and
its idioms can catalyze research on development of significant software frameworks and tools
for design automation, verification, and testing.

There are several interesting technical challenges in designing programming interfaces
for CPS. For example, their openness require that the CPS interfaces should not only spec-
ify progress—what changes with the successful completion of a control instruction, but like
the ISAs, they should also specify invariants—things that remain unchanged. These invari-
ants should be useful for compositionally restricting behavior while they should not be so
restrictive as to become platform specific or unimplementable. One way forward is to propose candidate abstractions and demonstrate their utility by building increasingly more general classes of CPS. It should be possible to support and sustain such endeavors as the computing industry has repeatedly demonstrated the profitability of investing in common core abstractions and supporting technologies, while competing for higher-level services and applications.

References


