Position Paper: Verifying Concurrent Programs using Locally Checked Invariants
Ernie Cohen
Microsoft Corp
ernie.cohen@acm.org

VCC (“Verified Concurrent C”) is a sound, deductive verifier for concurrent C programs, developed at EMIC and MSR. VCC is designed to allow developers to verify functional correctness of real-world software (particularly system software) by annotating their code with function contracts, loop/data invariants, and ghost data/code. Verification in VCC is thread- and function-modular, and annotations follow the scoping rules of C.

The VCC state is essentially a collection of objects (some of which may be ghost), each with a 2-state invariant (given by the object’s type). An object invariant can mention arbitrary parts of the state, subject to the admissibility requirement that its invariant cannot be violated by any state update, starting from a state in which all invariants hold, preserves the invariants of all updated objects. (This can be viewed as a state-based analog of the input-enablement requirement for IO automata, and is checked using only the type definitions.) Admissibility guarantees that when checking an atomic action, we only have to check that it preserves the invariants of the modified objects.

In VCC, forward simulation is done within the code, rather than in a separate proof. In a typical verification, the abstract state is encoded as a ghost object whose fields represent the protocol state, with a two-state invariant that specifies the protocol transition relation (much as one would do in TLA). Requirements/invariants of the protocol are given as invariants of a separate ghost object. To prove that an implementation simulates the protocol, we introduce coupling invariants between the concrete and abstract states. Updates to the concrete state are then “annotated” with ghost updates to the abstract data that “witness” the simulation.

A ghost object without volatile state can be viewed a first-class chunk of knowledge (the stability of this knowledge proved by the admissibility
Indeed, at a scheduler boundary, a thread forgets everything it knows about state except for the state and invariants of the objects that it (transitively) owns. Encapsulating knowledge as first-class objects allows it to be passed in and out of functions, stored in data structures, sent in messages, mentioned in object invariants, and its lifetime managed using conventional programming techniques (in ghost code). Moreover, unlike state assertions, the stability of a chunk of knowledge doesn’t have to be revalidated at a scheduler boundary, where the information needed to prove its stability are likely to be out of scope.

Another common idiom that can be represented with invariants is the notion of a right. In VCC, the right to perform an action is just knowledge that performing the action will not break any object invariants (or, in a more refined notion of right, knowledge that the action will not break certain object invariants). For example, a thread can be viewed as an object whose invariant is given by the proof outline needed to verify it. If some other object has as its only invariant that changes to its state preserves the invariant of the thread, it presents to the thread the logical illusion of ownership - the thread can admissibly say anything it wants about the object state (making it appear that nobody else can change it). Using this idea, it is easy to encode in VCC concurrent programming disciplines such as transferable ownership, but also separation logic concepts such as counting/fractional permissions and concurrent abstract predicates; indeed, these disciplines can be combined within a single program, without baking them into the logic.

To handle protocols like 2PC, we model transient failure as a program that fails by simply giving up to the system all of the objects that it owns, allowing another copy of the same thread to start. This induces the proof obligation that the objects required at thread entry can never be destroyed or their ownership transferred from the thread while it is running. We have used similar techniques to reason about interrupt handlers in operating system kernels.