Automating Proofs of Coarse-Transaction Properties of Data Abstractions
Trek Palmer and Eliot Moss
Computer Science, University of Massachusetts Amherst

Overview and Motivation: Coarse-grain transactions, such as those implemented by transactional boosting and open nesting, offer the possibility of the maximal concurrency and reordering allowed by transactional semantics on a given data abstraction. To support this, both boosting and open nesting expect the programmer to supply information about which operation invocations conflict semantically, as opposed to conflicting just incidentally on fields of a concrete representation. For pessimistic concurrency control they also require the programmer to designate inverses for operation invocations (also called undos or compensating actions). These requirements raise the concern that programmers may offer incorrect descriptions of conflicts or inverses. We address that concern by offering (a) a notation (language) for specifying abstractions as opposed to implementations of abstractions, (b) means to describe putative conflicts and inverses, and (c) a tool that can prove or disprove the correctness of conflict predicates and inverses relative to an abstraction. We have further added to our notation means to describe abstract locks, which offer a computationally effective way of enacting conflict testing, and which we can prove relative to an abstraction or relative to conflict predicates. As a necessary adjunct, we support describing invariants of abstractions and can prove those invariants correct.

The Language: We stress that our language, called ACCLÅM, is for describing abstractions, not implementations. Some of the classic approaches to defining data abstractions are axiomatic semantics, algebraic semantics, and abstract models. We pursue abstract models as being easier for programmers to write (you don’t have to be a mathematician) and as admitting simpler proof technology. In ACCLÅM, code for basic operations on an abstraction is free of loops or recursion, which ultimately allows us to translate desired proof properties into propositions suitable for off-the-shelf satisfiability solvers. However, ACCLÅM replaces the arrays of Java with arrays where the index can be any type. This allows simpler descriptions of abstractions without overly committing to an implementation strategy. For example, a set of objects can be represented by an array $a$ of boolean indexed by object, where $a[x]$ is true iff $x$ is in the set. This is entirely agnostic about the physical organization of an actual implementation. Even though basic operations admit no loops or recursion, ACCLÅM does offer a conditional bulk update construct that acts in essentially SIMD fashion. Coupled with our abstract arrays, it allows succinct expression of many operations without true iteration. For more powerful operations such as transitive closure, one can write non-basic operations in terms of basic ones.

Relation to Transactions: ACCLÅM assumes that each basic operation is executed as part of a transaction, and that any concrete implementation of the abstraction gives a correct linearizable implementation of the basic operations. Similar to boosting, the goal is to take a linearizable concurrent abstraction and build from it a transactional abstraction. A transactional abstraction allows a thread to present an arbitrary sequence of operations to a system that executes them serializably, that is, where the whole sequence appears atomic. Since non-basic operations are built from basic ones, their serializability follows from the serializability of general sequences of basic operations. Note: Proving that an implementation of a concurrent abstraction is actually linearizable is a different problem! However, such proofs are simplified if implementations are in terms of transactional

\[^1\text{Pronounced like acclaim.}\]
memory, where correctness derives from proper implementation of the TM system rather than possibly tricky coding of the specific data type implementation. But ACCLÄM does not care about implementations, as long as they are linearizable. Note also: While ACCLÄM, because it is a way to describe an abstraction as distinct from any particular implementation, might be a useful component in an overall scheme to prove correctness of an implementation of an abstraction, that also is a different problem from the one we address here. Here we are concerned only with whether conflict predicates and inverses are sound, and that abstract invariants hold.

What we can prove: First, we can prove correctness of abstract inverses. If running $a$ on argument vector $\vec{x}$ in state $\sigma$ produces result $r$, final state $\sigma'$, and designates inverse $a^{-1}$, then we can show whether or not running $a^{-1}$ in state $\sigma'$ produces the original state $\sigma$. Since we use a SAT solver, we actually search for possible states $\sigma$ and argument vectors $\vec{x}$ where applying $a$ and then the designated inverse, which may depend on $\sigma$ and $\vec{x}$, fails to produce $\sigma$.

Likewise, given an operation $a$ with arguments $\vec{x}$ that executed in state $\sigma$ produced result $r$ and state $\sigma'$, and a desired operation $b$ on arguments $\vec{y}$ that then produces result $s$ and state $\sigma''$ from state $\sigma'$, we can demonstrate whether or not executing $a(\vec{x})$ and $b(\vec{y})$ in the opposite order produces different results or a different final state. More than that, we actually prove, given a conflict predicate that claims to indicate when the operation invocations conflict, whether there exists any combination of initial state and arguments that produces different results or final state when the conflict predicate is false. This shows whether a conflict description is sound. We further show whether supposedly non-conflicting operations undermine the applicability of inverses of previously executed operations.

By considering a converse to our conflict predicate proofs, we can tell a programmer whether a conflict predicate is needlessly broad, possibly restricting concurrency more than necessary.

Conflict predicates are an approach to specifying the maximum concurrency allowed by a transactional construct, but they do not, of themselves, offer an effective implementation of concurrency control. Therefore we also support describing abstract locks that an operation invocation acquires, and can prove whether the locks acquired enforce the stated conflict predicate. We can also prove correctness of abstract locks directly from (non)commutativity of the abstraction, without requiring the programmer to write conflict predicates.

Finally, ACCLÄM allows writing of invariants on abstract models. For example, in specifying an ordered set abstraction, we use invariants related to proper recording of the next higher and next lower member of the set for any given object of the kind stored in the set. These are sometimes necessary to prove inverses, conflict predicates, or locking schemes. Therefore we also support proving that the invariants themselves are correct. Given our proof technology, invariants may not alternate quantifiers.

Initial results: We are finding that ACCLÄM descriptions are concise and not too hard to understand. Writing models does require different thinking from writing code (implementations), but it grows on you. Of particular promise is that proofs with SAT solvers complete in fractions of a second to a few seconds, even though there remains significant room for optimizing the size of our SAT problems.