Introducing Abstractions Via Rewriting

William D. Young

Department of Computer Sciences
The University of Texas at Austin*
byoung@cs.utexas.edu

Abstract. Mechanically assisted proofs of properties of a complex system require an accurate formal model of the system. If the model is too detailed the proof becomes intractible. We outline techniques for automatically "retrofitting" a detailed low-level model with abstractions that facilitate reasoning about the properties of a model. The abstractions are introduced through semantics-preserving rewrite rules. We have applied this technique to the Rockwell-Collins AAMP7 processor model and been able to improve significantly the analyzability of the model.

Mechanically assisted proofs of properties of a complex system require a formal model of the system. However, if the model is too detailed the proof may become intractible because of the overwhelming morass of low-level detail that must be managed. This is especially true if portions of the model are machine generated. This was the case with the Rockwell Collins AAMP7 processor model.[1, 3] This is a very low-level specification of the AAMP7 instruction-level semantics and was partly generated by macro expansion from an imperative notation embedded in the ACL2 formal logic. Because of the lack of abstraction, the model is hard to understand and difficult to reason formally about.

Our goal was to prove properties of machine language programs for the AAMP7 using the existing formal model as an operational semantics. However, the model proved too low-level for our purposes. Rather than reconstruct it, we developed techniques for automatically "retrofitting" the detailed low-level model with abstractions to facilitate reasoning about properties of the system. The abstractions are introduced through semantics-preserving rewriting. We have applied this technique to the AAMP7 model and been able to improve significantly the analyzability of the model. We used the ACL2 system[2] to manage the process, the ACL2 rewriter to replace complex terms by more abstract versions, and the theorem prover to assure that the process preserves semantic equivalence.

In addition to providing a more intelligible and accessible formal characterization of the AAMP7 instruction-level semantics, there was a rather surprising additional benefit. The addition of abstractions illuminated numerous inefficiencies; the abstracted model could actually be faster than the low-level model.

* This work was supported at the University of Texas at Austin by a contract from Rockwell Collins, Project #450117702, *Instruction-level Model of the AAMP7 in ACL2.*

1 The AAMP7 Model

The AAMP7 model is a detailed instruction-level model of the Rockwell-Collins AAMP7 microprocessor. Executable specifications for the AAMP7 processor were written in the logic of ACL2[2] and formally analyzed to satisfy a variety of properties, including well-formedness of definitions, type restrictions on the arguments to functions, and formal relationships among various functions in the specification. All of these proofs were mechanically checked using the ACL2 theorem prover. The model comprises many megabytes of formal specification, executable code, and supporting theory.

To make the specification more perspicuous, a macro was defined that allows specifying the semantics of individual AAMP7 instructions in an imperative style. For example, the `op-addu` function below describes the semantics of the AAMP7 `addu` operation, which takes two 16-bit unsigned values from the top of the stack, adds them using modular unsigned integer arithmetic, and pushes the result back onto the stack.

```
(defun op-addu (st)
  (aamp *state->state*
        (pop ux) (pop uy) (push (uword16 (+ ux uy)))
        st))
```

Here, `aamp` is a macro defined within ACL2 that interprets its arguments as follows: The first argument specifies that this function is a state to state (as opposed to a value-returning) transformation. The effect on the state is equivalent to executing the listed pseudo-instructions in sequence. Local variables such as `ux` and `uy` are introduced where needed.

The `aamp` macro essentially embeds within ACL2 a readable and intuitive, imperative language for specifying operation semantics. But because ACL2 is an applicative language, expansion of the macro must emulate this imperative notation by translating it into an applicative form. The required translation is quite complex. The list of instructions in the body of the `aamp` form is transformed into a nested series of accesses and updates on a record of some 60 fields that represents the processor state. The expansion contains conditional branches for reset, trap and interrupt behaviors, user versus supervisor modes, and all of the possible exceptions that could arise. Details of the modular arithmetic and bit string manipulations involved in AAMP7 address computation and instruction execution are revealed. When macro-expanded, the call `(op-addu st)` takes over 1200 lines (as formatted on my screen).

Because the semantics is defined using macros that are eliminated by ACL2 during preprocessing, there are essentially *no intermediate abstractions* between the easily comprehensible definition of `op-addu` above and the "real story" that confronts the user of ACL2 attempting to reason about a program involving the `addu` operation.

2 Introducing Abstractions

Our solution was to develop automated techniques to introduce conceptual abstractions into the existing specification. The approach we took was to identify recurring low-level forms within the AAMP7 specification, and mechanically rewrite them into a more abstract and perspicuous form. This in turn may reveal a second level of abstract notions, which can then be introduced mechanically, and so on.

For example: in the expansion of `(op-addu st)`, the following form appears numerous times: `(wfixn 8 16` k`)`. This is a standard locution generated by the `aamp` macro for coercing an arbitrary value k into an unsigned 16-bit integer. This suggest introducing an abstraction for this concept, say `(fix16` k`)`. Using the ACL2 macro facility, we defined a new syntax to add such abstractions.

```
(defabstractor fix16 (x) (wfixn 8 16 x))
```

This form defines a new function symbol `fix16` of one argument and introduces a rewrite rule to unconditionally replace occurrences of expressions of the form `(wfixn 8 16` x`)` with the corresponding expression `(fix16` x`)`. To prevent looping the non-recursive function `fix16` is also "disabled" to prevent it from being automatically expanded by the prover. Whenever ACL2 subsequently encounters an expression of the form `(wfixn 8 16` x`)`, it will replace it with the corresponding expression `(fix16` x`)`.

This simple idea is surprisingly powerful. Using these abstractor functions, we can construct a hierarchy of abstractions, and begin to build an "algebra" of rewrites for our specification domain. For example, updates to different state components can be commuted and multiple, redundant and offsetting updates to the same state component can be collapsed into a single update.

As an example, within the macro-expansion of `(op-addu st)`, the following expression appears:

```
(ash (makeaddr (aamp.denvr st)
       (gacc::wfixn 8 16 (logext 32 (+ -1 (gacc::wfixn 8 16
           (+ 1 (gacc::wfixn 8 16 (+ 1 (aamp.tos st))))))))) 1)
```

Under the assumption that certain intermediate results are representable, this entire expression reduces to: `(stack-address 1 st)`. Because the same basic forms are used throughout the AAMP7 specification, a relatively small collection of well-chosen abstraction functions provide enormous conceptual clarity.

Using our abstraction approach, we generate for each AAMP7 instruction a theorem that characterizes its operational semantics, assuming that we are executing in the "expected" case. For example, for the `addu` instruction we assume:

1. the reset, trap, and interrupt flags are not asserted;
2. the top-of-stack and program counter computations are within bounds;
3. the PMU is configured to allow the accesses required.

Under these conditions, our semantic theorem says that the state resulting from stepping the machine over an `addu` instruction is like the input state except that:

1. the sum of the top two stack elements replaces the second stack element;
2. the top-of-stack pointer is incremented;
3. the next instruction byte has been pre-fetched;
4. the pc is incremented;
5. two temporary locations contain specific values.

Subsequently, symbolically stepping the AAMP7 model on an `addu` instruction can be accomplished by applying this rewrite rule, which provides an alternative semantics for the `addu` operation. This semantics is significantly easier to deal with in a proof context than the definition, and allows conceptualizing execution at the level of the abstraction functions, rather than having to deal with the low-level details.

The introduction of abstractions had a rather surprising side benefit: the abstracted versions are more computationally efficient. The macro-expansion in the original emulates an imperative program in an applicative context. The result is a set of nested updates to the state, many of which are redundant, cumulative, or offsetting. The fog of detail in the macro-expanded version tends to hide these inefficiencies. The abstracted version, on the other hand, reveals obvious simplifications that can be implemented as rewrites. Our abstract semantic function for `addu`, for example, replaced several dozen distinct state updates with six. Moreover, since the abstracted, optimized version is proven semantically equivalent to the original, we could replace the original simulator with one that runs our more efficient versions.

We have demonstrated an approach to "retrofitting" an existing low-level specification with abstractions. This is a potentially valuable tool for rendering a complex low-level specification more intelligible and more amenable to formal analysis. Moreover, even a specification that was designed for efficient execution may have inefficiencies that are hidden by complexity. The abstraction process may make such inefficiencies more readily apparent. This effort re-emphasizes the value of abstraction to manage complexity and to facilitate proof. But it also suggests that it is possible in some cases to introduce abstraction into an existing specification.

References

1. David Greve, Matthew Wilding, and David Hardin. High-speed, analyzable simulators. In M. Kaufmann, P. Manolios and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Press: Boston, 2000.
2. M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, Boston, 2000.
3. Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, 18(3):233–248, May 2001.