Reverse Abstraction in ACL2

William D. Young
University of Texas at Austin[1]
Department of Computer Sciences

1 Introduction

Formal models of digital systems are constructed for a variety of purposes, not all of them mutually compatible. A model that is constructed as a formal simulator for a digital system may be highly optimized for efficient execution. A more abstract model may be less efficient, but structured to facilitate proofs of system properties. In an ideal world, any model would be abstract, comprehensible, analyzable *and* efficiently executable. However, it is often difficult to build a single model that supports such disparate goals.

One possible solution is to construct an abstract system model, and then refine it through some series of steps to eke out the required level of execution efficiency (eg. [4]). Sometimes, however, the artifact at hand is not an abstract model that can be refined for efficiency, but a very low-level model that has been hand-tooled for efficient execution. This was the case with the Rockwell Collins AAMP7 processor model[3]; the need for execution efficiency led to a very low-level specification highly optimized for speed. Because of the lack of abstraction, the model is hard to understand and proofs very difficult to construct.

To facilitate proofs of AAMP7 programs, it became desirable to introduce abstractions into the model. Since it was not possible in the current project to rebuild the model from scratch, we decided to investigate the feasibility of retrofiting the model with appropriate abstractions. By analogy with "reverse engineering," we call this process "reverse abstraction."[2] The goal is to replace a low-level system model with a more conceptually abstract version that is provably semantically equivalent, but more amenable to formal reasoning. It is clearly infeasible to mechanically recognize appropriate abstractions from a low-level specification. But, once a human has identified useful abstractions, automation can assist in managing the reverse abstraction process, and assuring that inserting abstractions preserves the semantics of the target system.

In this report, we describe the reverse abstraction process as applied to a very detailed existing formal specification, the Rockwell Collins AAMP7 formal model. We used the ACL2 theorem proving system[2] to manage the process, used the ACL2 rewriter to replace complex terms by more abstract versions, and used the prover to assure that the process preserved 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. Even though the

[1]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.*

[2]One of the anonymous reviewers thought this term misleading, because the two versions are semantically equivalent. I would argue that "abstraction" is often used in places where that is true. "Procedural abstraction," for example, doesn't necessarily imply elimination of detail, merely hiding detail to repackage functionality into a more congenial form. I believe that that is precisely what is accomplished here.

low-level AAMP7 model was constructed for efficient execution, the reverse abstraction process illuminated several inefficiencies. The abstracted model could actually be faster than the low-level model, though we have not yet validated this.

2 The AAMP7 Model

The AAMP7 model is a detailed instruction-level model of a commercial processor. It comprises many megabytes of formal specification, executable code, and supporting theory, and represents a monumental intellectual effort in the application of formal methods to digital design. This effort was supported by the ACL2 automated analysis tool-suite. Executable specifications were written in the logic of ACL2 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 ACL2 macro facility is used in an extremely sophisticated manner in this specification. A *reader* macro is defined that allows the specification of individual AAMP7 instructions in an imperative style. For example, the following function describes the semantics of the `LIT16` operation, which takes a 16-bit literal from the instruction stream and places it on the processor stack.[3]

```
(defun op-lit16 (st)
  (declare (xargs :stobjs (st)))
  (AAMP *state->state*
        (fetch_word ux);
        (push ux);
        st))
```

Here, `AAMP` is a macro that interprets its arguments as follows. The first argument specifies that this function is a state to state transformation. The effect on the state is equivalent to executing the listed pseudo-instructions in sequence and then returning the resulting state. Local variables are introduced where needed.

The `AAMP` macro essentially embeds within ACL2 an intuitive, imperative language for specifying processor operations. Macro expansion must transform this into an applicative form, typically a functional expression involving accesses and updates on the single-threaded object[1] representing the state. The macro expansion of `(op-lit16 st)` is shown in Figure 1. Note that `LIT16` is one of the simplest AAMP7 operations. By comparison, the macro expansion of the corresponding body of the `ADD` instruction is several hundred lines of text.[4]

The AAMP macro facility provides a congenial vehicle for specifying the semantics of operations. However, it presents a nightmare for anyone attempting to prove properties of AAMP7 instructions. The prover immediately expands the macro call, and whatever abstraction is provided by the

[3]The AAMP7 model is a moving target. The semantics described here is not entirely up to date.

[4]This complexity arises because the semantics accurately captures the potential exception behavior of the operation. The `LIT16` instruction cannot raise an exception.

```
(update-nth
   *aamp.ram*
   (write_memory
      (makeaddr (nth *aamp.denvr* st)
                (logand 65535
                        (logext 32 (+ -2 (nth *aamp.tos* st)))))
      (gacc::rx 16
                (makeaddr (nth *aamp.cenvr* st)
                          (nth *aamp.pc* st))
                (nth *aamp.ram* st))
      (nth *aamp.ram* st))
   (update-nth
      *aamp.tos*
      (logand 65535
              (logext 32 (+ -2 (nth *aamp.tos* st))))
      (update-nth *aamp.pc*
                  (logand 65535
                          (logext 32 (+ 2 (nth *aamp.pc* st))))
                  st)))
```

Figure 1: Semantics of LIT16 instruction

imperative form disappears. The user attempting to reason formally about AAMP7 operations or programs is confronted with conjectures involving huge and complex terms.

3 Reverse Abstraction

While the AAMP7 model functions well in the role of a formal simulator, it is hopeless as a basis for reasoning about AAMP7 programs. Two remedies to this situation seemed possible:

1. rewrite the entire model in a more abstract style;
2. discover a way to preserve the existing model, but "retrofit" it with abstractions that are more amenable to formal analysis.

The first was infeasible under the current contract. The second required a new approach that we call *reverse abstraction.*

The idea of reverse abstraction is to take recurring low-level forms within a specification and to rewrite them into a more abstract and perspicuous form. For example: in the `OP-LIT16` definition in Figure 1, the following form appears three times:

```
(logand 65535 (logext 32 (+ k x)))
```

This is a standard locution in the AAMP7 specification for adding two 16-bit quantities. This expression is provably equivalent to the slightly simpler logical expression:

```
   (loghead 16 (+ k x)).
```

By applying a rewrite rule, we can always eliminate the more complex form in favor of this simpler form. However, this still leaves the specification in terms of the logical operation `LOGHEAD`. We would like to replace this form with something more intuitive.

We define the following function and rewrite rule:

```
(defun plus16 (k x)
  (declare (xargs :guard (and (integerp k)
                              (integerp x))))
  (loghead 16 (+ k x)))

(defthm plus16-abstractor
  (equal (loghead 16 (+ k x))
         (plus16 k x)))
```

The lemma proves immediately simply by opening up the definition of `PLUS16`. Having this rule around during a proof attempt ensures that terms of the form

```
   (loghead 16 (+ k x))
```

will be rewritten to corresponding terms of the form

```
   (plus16 (+ k x)).
```

It is necessary to disable the function `PLUS16`; otherwise, the rewriter will get into an infinite loop of opening the function, rewriting it into its more abstract form, opening it, etc.

This process is very stylized and can all be accomplished with a macro. We define a macro `DEFABSTRACTOR` to perform these steps. The relevant call then appears as:

```
(defabstractor plus16 (k x)
  (loghead 16 (+ k x)))
```

This encapsulates the definition of the `PLUS16` function, the definition of the rewrite rule, and the disabling of the function. (It would probably be good to disable the rewrite rule as well, since we want reverse abstraction to occur selectively. We will likely do that in the next version.)

This simple example illustrates the technique of reverse abstraction. It can be summarizes as follows:

1. identify common low-level forms in the specification;
2. define an “abstraction function” in terms of the low-level form;
3. rewrite the low-level form into the more abstract version;

4. disable the abstraction function to prevent looping.

The result is an automatic capability to replace a given form by a conceptually more abstract equivalent.

At times, various low-level forms can be rewritten to the same abstraction. For example, the AAMP macro sometimes emits

```
   (logand 65535 (add32 x k))
```

instead of

```
   (logand 65535 (+ k x)).
```

Adding the following rewrite rule establishes that the two forms are semantically equivalent and eliminates a second syntactically different form in favor of our preferred abstraction.

```
(defthm plus16-abstractor-2
  (equal (logand 65535 (add32 x k))
         (plus16 k x)))
```

Abstractions may be nested, i.e., defined in terms of other abstractions. For example, the abstraction function `NEXT-STACK-ADDRESS` is defined in terms of the previously introduced `PLUS16` function. The abstractor lemma for `NEXT-STACK-ADDRESS` won't apply until the abstractor for `PLUS16` has already done its work.

```
(defabstractor next-stack-address (st)
  (makeaddr (nth *aamp.denvr* st)
            (plus16 -2 (nth *aamp.tos* st))))
```

By introducing a series of abstractions, which are automatically applied, it is possible to replace a complicated and non-intuitive expression such as that in Figure 1 with a form that is much easier to understand.

After the abstractions are applied, it often becomes easier to spot useful simplifications. For example, multiple updates to the same state component can be consolidated. This requires proving rewrites in terms of the abstraction functions. For example, we can prove the following rewrite:

```
(defthm inc-pc-inc-pc
  (implies (and (st-p st)
                (unsigned-byte-p 16 (+ i j (pc st))))
           (equal (inc-pc i (inc-pc j st))
                  (inc-pc (+ i j) st))))
```

In particular, we proved the following lemma, where the right hand side of the rewrite was "automatically" generated via reverse abstraction from the expansion of `(OP-LIT16 ST)` given in Figure 1.[5] This lemma is trivial to prove since the left and right sides are essentially identical, once the abstraction functions open.

```
(defthm lit16-rewriter
  (implies (st-p st)
           (equal (op-lit16 st)
                  (write-to-ram (next-stack-address st)
                                (fetch-code-word (pc st)
                                                 (cenvr st)
                                                 (ram st))
                                (inc-tos -2 (inc-pc 2 st))))))
```

This lemma now provides an alternative semantics for the `LIT16` operation. Hopefully, this semantics is easier to deal with in a proof context for several reasons. We can conceptualize the proof at the level of the abstraction functions. By proving appropriate lemmas about our abstractions we can raise the level of proof to a more abstract plane. By selectively disabling the abstraction functions we can hide the morass of details generated by the AAMP macro during any proof attempt.

Moreover, because the same basic forms are used throughout the specification, a relatively small collection of well-chosen abstraction functions can provide enormous benefits. For example, the abstractions defined for the `LIT16` operation applied to most of the other operations in the AAMP7 model. Thus, the incremental effort in applying reverse abstraction to each subsequent operations was less and less.

4 Performance Issues

The use of macros allows the semantics of AAMP7 operations to be modeled very intuitively in an imperative style. However, since the underlying logic is applicative the expansion of these macro expressions is ultimately a nest of accesses and updates to a single-threaded object representing the processor state. For even a fairly simple operation such as `ADD`, the resulting expansion is truly daunting.

The expansion may also contain some surprising inefficiencies. This is because operational behavior necessary in an iterative context is often highly counterproductive in an applicative context. Consider a potential operation `(ADDI X Y)`, adding two literal values `X` and `Y` on a hypothetical stack-based machine.[6] Assuming an AAMP7-style reader macro, the semantics of this operation might be defined in an imperative style by the following definition.

[5]This was done by using the ACL proof-checker utility to manipulate the complex term in Figure 1. After identifying likely abstractions and introducing abstraction functions and rewrites, these were applied interactively in the proof checker until the term was massaged into a congenial form, namely the right hand side of the `LIT16-REWRITER` lemma shown below.

[6]The point is more easily made on a hypothetical machine than on the AAMP7, though it certainly applies there.

```
(defun op-addi (st)
  (reader
    '( (fetch-word x)
       (push x)
       (fetch-word y)
       (push y)
       (add)
     )))
```

As a specification artifact, this is consise and intuitive. But, if translated naively, it may actually lead to very inefficient execution.

Emulating this imperative machine directly in an applicative context, macro expansion might generate something like the semantic function.

```
(defun op-addi-2 (st)
  (let ((X (fetch-code-word st))
        (Y (fetch-code-word (increment-pc st))))
     (increment-pc 2
      (write-to-stack (+ (fetch-tos 0 st)
                         (fetch-tos 1 st)))
       (decrement-tos
        (write-to-stack Y
         (increment-tos
          (write-to-stack X
           (increment-tos st))))))))
```

Assuming that the various components of the state are disjoint, `OP-ADD-2` is clearly—and hopefully, provably—equivalent to:

```
(defun op-addi-3 (st)
  (write-to-stack (+ (fetch-code-word st)
                     (fetch-code-word (increment-pc st)))
   (increment-pc 2
    (increment-tos st)))).
```

This version likely executes somewhat more efficiently than `OP-ADD-2`. The transformation from `OP-ADD-2` to `OP-ADD-3` is easily accomplished with a series of obvious rewrite rules. However, if the system simulator directly executes the output of the reader macro—as happens with the AAMP7—there is no opportunity for this rewriting optimization to occur. The result is that the simulator performs multiple, and sometimes redundant, updates on selected elements of the state.

As an alternative, we can prove that the naive translation is equivalent to the more efficient, optimized version. After doing this for each of the available operations, we can replace the original simulator with one that runs the more efficient versions. This simulator is equivalent to the naive version, but considerably more efficient.[7]

5 Conclusions

We have demonstrated an approach to "retrofitting" an existing low-level specification with abstractions. On analogy with reverse engineering, we call this *reverse abstraction.* Reverse abstraction 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. This became apparent in our efforts to apply reverse abstraction to the Rockwell Collins AAMP7 formal processor specification.

Our reverse abstraction process is not ideal. It requires considerable low level effort and ingenuity. Perhaps a better solution would have been to replace the original model with a more abstract version. Reverse abstraction might be viewed as a first step to identifying what such a model might look like.

This effort validates the importance 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. It is unlikely that there are many large formal specifications that will require reverse abstraction. Still, it could prove to be a valuable addition to the specifier's toolbox.

References

[1] R. Boyer and J Moore. Single-threaded objects in ACL2. In *Proceedings of Practical Aspects of Declarative Langauges, 2002*, pages 9–27, 2002.

[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.

[4] William D. Young and William R. Bevier. Developing an abstract separation kernel via successive refinement. Technical report 109, Computational Logic, Inc., May 1995.

[7]The insertion of abstraction functions in place of primitive update and accessor functions may introduce some function call overhead. However, most of our abstraction functions are simple, non-recursive functions and could be replaced by in-line code.