Axiomatic Language and Proof

Walter W. Wilson
wwwilson@acm.org
(retired)

ABSTRACT

A logic programming specification language called "axiomatic language" is described. [http://www.axiomaticlanguage.org] This minimal and pure language is well-suited to proof. A method is given for representing and proving assertions about specifications in the language.

CCS CONCEPTS

• **Software and its engineering → Software verification**.

KEYWORDS

formal verification, specification language, proof, logic programming

ACM Reference Format:
Walter W. Wilson. 2022. Axiomatic Language and Proof. In *Proceedings of FormaliSE: International Conference on Formal Methods in Software Engineering (FormaliSE 2022).* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

This paper describes a type of logic programming called "axiomatic language" [2] [3] [http://www.axiomaticlanguage.org]. Axiomatic language is a pure specification language so its implementation requires automatically transforming specifications to equivalent efficient programs – a grand challenge of computer science. Proof is needed to guarantee correctness of this transformation and to prove properties that help validate a specification. Section 2 defines the language and section 3 describes a system of proof. Section 4 gives some final comments.

2 AXIOMATIC LANGUAGE

Axiomatic language has the following goals: (1) pure specification, (2) extreme minimality, and (3) metalanguage extensibility. It is based on the idea that the external behavior of a program can be represented by a static, infinite set of symbolic expressions which enumerate program inputs along with the corresponding outputs. The language is just a formal system for defining these symbolic expressions. [http://csl.stanford.edu/~christos/pldi2010.fit/wilson.specio.pdf]

FormaliSE 2022, May 22–23, 2022, Pittsburgh, PA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2.1 The Core Language

In axiomatic language a finite set of axioms generates a (usually) infinite set of valid expressions. An **expression** is

an **atom** – a primitive, indivisible element,
an **expression variable**,
or a **sequence** of zero or more expressions and **string variables**.

Syntactically, atoms are represented by symbols that begin with a backquote: `` `abc ``, `` `+ ``. Expression and string variables begin with `%` and `$`, respectively. Sequences have their elements separated by blanks and enclosed in parentheses: `` (`M () (% $1)) ``.

An **axiom** consists of a **conclusion** expression and zero or more **condition** expressions:

```
<conclu> < <cond1>, ..., <condn>.
<conclu>.              ! an unconditional axiom
```

Comments start with an exclamation point.

Axioms generate **axiom instances** by the substitution of values for the expression and string variables. An expression variable can be replaced by an arbitrary expression, the same value replacing the same variable throughout the axiom. A string variable can be replaced by a string of zero or more expressions and string variables. For example, the axiom

```
(`A %x $w)< (`B ($ %y %x)), (`C $w).
```

has an instance

```
(`A `x `u `v)< (`B (() `x)), (`C `u `v).
```

by the substitution of `` `x `` for `%x`, `()` for `%y`, the string `` `u `v `` for `$w`, and the null string for `$`.

Axiom instances generate **valid expressions** by the rule that if all the conditions of an axiom instance are valid expressions, the conclusion is a valid expression. By default, the conclusion of an "unconditional" axiom instance is immediately a valid expression. For example, the two axioms

```
(`a `b).
((%) $ $)< (% $).
```

generate the valid expressions `` (`a `b) ``, `` ((`a) `b `b) ``, `` (((`a)) `b `b `b `b) ``, ….

2.2 Syntax Extensions

The expressiveness of axiomatic language is enhanced with some syntax extensions. A single character in single quotes is equivalent to writing an expression that gives the binary code of the character using bit atoms:

```
'A'  ==  (`char (`0 `1 `0 `0  `0 `0 `0 `1))
```

A character string in single quotes within a sequence is equivalent to writing the characters separately:

```
(... 'abc' ...)  ==  (... 'a' 'b' 'c' ...)
```

A character string in double quotes represents the sequence of those characters:

```
"abc"  ==  ('abc')  ==  ('a' 'b' 'c')
```

A symbol that does not begin with ` % $ ( ) ' " is syntactic shorthand for an expression that gives the symbol as a character string,

```
ABC  ==  (` "ABC")
```

and uses the atom represented by just the backquote.

2.3 Examples

Here are axioms for natural numbers in successor notation and the plus operator:

```
(n 0).                  ! n0: zero is a natural number
(n (s %n))< (num %n).   ! n1: successor of num is num

(p % 0 %)< (n %).       ! p0: n + 0 = n
(p %1 (s %2) (s %3))<   ! p1: n1 + (n2+1) = (n3+1)
  (p %1 %2 %3).         !   if n1 + n2 = n3
```

These axioms generate valid expressions such as `(n (s (s 0)))` and `(p (s (s 0)) (s 0) (s (s (s 0))))`, representing the statements "2 is a natural number" and "2 + 1 = 3", respectively.

3 PROOF IN AXIOMATIC LANGUAGE

This section proposes a system of proof for axiomatic language. Consider the following candidate axiom:

```
(n (s (s %)))< (n %).  ! n2: 2+n is num if n is num
```

If added to the above natural number axioms n0,n1, no new natural number valid expressions are generated.

3.1 Valid Clauses

Some definitions are needed. A **clause** is defined the same as an axiom – a conclusion and zero or more conditions. (Axioms are just specially designated clauses.) Assigning values to the clause variables gives a **clause instance**. If all the conditions of a clause instance are valid expressions for a set of axioms, then the conclusion is a **generated expression**. A clause is a **valid clause** with respect to a set of axioms if all its generated expressions are valid expressions for those axioms. Thus, adding a valid clause to a set of axioms does not add to the set of valid expressions. Clause n2 above is thus a valid clause with respect to the natural number axioms. One can say that a valid clause is "implied" by the set of axioms – it is "redundant" for those axioms. It can be considered a "true statement" about the axioms.

3.2 Proving Valid Clauses

Given a set of axioms, the following rules can be used to derive valid clauses:

R1. An axiom is a valid clause.

R2. An instance of a valid clause is a valid clause.

R3. Adding a condition to a valid clause gives a valid clause.

R4. For any set of axioms we have this tautological valid clause:

```
% < % .
```

R5. For every valid expression ve we have this valid clause:

```
ve.
```

R6. If no instance of expression nve is a valid expression, its occurrence as a condition gives a valid clause:

```
% < nve.
```

R7. Given valid clauses A and B,

```
A: cA0 < cA1,...,cAna.
B: cB0 < cB1,...,cBnb.
```

where cA0,cB0 are conclusions and cA1..cAna,cB1..cBnb are conditions, if some condition cAk is identical to conclusion cB0, then we can construct valid clause C from clause A where condition cAk is replaced by conditions cB1..cBnb of clause B:

```
C: cA0 < cA1,..,cAk-1,cB1,..,cBnb,cAk+1,..,cAna.
```

We call this an unfold of valid clause A condition k with valid clause B.

Using the above rules we can show that clause n2 is valid:

```
a: (n (s %))< (n %).          R1 - axiom n1
b: (n (s (s %)))< (n (s %)).  R2 - instance of a
n2: (n (s (s %)))< (n %).     R7 - unfold b with a
```

R8 - Induction Rule. Valid expressions can be assigned "level numbers" based on the number of steps needed to generate the expression from the set of axioms. An induction proof of clause C,

```
C: c0 < c1,..,ci,..,cn.
```

means showing that C is a valid clause for all levels of valid expressions that match condition ci. A complete unfold of ci with respect to the set of axioms means finding all the most-general unifications between ci and axiom conclusions. (That is, we find the most-general instances of C and each axiom that make ci and the axiom conclusion identical.) Each successful unification j gives a clause Cj where the instantiated axiom conditions a1',..,am' replace ci in the instantiated clause C:

```
complete unfold of C condition ci:
  ...
  Cj: c0' < c1',..,ci-1',a1',..,am',ci+1',..,cn'.
  ...
```

These Cj clauses represent all the ways that C can generate expressions. Proving the Cj clauses valid proves clause C valid. Some of the Cj clauses may have conditions that match the same valid expressions as ci, but at a lower level number. We can treat C as an induction hypothesis and allow a valid clause to be unfolded against it (by rule R7) to prove a Cj clause. Note that some Cj clauses represent induction base cases while others represent an induction step.

3.3 Example Proofs

Here is an alternative set of axioms for the natural number plus operator, with the induction on the first argument:

```
(p 0 % %)< (n %).       ! pa0: 0 + n = n
(p (s %1) %2 (s %3))<   ! pa1: (n1+1) + n2 = (n3+1)
  (p %1 %2 %3).         !   if n1 + n2 = n3
```

We will refer to pa0,pa1 as PA axioms and p0,p1 as P axioms. Note that the P and PA axioms generate the same set of addition valid expressions. Given axiom sets A and B, if all the A axioms are valid clauses with respect to axiom set B and, conversely, all the B axioms are valid clauses with respect to A, then A and B are **equivalent axiom sets**.

We prove that PA axioms pa0,pa1 are valid clauses with respect to axiom set p0,p1,n0,n1, as follows:

```
some valid clauses from n0,n1,p0,p1 using R1-R7:
  p00: (p 0 0 0).           unfold of p0 with n0
  p0x: (p (s %) 0 (s %))< (n (s %)).  instance of p0
  p0y: (p (s %) 0 (s %))< (n %).    unfold of p0x w n1
  p1x: (p 0 (s %) (s %))< (p 0 % %).  instance of p1
 p1y: (p (s %1) (s %2) (s (s %3)))<(p (s %1) %2 (s %3)).
     - instance of p1

prove pa0: (p 0 % %)< (n %).
  R8 - complete unfold of pa0 cond 1 wrt ax set:
  pa00: (p 0 0 0).      = p00
  pa01: (p 0 (s %) (s %))< (n %).
    = unfold of p1x with induction hypothesis pa0
- since all Cj clauses are proved, pa0 is proved

prove pa1: (p (s %1) %2 (s %3))< (p %1 %2 %3).
  R8 - complete unfold of pa1 cond 1 wrt ax set:
  pa10: (p (s %) 0 (s %))< (n %).  = p0y
  pa11: (p (s %1) (s %2) (s (s %3)))< (p %1 %2 %3).
    = unfold of p1y w induc hypoth pa1
- all Cj clauses proved, so pa1 is proved
```

Similar proofs show that P axioms are valid clauses with respect to axiom set n0,n1,pa0,pa1. Thus, axiom sets P and PA are equivalent – they generate the same set of valid expressions.

Let us add to our set of axioms the following axiom that asserts that two expressions are identical:

```
  (= % %).          ! =: identical expressions
```

We use this in the following clause that asserts that our addition definition is commutative:

```
  C: (= %12 %21)< (p %1 %2 %12), (p %2 %1 %21).
```

For this clause to be valid, the addition results must always be the same when argument order is reversed. We need to first prove that the following supporting clause is valid:

```
  =s: (= (s %a) (s %b))< (= %a %b).
  proof - unfold cond 1 wrt ax set n0,n1,p0,p1,=:
    =s': (= (s %) (s %)).   = instance of =
```

The proof of the commutativity clause C is as follows:

```
  C: (= %12 %21)< (p %1 %2 %12), (p %2 %1 %21).
  unfold C cond 1 wrt ax set n0,n1,p0,p1,=:
    C0: (= % %21)< (n %), (p 0 % %21).
    unfold C0 cond 2 wrt equiv ax set n0,n1,pa0,pa1,=:
      C00: (= % %)< (n %), (n %).
        = axiom = with added conditions
   C1: (= (s %12) %21)< (p %1 %2 %12), (p (s %2) %1 %21).
    unfold C1 cond 2 wrt equiv ax set n0,n1,pa0,pa1,=:
    C11: (= (s %12) (s %21))<(p %1 %2 %12),(p %2 %1 %21).
        = unfold of =s w induc hypoth C
```

More example proofs can be found at http://axiomaticlanguage.org/proof.htm.

4 FINAL COMMENTS

Axiomatic language is an elegant and potentially powerful formal system for software specification. It provides an idealistic separation between specification and implementation. The language is completely pure – no built-in functions, no input/output operations, and no procedural semantics. Its metalanguage extensibility enables the definition and use of other language features for specification within axiomatic language. Axiomatic language should be a good host for embedded DSLs. [1] Future work will address the grand challenge of axiomatic language implementation: http://axiomaticlanguage.org/BabySteps.pdf

The concept of valid clauses provides a straightforward representation for assertions about a specification. The proof system is "close" to the language, which should make proof a more understandable and usable tool for users of the language. A near-term goal is to define a formal representation for proofs and a program to check those proofs.

REFERENCES

[1] M. P. Ward. 1995. Language Oriented Programming. *Software—Concepts and Tools* 15 (1995), 147–161.

[2] Walter W. Wilson. 1982. Beyond PROLOG: Software Specification by Grammar. *SIGPLAN Not.* 17, 9 (Sept. 1982), 34–43. https://doi.org/10.1145/947955.947959

[3] Walter W. Wilson and Yu Lei. 2012. A Tiny Specification Metalanguage. In *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE'2012), Hotel Sofitel, Redwood City, San Francisco Bay, USA July 1-3, 2012.* Knowledge Systems Institute Graduate School, 486–490. http://ksiresearchorg.ipage.com/seke/Proceedings/seke/SEKE2012_Proceedings.pdf