HALINA PRZYMUSIŃSKA

LOGIC PROGRAMMING TOOLS FOR FORMAL SOFTWARE SPECIFICATION

STRESZCZENIE

Jednym z najistotniejszych problemów w inżynierii oprogramowania jest opracowanie i zawężenie wymagań dotyczących oprogramowania w celu określenia dokładnych specyfikacji, a także kwestia przekształcenia specyfikacji dotyczących złożonych problemów w wykonywalne kody. Głównym powodem podjęcia się badania formalnych specyfikacji jest udowodnienie, że oprogramowanie produkowane jest z nimi zgodne (chodzi o weryfikowalność). Artykuł ukazuje, iż programowanie logiczne połączone z rachunkiem sytuacyjnym można z powodzeniem stosować jako formalną specyfikację oprogramowania.

INTRODUCTION

Two of the most important problems in software engineering are the problem of elaborating and refining software requirements into accurate specifications, and the problem of transforming specifications of complicated problems into executable code. Proving that software products produced agree with the specifications (verifiability) is the main reason for pursuing the study of formal specifications. Good formal specifications should also have the following properties:

- it should be relatively easy to test whether formal specifications correspond to the informal ones, and
- it should be easy to modify specifications and therefore the corresponding codes according to additional or modified requirements.

In current software engineering practice many formal specification methods are being used. Most of the specification languages used such as for example Z language have a robust mathematical symbolism available to them, but lack an appropriate procedural component, as we explain below.

In [Kow87], R. Kowalski introduced logic programming as a tool which adds to a specification language an inference mechanism making it possible to execute specifications, for the sake of rapid prototyping.

For our approach to logic specification it is essential that a precise meaning or semantics be associated with the programs considered. The developments in logic programming (see, for example [Prz 81], [Prz 90]) provide logic programs with very natural and intuitive declarative semantics. We argue here that logic programming coupled with situation calculus can be successfully used for formal software specification. Logic provides a rigorous specification of meaning which is context-free and easy to manipulate, exchange and reason about. It therefore seems to be a natural choice as a software specification language, or *software specification formalism*.

To illustrate the proposed approach let us consider a formal specification for a storage allocation system. The (basic) requirements that the specification was supposed to satisfy are the following:

- The storage allocator system models users and blocks of storage. Users request a block and the system allocates it, if the block is currently free. Users may also release a block which subsequently becomes free, provided that the block was actually owned by the user.
- Figure 1 displays some of a Z specification for a storage allocator, which is similar to that presented in [Woo89].

RequestBlock

SM
u? : U
b! : N
r! : Report

(free <> {}
b! member free
free' = free \ {b!}
dir' = dir U {b! |-> u?}
r! = okay)
v
(free{}
r! = fail
free = free'
dir = dir')

Figure 1. Z language block request specification

We propose that a logic programming specification be given in two components. The first component, which we characterize here as the *semantic component,* is a collection of logical formulas specifying the system requirements using what is known in the literature as *situation calculus*. It is rather straightforward to translate the English requirements statement above into the language of situation calculus. A listing is given in Figure 2.

Situation calculus is a logic-based formalism which allows one to express the changes in the properties of objects as a result of the actions that take place. The language of situation calculus has predicates 'holds' and 'affects', a function symbol 'result', together with variables and constants for relations, actions and situations. Intuitively, 'holds(P, S)' means that property P holds in situation S, 'affects(A,P,S)' means that action A taking place in situation S affects property P, and 'result(A, S)' is the situation that results when action A is performed in situation S.

```
holds(owns(U,B),result(allocate(B,U), S)) <-
        holds(free(B), S).
affects(free(B), allocate(B,U), S).
holds(free(B), result(retrieve(U,B), S)) <-
        holds(owns(U,B), S).
affects(owns(U,B),retrieve(U,B), S).
 ;;  inertia/frame axiom
holds(Property,result(Action, S)) <-
        holds(Property, S),
        ~ affect(Action,Property, S).
```

Figure 2. Semantic specification

The essential feature of the situation calculus is a logical characterization of which properties hold as the result of performing certain actions, and which properties are affected by the actions. The main intention of the semantic component is to capture the logical properties of the specification in a highly declarative form. We have written the logical clauses in the semantic component in the form of Horn clauses, and we intend that the negation in the inertia axiom be treated

as negation as failure to be true (when the formal semantics is specified). It is explicitly intended that appropriate formal semantics be associated with this semantic component, say, for example, well-founded semantics [Gel 91], but there is no requirement that the semantic component itself be an executable logic program. It is important to realize that the semantic component allows us to reason about the effects of different actions, but no actual action takes place and no explicit input data needs to be provided with the specification.

For example, the provable assertion

$$\{holds(free(b_1), s_0), holds(free(b_2), s_0), holds(free(b_3), s_0), \ldots\} \models$$
$$holds(owns(u_1, b_1), result(allocate(u_2, b_2), result(allocate(u_1, b_3), result(allocate(u_1, b_1), s_0))))$$

follows logically in the semantic component. Here ' |= ' is the usual logical consequence operator. A rough translation of this logical assertion is that once user u_1 has been allocated block b_1 (which requires that it was free previously), the subsequent actions of allocation of blocks b_2 and b_3 to users u_2 and u_3 respectively do not disturb the property of u_1 owning b_1.

It is highly likely that realistic (and more complex) specifications could require more expressive power than that available currently in different extensions of logic programs.

The other component of the logic specification is what we call the *procedural component*. This component is intended to be an executable logic program. The procedural component serves as an executable prototype for the intended software. A listing of the Prolog program for this component is given in Figure 3.

```
;; storage.pro
;; free blocks and directory are asserted and retracted.
allocate(U,B) : -
                free(B),                    ;;  find free
                retract(free(B)),           ;; no longer free
                assertz(owns(U,B)).         ;; now allocated
retrieve(U,B) :-
                owns(U,B),                  ;;  find owned
                retract(owns(U,B)),         ;;  no longer owned
                assertz(free(B)).           ;;  now free
execute(request(U,B)) :-
                 allocate(U,B),
                 report(”okay”).
execute(request(U,B)) :-  report(”errNotFre”).
execute(release(U,B)) :-
                 retrieve(U,B),
                 report(”okay”).
execute(release(U,B)) :-
                 free(B),
                 report( “errBlockFree” ).
execute(release(U,B)) : -
                 report(” errNotOwner”).
;;  show allocation dump
execute(dump) :- dump.
;; and default
execute(Nonsense) :-
                 report (“errNonsense”).
```

```
report(Message) :- display(“ “), display(Message), nl.
dump :- owns(U,B), write(owns(U,B)), display(“ “), fail.
dump :- nl.
;;  the driver for the executable prototype
operate :-
                nl,
                task(Task),
                execute(Task),
                operate.
task(Task) :-
               display(”Task?> “),
               read(Task),
               nl.
;; sample initialization
free(b1).   free(b4).
free(b2).   free(b5).
free(b3).
```

Figure 3. Procedural component

The Prolog program obtained as a result of translating the semantic component into a procedural one consists of two parts, one of which is a pure logic program (the first two Prolog clauses) and the second of which has the form of a loop monitoring and executing tasks, or providing initial conditions for a prototype (the other clauses). Execution of some tasks (such as a “request”) may change the declarative part while execution of others (such as “dump”) require only a query of the declarative part. The first kind of task corresponds to *primary tasks* which need to be characterized in the semantic component, whereas the other tasks correspond to *derived tasks*, which we currently believe do not need to be characterized in the semantic component. Likewise, the procedural component could characterize derived properties that were not specified in the semantic component, but the procedural component must characterize primary properties of the semantic component. (We did not illustrate this for the storage allocator.) Also, the correspondence between task names and action names is informal here. The basic formal correspondence between the semantic component and the procedural component involves a translation schema which might be pictured in simple cases as in Figure 4.

```
;;  the group of related situation clauses
holds(prop1 ,result(action, S)) <-
      holds(prop2, S).
holds(prop3,result(action, S)) <-
      holds(prop2, S).
holds(prop4,result(action, S)) <-
                holds(prop3 , S).
affects(prop5,action, S) <- holds(prop2).
affects(prop6,action, S) <- holds(prop3).
;; is translated into Prolog clauses
action :- prop2,
          retract(prop5),
          assertz(prop 1),
            assertz(prop3).
```

```
action :- prop3,
          retract(prop6),
          assertz(prop4).
```

Figure 4. Translation

Note that the "retracts" are done before the "asserts" in the Prolog clause. Figure 4 is intended to illustrate some patterns for translation. It is not intended to be a complete translator for all possible semantic specifications. For the storage allocator example, only one Prolog "action" clause was generated for each of "allocate" and "retrieve" actions.

Figure 5 gives an indication of what the customer might see when the prototype is executed. The underlined parts are typed in by the "users".

```
?- operate.
Task? > request(u1,B).
  okay
Task? > request(u5,b3).
  okay
Task? > dump.
 owns(u1,b1)  owns(u5,b5)
```

Figure 5. Sample prototype execution

One problem with this approach is that parts of the Prolog program change due to the changes of situations as represented by the collection of facts known to be true at the given moment. Instead of a single logic program, we are dealing with a sequence of such programs which are obtained by asserts and retracts. Asserts and retracts have no (direct) semantics associated with them. In the semantic component, the situation calculus is used to characterize the type of nonmonotonic reasoning in which changes of the world over time are crucial. (See [Ni180] chapter 7, for example.) Situation calculus allows us to formally describe the changes in a logic program that result from asserts and retracts, and thus provides the semantics for these operations -- and hence the name *semantic component*.

We require that the two above mentioned components be compatible in the following way.

SOUNDNESS

Suppose that P_o is an initial program, that $a_1,a_2,\ldots a_n$ is a sequence of actions corresponding to primary tasks, and that $P_o,P_1,\ldots,P_n$ is the sequence of programs that result from executing the sequence of primary tasks (by means of various asserts and retracts). Let s_o be an initial situation constant, and let

$s_i = result(a_i, s_{i-1})$ for $i = 1,\ldots,n$, be the corresponding sequence of situations in the semantic component.

If the Prolog goal
?- prop(c).
succeeds for program P_n where prop is a primary property, then
$\{holds(--,s_o), \ldots\} \models holds(prop(c), s_n)$

is a logical consequence in the semantic component, where the precondition set corresponds to the initial conditions given in P_o, and c is an appropriate constant argument.

COMPLETENESS

Suppose that s_o is a situation for the semantic component in which a finite number of initial properties p_k are hypothesized to hold

holds($p_k(c_k)$, s_o), where the c_i's are constants, and suppose that P_o is the corresponding procedural component having a subprogram consisting of all the unit clauses $p_k(c_k)$. Suppose further that $s_o,s_1,\ldots,s_n$ is a sequence of situations in the semantic component such that $s_i = result(a_i,s_{i-1})$ for i = 1,. . . ,n where each a_i is an action. Let $P_o,P_1,\ldots,P_n$ be the corresponding sequence of Prolog programs that result from executing the tasks corresponding to $a_1,\ldots,a_n$.

If holds(prop(c), s_n) is a semantic consequence of the semantic component, including the preconditions, then the goal

?- prop(c).

succeeds for program P_n .

We are working on refining requirements for the two components so that compatibility would be provably true. Note well that there would then be no need to explicitly work with „two" components if one could trust that the procedural component specified the semantic component faithfully and completely. It seems clear that the procedural component would be the preferable version of a specification to develop since it also serves as a prototype. In that ideal case, one would truly have an „executable specification".

We see the following advantages to the logic programming approach.

- The use of logic makes both components of the specification highly declarative.
- Formal mathematical and logical proof methods can be applied to the semantic component.
- Logic programming theory applies to the procedural component.
- The procedural component can serve as an executable prototype to be used with the customer as a tool for further refining the specification. Behaviors that are wrong may be observed by executing the prototype. Subtleties of the requirements can be viewed in the context of the working prototype.
- There is the long range possibility that portions of the procedural component of the specification can itself be refined into the target software. That is, the specification may become part of the source code for the intended software product, thus providing the possibility that some of the target software is its own specification.

REFERENCES

[Fis 91] Fisher J., Lee C., *Software engineering education, logic programming, and A.I. tools*, in: *Proc. 2nd Annual Al Symposium for the California State University*, California Polytechnic State University, San Luis Obispo, June 1991

[Kow 87] Kowalski R., *Logic Programming in Artificial Intelligence and Software Engineering*, in: *Intelligent Knowledge-Based Systems*, (eds.) O'Shea T., Self J., Thomas G., Harper and Row, 1987

[Nil 80] Nilsson N., *Principles of Artificial Intelligence*, Tioga, 1980

[Prz 88] Przymusinski T., *Non-monotonic reasoning vs. logic programming: a new perspective*, in: *Handbook of Formal Foundations of A.I.*, (eds.) Lilks Y., Partridge D., to appear. Abstract in Proc. AAAI'88

[Prz 90] Przymusinska H., and Przymusinski, T., *Semantic issues in deductive databases and logic programs*, in: *Formal Techniques in Art4ficial Intelligence, A Sourcebook*, (ed.) Banerji R.B., Elsevier, 1990

[Gel 91] Gelder van A., Ross A., Schlipf J.S., *The well-founded semantics for general logic programs*, „Journal of the ACM" 1991, Vol. 38, No. 3, pages 620-650

[Woo89] Woodcock J., *Software Engineering Mathematics*, Addison-Wesley, 1989