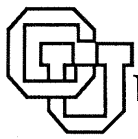
A Process Program for Gries/Dijkstra Design

Robert B. Terwilliger

CU-CS-566-91 December 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Process Program for Gries/Dijkstra Design

Robert B. Terwilliger

CU-CS-566-91 December 1991

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

(303) 492-7514
(303) 492-2844 Fax

A Process Program for Gries/Dijkstra Design

Robert B. Terwilliger

Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
email: 'terwilli@cs.colorado.edu'

ABSTRACT

It has been suggested that *process programming* can improve the effectiveness of software development. The basic idea is simple: describe development processes using programming language constructs. In this paper we present a process program for the formal design process developed by Dijkstra and Gries. This method takes as input a pre- and post-condition specification written in predicate logic, and through a sequence of steps transforms it into an algorithm written using guarded commands. Our process program uses a library of *cliches* describing solutions to common programming problems. We have constructed a prototype implementation written in Prolog and used it to generate a design for Kemmerer's Library Problem.

1. Introduction

Despite years of effort by both researchers and practitioners the production of software remains both difficult and expensive [8]; software developments remain notoriously difficult to manage and are routinely completed late and/or over budget [11]. There are many opinions as to the essential (or accidental) causes of this problem, and many suggestions for its solution [3]. One interesting way that solutions strategies may be arranged is along the spectrum from *product centered* to *process centered.*

A commercial software release contains many different products emeshed in a web of extremely complex relationships. The processes used to produce software systems are also complex, as they must produce and maintain components in the correct relationships. A product centered strategy focuses attention on the products produced by the software process, while a process centered strategy focuses on the process itself. Currently, there is significant interest in process centered strategies [1, 12, 13].

One approach being pursued is *process programming* [16, 17, 23]. The initial idea is simple: describe software processes using programming language constructs and notations. Ultimately, this should allow software development to become automated; unfortunately, an engineer wishing to automate software development is met with an immediate obstacle: typically, the processes being used are not well defined or understood. Fortunately, process programming can also enhance process understanding. Describing techniques in enough detail to even approach automation should dramatically increase comprehension and possibly even improve the processes themselves.

In this paper, we present a process program for the formal design method developed by Dijkstra and Gries [6, 7, 10]. This method takes a pre- and post-condition specification written in first-order predicate logic and produces an algorithm design written in guarded commands. The process uses a library of *cliches* describing solutions to common programming problems. The process consists of a sequence of steps, each of which applies a pre-verified cliche to the current partial design. Since each cliche only generates correct transformations, the final design satisfies the original specification.

The undertaking just described is somewhat similar to work performed in the artificial intelligence community under the title automatic programming [2, 9, 18, 20-22]. According to [19], an ideal automatic programming system would be general purpose, completely automatic, and end user oriented. While this is unrealistic, there are a number of workable approaches. For example, a very high-level language sacrifices end-user orientation, but is general purpose and completely automatic. The process program described in this paper can be thought of as a translator for a very high-level language: the specifications extend conventional programming languages with quantifiers and high-level data types. The process is also non-interactive: design derivation proceeds with no intervention from the programmer.

In the remainder of this paper, we describe this process program in detail. In section two we give some background on the Gries/Dijkstra design method, and in section three we present our process program and argue for its correctness. In section four we present some cliches used by the process, and in section five

This research was supported by a grant from AT&T, as well as NSF Grant CCR-8809418

we show how the program generates a design for Kemmerer's Library Problem from a formal specification. In section six we briefly discuss a prototype implementation of this process written in Prolog, and finally, in section seven we summarize and draw some conclusions from our experience.

2. Gries/Dijkstra Design

The process we are considering was developed by Dijkstra and Gries [6, 7, 10]. The method is in some sense general, but is most applicable to problems in algorithm design. The Gries/Dijkstra process is an example of the general class of methods described by Figure 1. Processes of these types have two levels. At the lower level, a derivation process transforms a problem specification into a solution using a library of cliches that represent solutions to common problems. Since the correctness of the final solution depends on the correctness of the cliches used in its derivation, the upper level uses verification rules (either formal or informal) to certify that the cliches in the library are correct.

In the Gries/Dijkstra process, the problem specifications are pre- and post-condition specifications written using first-order predicate logic, the verification rules are formal proof rules for programming constructs, the cliches are transformations from specifications to partial programs, and the correct solutions are programs that are totally correct with respect to their specifications. For example, the following specification is for a code fragment to compute the sum of the elements of an array.

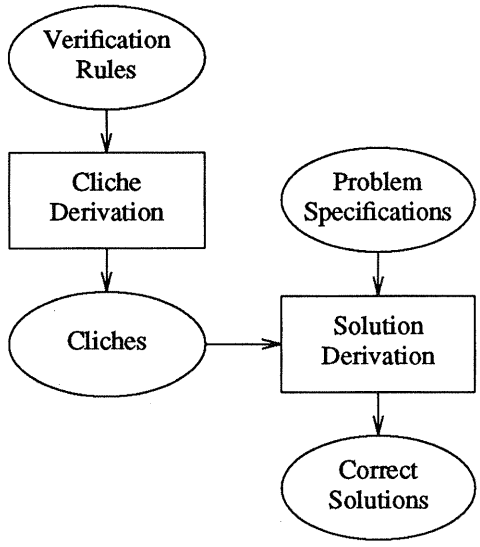


Figure 1. Cliche Driven Development Process

```
var b : array[0..n-1] of integer ;
var s : integer ;
{Q: true}
< S >(s:out integer) ;
{R: s = (Σk:0≤k<n:b[k])}
```

The post-condition "R" states that s is equal to the sum of the array elements from zero through "n-1".

Using the Gries/Dijkstra method, design might proceed as follows. First, we notice that the summation operator is not built into our programming language; therefore, we must use a loop to iterate over the array. We will construct this loop and formally verify its correctness simultaneously. We will specify the loop using a predicate called the *invariant*, which must be true both before and after each iteration of the loop, and an integer function called the *bound*, which must remain greater than or equal to zero, but be decreased by each execution of the loop body.

We can develop an invariant by *weakening* the post-condition; in other words, the invariant is an easier to satisfy version of the post-condition. There are at least three ways to weaken a post-condition: delete a conjunct, replace a constant by a variable, and enlarge the range of a variable. In this case, we can replace the constant "n" with the variable "j" to obtain the following invariant.

```
{inv P: 0≤j≤n ∧ s=(Σk:0≤k<j:b[k])}
```

It can be initialized with the simultaneous assignment "s,j:=0,0". Since we replaced a constant by a variable to obtain the invariant from the post-condition, the loop guard is just that the variable is not equal to the constant ("j≠n"), and the bound function is just the difference between the constant and the variable ("n-j"). We therefore have the following design.

```
var j : integer ;
{Q: true}
s,j:=0,0 ;
{inv P: 0≤j≤n ∧ s=(Σk:0≤k<j:b[k])}
{bnd t: n-j}
do j≠n → < S > od
{R: s = (Σk:0≤k<n:b[k])}
```

To complete the design, we must construct a body for the loop that maintains the invariant and

decreases the bound. We can decrease the bound with the statement "j:=j+1", but to maintain the invariant we must also adjust the value of "s". This can be accomplished with the simultaneous assignment "s,j:=s+b[j],j+1". The combination of the loop invariant ("0≤j≤n) and guard (j≠n) guarantees that "b[j]" will always have a valid value.

We have now produced the following design, which can be proven correct using standard rules.

```
{Q: true}
s,j:=0,0 ;
{inv P: 0≤j≤n ∧ s=(Σk:0≤k<j:b[k])}
{bnd t: n-j}
do j≠n → s,j:=s+b[j],j+1 od
{R: s = (Σk:0≤k<n:b[k])}
```

We believe that the Gries/Dijkstra process is a good subject for process programming. It is formal and reasonably well defined; therefore, it should be reasonably easy to either construct a program for the process, or determine why it can not be done. We have undertaken this exercise and determined that a process program can be written. We will now describe that program in some detail.

3. Process Program

In this section, we present the design for our process program using a guarded command style notation. First, we describe the basic types used in the process, then we present some routines that support the program, and then finally we present the two main routines: "derive_design" and "derive_subs". We then argue for correctness of the process assuming the correctness of the cliches it uses. While our argument does not constitute a fully formal proof, it does significantly increase our confidence in the correctness of the design.

Figure 2 shows the declarations for the basic types used in the design process. The most fundamental is the design ("dsgn") record. Each "dsgn" contains a pre- and post-condition, both of which are boolean expressions; a symbol table ("st"), defining the context in which the design is to be interpreted; and an abstract syntax tree for the command. The structure of the tree depends on the statement represented. For example, the record representing an assignment statement has a "cmd" field of "asgn", and its variant record part includes a list of the expressions to be computed and the variables they are to be assigned to.

The record representing a loop has a "cmd" field of "do" and a field for each sub-component of the loop. The "init" field is a design for the loop's initialization, while the "inv" field holds the predicate that is the loop's invariant. The "bnd" component is an integer

```
type dsgn   =   record
                pre,post : bool_expr;
                st       : symtab   ;
                case cmd of
                asgn: ( v : seq(sym)      ;
                        e : seq(expr) )   ;
                if  : ( cmds : seq(gcmd));
                do  : ( init : dsgn       ;
                        inv  : pred       ;
                        bnd  : int_func   ;
                        grd  : bool_expr  ;
                        dec  : dsgn       ;
                        body : dsgn     ) ;
                seq : ( s1,s2: dsgn     ) ;
                skip, undef  : (        ) ;
                end dsgn ;

type spec   =   dsgn where
                s:spec => s.cmd=undef ;

type gcmd   =   record
                grd : bool_expr ;
                cmd : dsgn      ;
                end gcmd ;

type cliche =   record
                function
                    pre(s:spec) : boolean;
                function
                    apply(s:spec) : dsgn ;
                end cliche
where c:cliche,s:spec =>
    (c.pre(s) => refines(s,c.apply(s)));

var  all_cliches : seq(cliche) ;
```

Figure 2. Design Process Type Definitions

function that serves as the loop's bound function, while the "grd" field holds the boolean expression that is the loop guard. The "dec" component is a design for code that decreases the bound, while "body" is a design for the remainder of the loop body.

The record representing an if statment has a "cmd" field of "if" and holds a list of guarded commands. Each "gcmd" record includes both a boolean expression (the guard) and a design (the command). The record representing a null statement has a "cmd" field of "skip" and its variant part is empty. A design record with a "cmd" field of "undef" represents a specification ("spec"): pre- and post-conditions with no statement in between.

The records representing cliches consist of two functions. "Pre" takes a specification as an argument and returns a boolean result, while "apply" takes a specification and returns a design. "Pre" evaluates to true if the cliche is applicable to the specification in question, while "apply" returns the result of applying the cliche in the proper manner. Cliches are constrained to maintain the "refines" relation; specifically, for any cliche "c", if "c.pre(s)" evaluates to true, then the value returned by "c.apply(s)" refines "s". There is a global variable "all_cliches" that holds all the cliches currently known to the system.

```
function dsgneq(d1,d2:dsgn):boolean;
    post    dsgneq =
                ( d1.st=d2.st ∧
                  d1.pre=d2.pre ∧
                  d1.post=d2.post ∧
                  d1.cmd=d2.cmd ) ;

function refines(d1,d2:dsgn):boolean
where
    d,d1 : dsgn =>
    ( dsgneq(d,d1) ∧
        d.cmd∈{asgn,skip,undef} =>
            refines(d,d1) ) ∧
    ( dsgneq(d,d1) ∧ d.cmd∈{do} ∧
        refines(d.body,d1.body) =>
            refines(d,d1) ) ∧
    ( dsgneq(d,d1) ∧ d.cmd∈{seq} ∧
        refines(d.s1,d1.s1) ∧
        refines(d.s2,d1.s2) =>
            refines(d,d1) ) ∧
    ( dsgneq(d,d1) ∧ d.cmd∈{if} ∧
        (∀j:0≤j<|d.cmd|:
            refines(d.cmds[j],
                    d1.cmds[j])) =>
            refines(d,d1) ) ;

function optimize(d:dsgn) : dsgn ;
    where   d:dsgn =>
            refines(d,optimize(d)) ;

function complete(d:dsgn) : boolean ;
    post    complete = (
            d.cmd∈{skip,asgn} ∨
            d.cmd∈{if} ∧
                (∀c∈d.cmds:
                    complete(c.cmd) ) ∨
            d.cmd∈{do} ∧
                complete(d.body) ∨
            d.cmd∈{seq} ∧
                complete(d.s1) ∧
                complete(d.s2) ) ;
```

Figure 3. Design Process Support Routines

Figure 3 shows some support routines for the design process. "Dsgneq" is true if two designs are equivalent at the top-level; more precisely, if they have the same symbol table, pre- and post-condition, and command. "Refines" is used to define the correctness of the design process. For the purposes of this paper, the function is not precisely defined; however, the properties it must satisfy for the process to be correct are enumerated.

For example, if two designs are equal at the top-level and have a "cmd" of "undef", "skip", or "asgn" then they refine one another. This implies that any specification refines itself. If the designs of two loops are equal at the top-level and the body of the second refines the body of the first, then the second design refines the first one. If the designs of two statement sequences are equal at the top-level, and the sub-statments refine each other, then so do the designs. If the designs of two if statments are "dsgneq" and the guarded commands of the second refine those of the first, then the second design is a refinement of the first.

"Optimize" is a function that takes a design as input and returns a new one that has improved performance characteristics. At this point, we will specify nothing about it except that it preserves the "refines" relation. The function "complete" returns true if all the unknowns in a design have been filled in; in other words, if the design is for an assignment or null statement, or if it is for a loop and the body is completed, or if it is for a sequence and the sub-statements are done, or if it is for an if statement and all the guarded commands have been completed.

Figure 4 shows the code for the design process itself. The function "derive_design" takes a specification and if possible produces a complete design. In some cases it may not be able to produce a finished program, but it always preserves the "refines" relation. The body of "derive_design" consists of a single loop with an embedded conditional. Each iteration of the loop is concerned with a different cliche. If the current cliche is applicable to the specification, then it is applied and the result passed to "derive_subs" and then "optimize". If the cliche is not applicable then nothing is done. The loop terminates when a complete design is produced, or when all the cliches have been tried.

The function "derive_subs" takes a design and, if necessary, derives sub-designs to produce a complete program. The body consists of an if statment with an alternative for each command type. If the top-level design is for an if statement, then "derive_subs" loops

```
function derive_design(s:spec):dsgn is

{Q: true}
var d : dsgn := s ; k : integer := 0 ;
{inv P: 0≤k≤|all_cliches| ∧ refines(s,d)}
{bnd t: |all_cliches|-k}
do k≠|all_cliches| ∧ ¬complete(d) →
    c,k := all_cliches[k],k+1 ;
    if c.pre(s) →
        d := optimize(
                derive_subs(c.apply(s)));
     [] ¬c.pre(s) → skip ;
    fi ;
od ;
derive_design := d ;
{R: refines(s,derive_design)}
end derive_design ;

function derive_subs(d:dsgn) : dsgn is

{Q: true}
var ds : dsgn := d ;
if d.cmd∈{if} →
    var  k: integer := 0 ;
    {inv P: 0≤k≤|d.cmds| ∧
            dsgneq(d,ds) ∧
            (∀j:0≤j<k:
                refines(d.cmds[j],
                        ds.cmds[j]))}
    {bnd t: |d.cmds|-k}
    do k≠|d.cmds| →
        ds.cmds[k],k :=
            derive_design(d.cmds[k]),k+1;
    od ;
 [] d.cmd∈{do} →
    ds.body := derive_design(d.body) ;
 [] d.cmd∈{seq} →
    ds.s1 := derive_design(d.s1) ;
    ds.s2 := derive_design(d.s2) ;
 [] ds.cmd∈{asgn,skip,undef} → skip ;
fi ;
derive_subs := ds ;
{R: refines(d,derive_subs)}
end derive_subs ;
```

Figure 4. Design Process Code

through all the guarded commands generating a design for each one. If the top-level design is for a loop, then a design for the body is generated. If the top-level design is for a sequence of statements, then a design is generated for each one. If the top-level design is for an assignment statement, a null command, or an unknown, then nothing is done.

3.1. Proof of Process Code

We can now argue that the design process described in Figure 4 is correct in the sense that it always creates designs that refine their specifications. The argument is reasonably straight forward, but fairly lengthy. First we will demonstrate that "derive_subs" is correct with respect to its specification, then we use this result to show that "derive_design" is also correct. In both cases, the demonstrations will be in terms of the "refines" relation that must be maintained by application of the cliches. Therefore, the process will function correctly as long as it uses correct cliches.

Since "derive_design" and "derive_subs" are mutually recursive, to be extremely formal we should really perform an induction on the number of recursive calls, or a structural induction on the "dsgn" data type. For the purposes of this paper, suffice it to say that the base case is when "derive_subs" is called with an assignment, null or unknown statement, and that the induction step then assumes that only n recursive calls are needed and that they will execute correctly.

For the present, we will proceed less rigorously, performing a detailed check on the process design rather than a fully formal proof. We will begin by assuming that

d:dsgn => refines(d,derive_design(d))

and using this to show that

d:dsgn => refines(d,derive_subs(d)).

The latter implied by the pre- and post-conditions for "derive_subs"; therefore, we will prove the following theorem.

Theorem 1: {Q} derive_subs.body {R}

{Q} ds:=d {Q1} IF {R1} derive_subs:=ds {R}
where Q1: dsgneq(d,ds), R1: refines(d,ds)

1) {Q} ds:=d {Q1}
   true => dsgneq(d,d)
2) {Q1} IF {R1} by lemma 1
3) {R1} derive_subs:=ds {R}
   refines(d,ds) => refines(d,ds)

therefore, {Q} derive_subs.body {R}.

Lemma 1: {Q1} IF {R1}
where Q1: dsgneq(d,ds), R1: refines(d,ds)

1) Q1 => d.cmd∈{if} ∨
         d.cmd∈{do} ∨ d.cmd∈{seq} ∨
         d.cmd∈{asgn,skip,undef}

2.1) {Q1 ∧ d.cmd∈{if}} S1 {R1}
     lemma 1.1

2.2) {Q1 ∧ d.cmd∈{do}} S2 {R1}
dsgneq(d,ds) ∧ d.cmd∈{do} ∧
refines(d.body,ds.body) =>
refines(d,ds)

2.3) {Q1 ∧ d.cmd∈{seq}} S3 {R1}
dsgneq(d,ds) ∧ d.cmd∈{seq} ∧
refines(d.s1,ds.s1) ∧
refines(d.s1,ds.d1) =>
refines(d,ds)

2.4) {Q1 ∧ d.cmd∈{asgn,skip,undef}} skip {R1}
dsgneq(d,ds) ∧
d.cmd∈{asgn,skip,undef} =>
refines(d,ds)

therefore {Q1} IF {R1}.

Lemma 1.1: {Q1 ∧ d.cmd∈{if}} S1 {R1}
where Q1: dsgneq(d,ds), R1: refines(d,ds)

1) {Q1 ∧ d.cmd∈{if}} k:=0 {P}
{Q1 ∧ d.cmd∈{if}} => P^k_0
=> 0≤0≤|d.cmds| ∧ dsgneq(d,ds) ∧
(∀j:0≤j<0:
refines(d.cmds[j],ds.cmds[j]))

2) {P ∧ k≠|d.cmds|} S {P}
P ∧ k≠|d.cmds| ∧
refines(d.cmds[k],ds.cmds[k])
=> P^k_{k+1}
=> 0≤k+1≤|d.cmds| ∧ dsgneq(d,ds) ∧
(∀j:0≤j<k+1:
refines(d.cmds[j],ds.cmds[j]))

3) P ∧ k=|d.cmds| => R
dsgneq(d,ds) ∧
(∀j:0≤j<|d.cmds|:
refines(d.cmds[j],ds.cmds[j])) =>
refines(d,ds) ;

4) k≠|d.cmds| => |d.cmds|-k≥0

5) {P ∧ k≠|d.cmds|} t1:=t; S {t<t1}
P ∧ k≠|d.cmds| =>
|d.cmds|-(k+1)<|d.cmds|-k

therefore, {Q1 ∧ d.cmd∈{if}} S1 {R1}

The proof of "derive_subs" is now complete. We can now use this result to prove the following.

s:spec => refines(s,derive_design(d))

This is implied by the pre- and post-conditions for "derive_design" therefore, we will prove the following theorem.

Theorem 2: {Q} derive_design.body {R}
{Q} S1 {P} DO {P ∧ ¬B} S2 {R}

1) {Q} d,k:=s,0 {P}
Q => $P^{d,k}_{s,0}$
=> 0≤0≤|all_cliches| ∧ refines(s,s)

2) {P} DO {P ∧ ¬B} by lemma 2

3) {P ∧ ¬B} derive_design:=d {R}
refines(s,d) => refines(s,d)

therefore, {Q} derive_design.body {R}.

Lemma 2: {P} DO {P ∧ ¬B}

1) {P ∧ B} S {P}
let Q1: ¬complete(d) ∧ c=all_cliches[k-1] ∧
0≤k≤|all_cliches|
{P ∧ B} c,k:=all_cliches[k],k+1 {P ∧ Q1}
{P ∧ Q1} IF {P} by lemma 2.1

2) P ∧ k≠|all_cliches| ∧ ¬complete(d) =>
|all_cliches|-k≥0

3) {P ∧ B} t1:=t; S {t<t1}
P ∧ B =>
|all_cliches|-(k+1)<|all_cliches|-k

therefore, {P} DO {P ∧ ¬B}.

Lemma 2.1: {P ∧ Q1} IF {P}

1) P ∧ Q1 => c.pre(s) ∨ ¬c.pre(s)

2.1) {P ∧ Q1 ∧ c.pre(s)} S1 {P}
P ∧ Q1 ∧ c.pre(s) ∧ refines(s,d) =>
0≤k≤|all_cliches| ∧ refines(s,d)

2.2) {P ∧ Q1 ∧ ¬c.pre(s)} S2 {P}
P ∧ Q1 ∧ ¬c.pre(s) => P

therefore, {P ∧ Q1} IF {P}.

The proof of "derive_design" is now complete, and with it the proof of the entire process. We have not been extremely formal, but we have significantly increased our confidence that the program preserves the "refines" relation, assuming it is maintained by the "apply" function of each cliche. In other words, we have argued that this process will produce correct designs if it uses correct cliches. We will now turn to an examination of cliches and their correctness.

4. Cliches

The number of cliches that can be used in the design process is infinite; for the purposes of this paper, we will limit ourselves to three: the "simple_assignment" cliche generates (multiple) assignment statements, the "simple_if_then_else" cliche generates two branch if-then-else statements where the conditions are the negation of each other, and the "conditional_iteration_on_set" cliche generates loops with an embedded conditional.

We will discuss each cliche in turn, and argue that its application preserves the "refines" relationship. For the purposes of this paper, we will assume that "refines" holds when the design produced by application of a cliche is totally correct with respect to the specification from which it was produced. We will therefore show that application of each cliche produces designs that can be proven totally correct using standard proof rules [10, 15]

4.1. Simple_Assignment

The following is a simplified representation of the "simple_assignment" cliche.

```
cliche simple_assignment is

    {Q} Var1..VarN := Soln1..SolnN {R}
if
    Q => R[[Var1..VarN / Soln1..SolnN]]

end simple_assignment ;
```

The cliche states that "{Q} $Var_1..Var_N$:= $Soln_1..Soln_N$ {R}" is true if "Q" implies "R" with "$Soln_1..Soln_N$" substituted for "$Var_1..Var_N$". This representation makes understanding the cliche simple, and we can see that its correctness follows directly from the proof rule for assignment statements [10, 15].

$$\text{if } Q \Rightarrow R_e^x \text{ then } \{Q\}\ x{:=}e\ \{R\}$$

However, the above representation does not give much detail about how the cliche is implemented

Figure 5 shows a more detailed description of the "simple_assignment" cliche. To discuss this cliche description, we must first present some of the infrastructure on which our process program is built. A fairly standard symbol table underlies much of the program. We will not describe it in detail; however, understanding of the following routines is necessary.

```
function modlist(s:symtab) : seq(sym);
function uselist(s:symtab) : seq(sym);
function newsym(ss:seq(sym)):seq(sym);
```

The function "modlist" takes a symbol table as an argument and returns a list of the modifiable symbols in the current context. Similarly, "uselist" returns a list of the usable or accessible symbols. The function "newsym" takes a sequence of symbols and produces a new sequence that is identical to the original, except that the names of all symbols in the new list are unique, in other words they match no other symbol.

To apply a general cliche for assignment statements, we must somehow find a list of expressions that make a boolean expression true. Specifically, to generate an assignment "{Q} x:=e {R}" we must find a list of expressions, "e", that make "Q => R_e^x" evaluate to true. In general this problem is undecidable [15]. Our purpose is not to consider the difficulties and technology of theorem proving; therefore, we will encapsulate the problem by defining the following routines.

```
cliche simple_assignment is

    function pre(s:spec) : boolean is
        {Q: true}
        var m :seq(sym) := modlist(s.st);
        var u :seq(sym) := uselist(s.st);
        var ss:seq(sym) := newsym(m) ;
        pre := can_solve(ss,u
                bool_expr(s.pre =>
                    subst(ss,m,s.post)));
        {R: pre = asgn_able(s)}
        end pre ;

    function apply(s:spec) : dsgn is
        {Q: asgn_able(s)}
        var m :seq(sym) := modlist(s.st);
        var u :seq(sym) := uselist(s.st);
        var ss:seq(sym) := newsym(m) ;
        var np:bool_expr:=
                    subst(ss,m,s.post);
        apply.e :=
            solve(ss,u,
                  bool_expr(s.pre => np));
        apply.st,apply.pre,apply.post :=
                     s.st,s.pre,s.post;
        apply.cmd,apply.v := asgn,m ;
        {R: asgn_rule(apply)}
        end apply ;

end simple_assignment ;
```

Figure 5. *Simple_Assignment* Cliche

```
predicate solvable(m,u:seq(sym);
                   f:bool_expr) is
    (∃ss:seq(expr(u)):
        findable(ss,m,u,f) ∧
        provable(subst(ss,m,f)));

function can_solve(m,u:seq(sym);
                   f:bool_expr):boolean;
    post    can_solve(m,u,f) =
                   solvable(m,u,f) ;

function solve(m,u:seq(sym) ;
               f:bool_expr):seq(expr);
    pre     solvable(m,u,f) ;
    post    provable(subst(solve,m,f)) ;
```

A boolean expression "f" is "solvable" for modifiable symbols "m" by terms in the usable symbols "u" if there exists a list of expressions "ss" such that "f" with

"ss" substituted for "m" is provably correct, and "ss" can be found by the solution generation routine. The function "can_solve" returns true if and only if "solvable" is true, while "solve" is called with a solvable problem and returns a solution.

We can now translate the proof rule for assignments and the conditions necessary to apply the "simple_assignment" cliche into our programming notation.

```
function asgn_rule(a:dsgn):boolean;
    post    asgn_rule =
            (a.cmd = asgn ∧
            provable(
              bool_expr(a.pre =>
                subst(a.e,a.v,a.post))));

function asgn_able(s:spec):boolean;
    post    asgn_able =
            solvable(ss,u,
                bool_expr(s.pre =>
                   subst(ss,m,s.post))));
    where   ss= newsym(modlist(s.st)) ∧
            m = modlist(s.st) ∧
            u = uselist(s.st) ;
```

The predicate "asgn_rule" is true if the design in question is for an assignment statement, and the formula "a.pre => $(a.post)^{a.v}_{a.e}$" is provably correct. The predicate "asgn_able" holds for a specification if the formula required to prove the correctness of an assignment which would satisfy the specification is solvable.

The specification of "pre" requires that it return true if and only if "asgn_able(s)" is true. We can argue for this as follows.

Lemma 1: {Q} pre.body {R}
where Q: true, R: pre=asgn_able(s)

let ss'= newsym(modlist(s.st)),
u' = uselist(s.st),
m' = modlist(s.st),
f' = bool_expr(s.pre => subst(ss',m',s.post))

1) wp("pre:=can_solve(st ...)",R) =
Q3: can_solve(
ss,u,s.pre => subst(ss,m,s.post)) =
asgn_able(s)
2) wp("m:=modlist(s.st);
u:=uselist(s.st);
ss:=newsym(m)",Q3) =
Q1: can_solve(ss',u',f') = asgn_able(s)
4) Q => Q1
=> solvable(ss',u',f') =
solvable(ss',u',f')

therefore, {Q} pre.body {R}.

Lemma 2: {Q} apply.body {R}
where Q: asgn_able(s), R: asgn_rule(apply)

let ss'= newsym(modlist(s.st)),
u' = uselist(s.st),
m' = modlist(s.st),
f' = bool_expr(
s.pre => subst(ss',m',s.post)),
slv = solve(ss',u',f')
ff' = bool_expr(
s.pre => subst(slv,m',s.post)),
ass = dsgn(s.pre,s.post,s.st,asgn,m',slv)

1) wp(apply.body,R) = Q1: asgn_rule(ass)
2) asgn_able(s) => Q1
=> (asgn=asgn) ∧ provable(ff')
solvable(ss',u',f') => provable(ff')

therefore, {Q} apply.body {R}.

Lemmas one and two can be rewritten as follows.

Lemma 1: pre(s) = asgn_able(s)
Lemma 2: asgn_able(s) => asgn_rule(apply(s))

Therefore, it directly follows that

simple_assignment.pre(s) =>
asgn_rule(simple_assignment.apply(s))

We will take this as proof that "simple_assignment" preserves the "refines" relationship.

4.2. Simple_If_Then_Else

The following is a simplified representation of the "simple_if_then_else" cliche.

```
cliche simple_if_then_else is
    {Q}
    if B1 → {Q ∧ B1} < S1 > {B1 ∧ E1}
     [] B2 → {Q ∧ B2} < S2 > {B2 ∧ E2}
    fi
    {R: B1 ∧ E1 ∨ B2 ∧ E2}
if
    is_negation(B1,B2) ;

end simple_if_then_else ;
```

The cliche says that the statement
"if B1 → S1 [] B2 → S2 fi"
is correct with respect to any pre-condition "Q" and post-condition "B1 ∧ E1 ∨ B2 ∧ E2" if "B1" is the logical negation of "B2", "S1" is correct with respect to pre- and post-conditions "Q ∧ B1" and "B1 ∧ E1" respectively, and "S2" is correct with respect to "Q ∧ B2" and "B2 ∧ E2".

This representation is easy to understand, and allows a simple demonstration of correctness from the corresponding proof rule [10, 15].

let IF = if $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
•
•
[] $B_n \rightarrow S_n$
fi

BB = $(\exists k{:}1 \le k \le n{:}B_k)$

if 1) $Q \Rightarrow BB$
2) $\{Q \wedge B_k\}\ S_k\ \{R\},\quad 1 \le k \le n$

then $\{Q\}$ IF $\{R\}$

Figure 6 shows a more detailed description of the "simple_if_then_else" cliche. To discuss this figure, we must first describe a little more about how our process program is implemented. It is written in Prolog and makes significant use of the language's unification facility. Since our purpose is not to delve deeply into the theory or practice of logic programming, we will describe this facility very roughly as follows.

```
cliche simple_if_then_else is

    function pre(s:spec) : boolean is
        {Q: true}
        pre := simple_if_able(s) ;
        {R: pre = simple_if_able(s)}
        end pre ;

    function apply(s:spec) : dsgn is
        {Q: simple_if_able(s)}
        var q1,q2,r1,r2 : bool_expr ;
        var B1,B2,E1,E2 : logic_var ;
        unify(s.post,
              bool_epxr(
                B1 ∧ E1 ∨ B2 ∧ E2)));
        q1 := bool_expr(s.pre ∧ B1);
        q2 := bool_expr(s.pre ∧ B2);
        r1 := bool_expr(B1 ∧ E1)    ;
        r2 := bool_expr(B2 ∧ E2)    ;
        apply.pre,apply.post :=
                    s.pre,s.post;
        apply.st,apply.cmd := s.st,if ;
        apply.cmds :=
        [gcmd(B1,dsgn(q1,r1,s.st,undef)),
         gcmd(B2,dsgn(q2,r2,s.st,undef))];
        {R: if_rule(apply)}
        end apply ;

end simple_if_then_else ;
```

Figure 6. *Simple_If_Then_Else* Cliche

```
operator unify(
            e1,e2:inout expr):boolean;
  pre  e1=E1 ∧ e2=E2
  post unify=true ∧ e1=e2 ∧
       only_logic_vars_changed(e1,E1) ∧
       only_logic_vars_changed(e2,E2) ∨
       unify=false ∧ e1=E1 ∧ e2=E2 ;
```

If "unify" returns true, then expressions "e1" and "e2" are structurally identical. "Unify" modifies the expressions only by assigning values to logical variables; no changes to other structures are allowed. If "unify" returns false, then the the expressions are unchanged.

We can now translate the proof rule for if-then-else statements into our programming notation.

```
predicate if_rule(f:dsgn) is
    f.cmd=if ∧
    (f.pre => (∃c∈f.cmds:c.grd)) ∧
    (∀c∈f.cmds:
        (f.pre ∧ c.grd => c.cmd.pre) ∧
        (c.cmd.post => f.post) ∧
        correct(c)) ;

function simple_if_able(
              s:spec):boolean is
    var B1,B2,E1,E2 : logic_var ;
    simple_if_able :=
        unify( s.post,
               bool_epxr(
                   B1 ∧ E1 ∨ B2 ∧ E2)) ∧
        is_negation(B1,B2)) ;
```

The predicate "if_rule" is true of a design "f" if "f" satisfies the rule for if statements. More precisely, "if_rule" is true if: "f" is an if statement; the pre-condition of "s" implies that at least one of the guards is true; and for each guarded command "c" in "f", "s"'s pre-condition and "c"'s guard imply "c"'s statement's pre-condition, "c"'s statment's post-condition implies "s"'s post-condition, and "c"'s statement is correct with respect to its specification.

The function "simple_if_able" is true if a specification can be implemented with a simple if-then-else. More precisely, "simple_if_able(s)" is true if "s"'s post-condition is a boolean expression of the form "B1 ∧ E1 ∨ B2 ∧ E2" and "B1" is the negation of "B2". With these definitions in hand, we can turn our attention to the cliche itself.

The application condition for "simple_if_then_else" is simply that the function "simple_if_able" evaluates to true. We can argue for the correctness of the "pre" function as follows.

Lemma 1: {Q} pre.body {R}
where Q: true, R: simple_if_able(s)
1) simple_if_able(s)=simple_if_able(s)
therefore, {Q} pre.body {R}.

Application of the cliche involves a single unification and a number of assignments. The local variables "q1", "q2", "r1", and "r2" are used to hold the pre- and post-conditions for the branches of the if statement, while the logical variables "B1", "B2", "E1", and "E2" are used in the unification with the post-condition. The result of the application has the same pre-condition, post-condition, and symbol table as the specification, as well as a list of guarded commands constructed from the above. We can argue for the correctness of "apply" in the following way.

Lemma 2 : {Q} apply.body {R}
where Q: simple_if_able(s), R: if_rule(apply)
let q1' = bool_expr(s.pre ∧ B1),
q2' = bool_expr(s.pre ∧ B2),
r1' = bool_expr(B1 ∧ E1),
r2' = bool_expr(B2 ∧ E2),
s1' = dsgn(q1',r1',s.st,undef),
s2' = dsgn(q2',r2',s.st,undef),
gc1 = gcmd(B1,s1'),
gc2 = gcmd(B2,s2'),
if' = dsgn(s.pre,s.post,s.st,if,[gc1,gc2]),
post' = bool_expr(B1 ∧ E1 ∨ B2 ∧ E2))
1) wp(apply.assignments,R) = Q1: if_rule(if')
2) {Q} unify(s.post,post') {Q1}
Q => unify(s.post,post')= true ∧
is_negation(B1,B2)
{unify(s.post,post')= true ∧ is_negation(B1,B2)}
unify(s.post,post')
{is_negation(B1,B2) ∧ s.post=post'}
by specification of unify
s.post=post' ∧ is_negation(B1,B2) => Q1
=> if=if ∧
(s.pre => B1 ∨ B2) ∧
(s.pre ∧ B1 => s.pre ∧ B1) ∧
(B1 ∧ E1 => post') ∧
correct(s1') ∧
(s.pre ∧ B2 => s.pre ∧ B2) ∧
(B2 ∧ E2 => post') ∧
correct(s2')
correct(s1') ∧ correct(s2') by assumption
therefore, {Q} apply.body {R}.

Lemmas one and two can be rewritten as follows.

Lemma 1: pre(s) = simple_if_able(s)
Lemma 2: simple_if_able(s) => if_rule(apply(s))

Therefore, it directly follows that

simple_if_then_else.pre(s) =>
if_rule(simple_if_then_else.apply(s))

We will take this as proof that "simple_if_then_else" preserves the "refines" relationship.

4.3. Conditional_Iteration_on_Set

Figure 7 shows a simplified representation of the "conditional_iteration_on_set" cliche. Application of this cliche can solve problems that require the use of a loop with an embedded conditional. In such cases, computation of the desired result involves processing each element of a set in turn. In the completed design, a local set variable holds all the items still to be processed, while a local scalar holds the item currently under examination. The result variable is initialized to the identity element before the loop begins, and each iteration modifies the result depending on whether the item under examination satisfies a certain property.

The post-condition of the cliche states that "Var" is equal to the value of "Iop(Set,Cond)"; in other words, to the value of an iteration operator applied to a set with a certain condition. Many different concrete post-conditions can be unified with this abstract one. For example, the following post-conditions occur in our specification of Kemmerer's Library Problem [26].

```
cliche conditional_iteration_on_set is
    {Q}
    var Lset : set(Stype):= Set;
    var Lvar : Stype      := Id ;
    {inv P:Lset⊆Set ∧
          Var=Iop(Set-Lset,Cond)}
    {bnd t:|Lset|}
    do Lset≠{} →
        choose(Lset,Lvar);
        Lset:=Lset-Lvar  ;
        {Q1:Var=VAR}
        < S1 >( Var:inout Rtype ) ;
        {R1:¬Cond(Lvar) ∧ Var=VAR ∨
             Cond(Lvar) ∧
                Var=Op(VAR,Lvar)}
    od
    {R: Var = Iop(Set,Cond)} ;

if
    (Id,Op(Var,Lvar),Iop(Set,Cond))
          ∈ iop_table ;

end conditional_iteration_on_set ;
```

Figure 7. *Conditional_Iteration_on_Set* Cliche

```
nout       = (Nc∈checks:c.name=u)
is_out     = (∃u∈users:checked_out(u,b)))
active     = (∃b∈books:checked_out(u,b)))
by_author  = {b∈books:b.author=a}
on_subject= {b∈books:b.subject=sb}
what_out   = {b∈books:corec(u,b)∈checks}
who_has    = {u∈users:corec(u,b)∈checks}
```

Each of these matches the post-condition for the "conditional_iteration_on_set" cliche. More specifically, the first unifies with "Var" equal to "nout", "Iop" equal to "number_of", "Set" equal to "checks", and "Cond" equal to "c.name=u". The second matches with "Var" equal to "is_out", "Iop" equal to "there_exists", "Set" equal to "users", and "Cond" equal to "checked_out(u,b)". The others are similar.

The body of "conditional_iteration_on_set" declares two local variables. "Lset" is a set containing all the items still to be considered, while "Lvar" is the item currently being processed. In this case, our programming notation is polymorphic in that "Stype" is deduced from the context. "Lset" is initialized to "Set" and the result to "Id". The loop interates over all the items in "Set". If the item in question satisfies "Cond" then the result ("Var") is set to "Op(Var,Lvar)". When all items have been considered, the correct result has been calculated.

In general, knowing when this cliche can be applied is difficult: how can we determine which operators are allowed and what the identity elements are? In practice, checking for applicability is simple: the cliche can be applied if the operator unifies with one of the elements in a pre-computed table. Each entry in "iop_table" satisfies the following properties.

1) Id = Iop({},Cond)
2.1) (s∈ss ∧ Cond(s) =>
Iop(ss,Cond) = Op(Iop(ss-s,Cond),s))
2.2) (s∈ss ∧ ¬Cond(s) =>
Iop(ss,Cond) = Iop(ss-s,Cond))

These properties are exactly those necessary to prove the correctness of the cliche body and ensure that all the designs produced from the cliche will be correct.

The implementation this cliche is quite lengthy; for the sake of brevity, we will not describe it here. Rather, we will now prove the cliche's correctness using the simplified representation. Towards this purpose, Figure 8 shows a fully annotated version of the cliche body. The top level structure of the proof is as follows.

{Q} I {Q'} DO {R'} {R}

To prove the cliche is correct, we must prove that the initialization sets up the loop in the proper manner ({Q} I {Q'}), that the loop is correct ({Q'} DO {R'}), and that correct termination of the loop ensures the post-condition for the routine is satisfied (R' => R).

```
{Q}
var Lset : set(Stype):= Set;
var Lvar : Stype      := Id ;
{Q': Lvar=Id ∧ Lset=Set}
{inv P: Lset⊆Set ∧
          Var=Iop(Set-Lset,Cond)}
{bnd t: |Lset|}
do Lset≠{} →
    {Q1': P ∧ Lset≠{} ∧
          Lset=LSET ∧ Var=VAR}
    choose(Lset,Lvar);
    Lset:=Lset-Lvar  ;
    {Q1: Var=VAR ∧
        Q2: LSET⊆Set ∧
            VAR = Iop(Set-LSET,Cond) ∧
            Lset=LSET-Lvar ∧
            Lvar∈LSET}
    < S1 >( Var:inout Rtype ) ;
    {R1:    Q2 ∧
            ¬Cond(Lvar) ∧ Var=VAR ∨
             Cond(Lvar) ∧
                Var=Op(VAR,Lvar)}
od
{R': P ∧ Lset={}}
{R: Var = Iop(Set,Cond)} ;
```

Figure 8. Fully Annotated Cliche Body

The proof of the initialization uses two applications of the assignment rule and relys on property one of "iop_table" ("Id=Iop({},Cond)"). We can see that R' implies R by simply expanding their definitions.

R' => R
P ∧ Lset={} => R
Var=Iop(Set-Lset,Cond) ∧ Lset={} =>
Var=Iop(Set,Cond)

The proof of the loop is reasonably straight forward, but a bit more complicated. It uses two lemmas that we will prove before proceeding and assumes that the unknown in the loop body is completed correctly.

Lemma 1: {Q1'} S1' {Q1}

{Q1'} choose(Lset,Lvar) {Q1' ∧ Lvar∈Lset}
{Q1' ∧ Lvar∈Lset} Lset:=Lset-Lvar {Q1}
Q1' ∧ Lvar∈Lset
=> $(Q1)^{Lset}_{Lset-Lvar}$
=> Var=VAR ∧
LSET⊆Set ∧
VAR = Iop(Set-LSET,Cond) ∧
Lset-Lvar=LSET-Lvar ∧
Lvar∈LSET

therefore, {Q1'} S1' {Q1}.

Lemma 2: R1 => P

R1 => Q2
=> LSET⊆Set ∧
T1:(Lset=LSET-Lvar ∧ Lvar∈LSET)
=> P1:Lset⊆Set
R1 => Q2
=> T2:(VAR = Iop(Set-LSET,Cond))
R1 => T3:(Cond(Lvar) ∧ Var=Op(VAR,Lvar)) ∨
T4:(¬Cond(Lvar) ∧ Var=VAR)
T1 ∧ T2 ∧ T3 => P2:(Var = Iop(Set-LsetCond))
by prop 2.1 of iop_table
T1 ∧ T2 ∧ T4 => P2
by prop 2.2 of iop_table
P1 ∧ P2 => P

therefore, R1 => P

Using these two lemmas, we can prove the correctness of the loop.

1) Q' => P
Lvar=Id ∧ Lset=Set => $P^{Lvar,Lset}_{Id,Set}$
Q' => Set⊆Set ∧
Id = Iop(Set-Set,Cond)
Id = Iop({},Cond) by prop 1 of iop_table

2) {P∧B} S {P}
{P∧B} {Q1'} S1' {Q1} S1 {R1} {P}
P ∧ Lset≠{} => Q1'
{Q1'} S1 {Q1} by lemma 1
{Q1} S1 {R1} by assumption
R1 => P by lemma 2

3) P∧¬BB => R'
P∧¬(Lset≠{}) => P ∧ Lset={}

4) P => (t≥0)
P => |Lset| ≥0

5) {P∧B} t1:=t; S1'; S1 {t<t1}
{P∧B} t1:=t {t1=t} S1' {t<t1} S1 {t<t1}
{P∧B} t1:=t {t1=t}
{t1=t} choose(Lset,Lvar) {t1=t ∧ Lvar∈Lset}
{t1=t ∧ Lvar∈Lset} Lset:=Lset-Lvar {t<t1}
{t<t1} S1 {t<t1}
S1 modifies only Var

therefore, {Q'} DO {R'}.

The proof of the loop is now complete, and with it the proof of "conditional_iteration_on_set". We will take this as a demonstration that application of the cliche preserves the "refines" relation. We have now described three cliches and argued for their correctness. We will now turn to an example of design derivation using the framework we have constructed.

5. Example Design Derivation

Kemmerer's Library problem has received considerable attention in the software engineering literature and has been formally specified a number of times [14,27,28]. Using the cliches described in section four, the process program presented in section three can generate a complete design for the Library Problem from our formal specification [26]. The problem is concerned with a small library database that provides both query and update transactions to library staff and users. The architectural design for our solution consists of a single module that encapsulates the database and provides an entry routines for each transaction. The state of the module is modeled abstractly using high-level data types, and the entry routines are specified using pre- post-conditions.

In the derivation performed by our process program, the designs of the entry routines fall into two categories. The *simple routines* can all be implemented with a single assignment statement, while the *complex routines* require a loop with an embedded conditional. In other words, the simple routines can be designed with a single application of the "simple_assignment" cliche, while the complex routines require first an application of "conditional_iteration_on_set", then an application of "simple_if_then_else", and finally two applications of "simple_assignment".

We will present the derivation of one routine from each class in reasonable detail and leave it to the reader to infer how the other routines are derived.

5.1. Simple Routines

Let us begin with the design of the "add_book" routine.

```
procedure  add_book(  s:in vuser;
                      b:in book) ;
    pre    s.staff ∧
           b∉books ∧ books=BOOKS ;
    post   books=BOOKS+b ;
    modify books ;
```

The specification of "add_book" references the global variable "books", which is declared as follows.

```
var books : set(book) ;
```

The procedure takes two arguments. The first is the user performing the transaction, while the second is the

book being added to the library. The pre-condition states that the transaction is being invoked by a staff member, that the book being added is not already in the library, and that the initial value of "books" is represented by the constant "BOOKS". The post condition states that the current value of "books" is equal to the initial value with the addition of the new book.

If we rewrite the specification, we can easily see that the "simple_assignment" cliche is applicable.

```
{Q: s.staff ∧ b∉books ∧ BOOKS=books}
< S >(books:inout set(book))
{R: books=BOOKS+b}
```

To determine that the cliche is applicable to this problem, "simple_assignment.pre" calls "can_solve" to see if the system can solve the following formula for "e".

```
s.staff ∧ b∉books ∧ BOOKS=books =>
    e=BOOKS+b
```

It can, so when "apply" is invoked "solve" generates the solution "books+b", and the following complete design is produced.

```
{Q: s.staff ∧ b∉books ∧ BOOKS=books}
books := books+b ;
{R: books=BOOKS+b}
```

All of the other simple routines have similar derivations. The designs of the complex routines are considerably more difficult to produce.

5.2. Complex Routines

Consider the "who_has" function, which returns the user who currently has a book checked out.

```
function who_has( s:vuser ; b:vbook
                ) : set(vuser);
    pre  s.staff ;
    post who_has =
           {u∈users:corec(u,b)∈checks};
```

This specification uses the type "corec" and variable "checks" which are declared as follows.

```
type corec =   record
               name : vuser ;
               item : vbook ;
               end corec ;

var checks :   set(corec);
```

A "corec" records the fact that a book is checked out from the library. It contains both the book and the patron who borrowed it. "Checks" holds a check out record for each book currently on loan from the library.

The "who_has" function takes two arguments. The first is the user performing the transaction, and the second is the book in question. The pre-condition states that the transaction is being invoked by a staff member, while the post-condition states that the return value is the set of all users who have the book checked out.

If we rewrite the specification of "who_has", we can see that the "conditional_iteration_on_set" cliche is applicable.

```
{Q: s.staff}
< S >( who_has:out set(user) );
{R: who_has =
       {u∈users:corec(u,b)∈checks}}

{Q}
< conditional_iteration_on_set >
{R: Var = Iop(Set,Cond) }
```

The specification and cliche unify as follows.

Var = who_has
Iop = "set_of_all"
Set = users
Cond = corec(u,b)∈checks

Figure 9 shows the result of applying the "conditional_iteration_on_set" cliche to the specification. The overall structure of the design is now evident. The loop iterates over all the users in the library. For each user, the body checks if the user has the book in question checked out. If so, then the user is added to "who_has", if not then nothing is done. The variable "usrs" holds the set of all users still to be considered, while "usr" holds the user currently being examined.

The loop body must still be completed before the design is finished. It has the following specification.

```
{Q1:who_has=WHO_HAS}
< S1 >( who_has:inout set(user) ) ;
{R1:corec(usr,b)∈checks ∧
       who_has=WHO_HAS+usr ∨
    corec(usr,b)∉checks ∧
       who_has=WHO_HAS}
```

We can see that the "simple_if_then_else" cliche is applicable.

```
{Q}
< simple_if_then_else >
{R: B1 ∧ E1 ∨ B2 ∧ E2}
```

The specification and cliche unify as follows.

```
{Q: true}
var usrs : set(user) ;
var usr  : user      ;
who_has,usrs:={},users ;
{inv P:usrs⊆users ∧
       who_has =
           {u∈users-usrs:
               corec(u,b)∈checks}}
{bnd t: |usrs|}
do usrs≠{} →
    choose(usrs,usr) ;
    usrs:=usrs-usr ;
    {Q1: who_has=WHO_HAS}
    < S1 >(who_has:inout set(user));
    {R1:corec(usr,b)∈checks ∧
           who_has=WHO_HAS+usr ∨
        corec(usr,b)∉checks ∧
           who_has=WHO_HAS}
od
{R: who_has =
       {u∈users:corec(u,b)∈checks}}
```

Figure 9. Instantiated Loop Cliche

B1 = corec(usr,b)∈checks
B2 = corec(usr,b)∉checks
E1 = (who_has=WHO_HAS+usr)
E2 = (who_has=WHO_HAS)

Application of the "simple_if_then_else" generates the following design for the loop body.

```
if corec(usr,b)∈checks →
    {Q2:who_has=WHO_HAS ∧
           corec(usr,b)∈checks}
    < S2 >(who_has:inout set(user));
    {R2: corec(usr,b)∈checks ∧
           who_has=WHO_HAS+usr}
 [] corec(usr,b)∉checks →
    {Q3:who_has=WHO_HAS ∧
           corec(usr,b)∉checks}
    < S3 >(who_has:inout set(user));
    {R3:corec(usr,b)∉checks ∧
           who_has=WHO_HAS}
fi
```

We can complete the design of the loop body by applying the "simple_assignment" cliche twice. As a final flourish, the "optimize" routine transforms the assignment "who_has:=who_has" into "skip" producing the following.

```
if corec(usr,b)∈checks →
    who_has:=who_has+usr ;
 [] corec(usr,b)∉checks → skip ;
fi
```

Figure 10 shows the entire design for the "who_has" routine.

This completes our description of the derivations automatically performed by our process program. We will now turn to a very brief description of its implementation.

6. Implementation Status

The prototype implementation of the process program described in this paper is written in Prolog [4, 5] and makes significant use of the language's tree data structures and unification facilities. The prototype very closely follows the design given in section three; the implementation can be generated from the design using methods similar to [24, 25].

For example, Figure 11 shows the Prolog implementation of the "derive_design" routine. The relation "all_cliches" stores the cliches known to the system. The procedure "applicable" obtains the pre-condition

```
{Q: true}
var usrs : set(user) ;
var usr  : user      ;
who_has,usrs:={},users ;
{inv P:usrs⊆users ∧
       who_has =
           {u∈users-usrs:
               corec(u,b)∈checks}}
{bnd t: |usrs|}
do usrs≠{} →
       choose(usrs,usr) ;
       usrs:=usrs-usr ;
       if corec(usr,b)∈checks →
           who_has:=who_has+usr ;
        [] corec(usr,b)∉checks →
           skip ;
       fi
od
{R: who_has =
       {u∈users:
           corec(u,b)∈checks}}
```

Figure 10. Completed *Who_Has* Design

```
derive_design(S,D) :-
    all_cliches(C),
    applicable(C,S),
    apply(C,S,T1),
    derive_subs(T1,T2),
    optimize(T2,D).

applicable(C,S) :-
    C = cliche(_,pre(S,Pbody),_),
    Pbody.
apply(C,S,D) :-
    C = cliche(_,_,apply(S,D,Abody)),
    Abody.
```

Figure 11. Prolog Code for *Derive_Design*

for the cliche and executes it, while "apply" does the same for the cliche application function. The procedure "derive_subs" generates all the sub-designs for a construct, while "optimize" produces equivalent designs with enhanced performance characteristics.

Our prototype includes representations of all the cliches described in section four of this paper. For example, Figure 12 shows the Prolog implementation of the "simple_if_then_else" cliche. The cliche data structure is

```
cliche(simple_if_then_else,
    pre( ((Q,
           _,
           (B1 ∧ E1 ∨ B2 ∧ E2) ),
           St),
           ln_negation(B1,B2) ),
    apply( ((Q,_,R),St),((Q,S,R),St),
         ( R = (B1 ∧ E1 ∨ B2 ∧ E2),
           Q1 = (Q ∧ B1), R1 = (B1 ∧ E1),
           Q2 = (Q ∧ B2), R2 = (B2 ∧ E2),
           S=if([gc(B1,((Q1,S1,R1),St)),
           gc(B2,((Q2,S2,R2),St))])))).
```

Figure 12. Prolog Code for *Simple_If_Then_Else*

```
cliche(N,pre(S,Pbody),apply(S,D,Abody),
```

where "N" is the cliche name, "S" is the specification to which the cliche is applied, "D" is the design produced, "Pbody" is the body of the pre-condition, and "Abody" is the body of the application function.

The design data structure is "((Q,S,R),St)", where "Q" is the pre-condition, "S" is the command, "R" is the post-condition, and "St" is the symbol table. The if data structure is "if(GCLIST)", where "GCLIST" is a list of guarded commands. Each guarded command has the structure "gc(B,S)", where "B" is the guard and "S" is a command design.

The pre-condition of "simple_if_then_else" is true if the specification's post-condition unifies with "B1 ∧ E1 ∨ B2 ∧ E2" and "B1" is the negation of "B2". The body of "apply" produces a design for an if statement with two alternatives from the specification. The first alternative has guard "B1" and an undefined command with pre-condition "Q ∧ B1" and post-condition "B1 ∧ E1". The second is similar, but with "B2" and "E2" substituted for "B1" and "E1" respectively.

The current prototype automatically generates a complete design for Kemmerer's Library Problem from our formal specification [26]. The implementation is somewhat sketchy, especially the logic manipulation and theorem proving routines; however, it does demonstrate that the design in this paper is basically correct and definitely implementable.

7. Summary and Conclusions

It has been suggested that *process programming* can improve the effectiveness of software development [1, 12, 13, 16, 17, 23]. Describing development processes using programming language constructs should provide a better understanding of the activities and products involved. In addition, these concrete models may provide a basis for process automation. In this paper, we have presented a process program for the algorithm design technique developed by Dijkstra and Gries [6, 7, 10].

This method takes a pre- and post-condition specification written in first-order predicate logic and produces an algorithm design written in guarded commands. The process uses a library of *cliches* describing solutions to common programming problems. The process consists of a sequence of steps, each of which applies a pre-verified cliche to the current partial design. Since each cliche only generates correct transformations, the final design satisfies the original specification.

The Gries/Dijstra design method is an example of a two level, cliche driven process. At the lower level, a derivation process transforms a problem specification into a solution using a library of cliches.

Since the correctness of the final solution depends on the correctness of the cliches used in its derivation, the upper level uses verification rules (either formal or informal) to certify that the cliches in the library are correct.

We have presented the design for our process program using guarded commands and argued for its correctness assuming the correctness of the cliches it uses. We have also described three cliches that can be applied by the process and demonstrated that they only produce designs that are totally correct with respect to their specifications.

We have constructed a prototype implementation of the process in Prolog and used it to automatically generate a complete design for Kemmerer's Library Problem from our formal specification [26]. The implementation is somewhat sketchy, especially the logic manipulation and theorem proving routines; however, it does demonstrate that our description of the process is basically correct and definitely implementable.

The process program can be thought of as a translator for a very high-level language: the specifications extend conventional programming languages with quantifiers and high-level data types. The process functions as a black box: specifications come in and designs go out with no intervention from the programmer. We do not believe that this extreme prescription and lack of interaction represent the optimal description of this process; however, we do feel that the program as written defines the design technique in reasonable detail and represents a solid step in the right direction.

8. References

1. *Proceedings of the 5th International Software Process Workshop*, ACM Press, New York, 1989.
2. Balzer, R., "A 15 Year Perspective on Automatic Programming", *IEEE Transactions on Software Engineering SE-11, 11* (November 1985), 1257-1268.
3. Brooks, F. P., "No Silver Bullet", *IEEE Computer 20, 4* (April 1987), 10-19.
4. Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
5. Davis, R. E., "Logic Programming and Prolog: A Tutorial", *IEEE Software 2, 5* (September 1985), 53-62.
6. Dijkstra, E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *Communications of the ACM 18*, 8 (August 1975), 453-457.
7. Dijkstra, E. W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1976.
8. Fairley, R., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
9. Goldberg, A. T., "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques", *IEEE Transactions on Software Engineering SE-12, 7* (July 1986), 752-768.
10. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.
11. Humphrey, W. S., *Managing the Software Process*, Addison-Wesley, Reading, Massachusetts, 1989.
12. *Proceedings of the First International Conference on the Software Process*, IEEE Computer Science Press, Los Alamitos, CA, October 1991.
13. *Proceedings of the 6th International Software Process Workshop*, IEEE Computer Society Press, Los Alamitos, CA, October 1990.
14. Kemmerer, R. A., "Testing Formal Specifications to Detect Design Errors", *IEEE Transactions on Software Engineering SE-11, 1* (January 1985), 32-43.
15. Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, New York, 1984.
16. Madhavji, N. H., "Fragtypes: A Basis for Programming Environments", *IEEE Transactions on Software Engineering 14*, 1 (January 1988), 85-97.
17. Osterweil, L. J., "Software Processes Are Software Too", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 2-13.
18. Rich, C. and R. C. Waters, eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman Publishers, Los Altos, CA, 1986.
19. Rich, C. and R. C. Waters, "Automatic Programming: Myths and Prospects", *IEEE Computer 21*, 8 (August 1988), 40-51.
20. Ruebenstein, H. B. and R. C. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition", *IEEE Transactions on Software Engineering 17*, 3 (March 1991), 226-240.
21. Smith, D. R., G. B. Kotik and S. J. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute", *IEEE Transactions on Software Engineering SE-11, 11* (November 1985), 1278-1295.
22. Smith, D. R., "KIDS: a Semiautomatic Program Development System", *IEEE Transactions on Software Engineering 16*, 9 (September 1990), 1024-1043.
23. Sutton, S. M., D. Heimbigner and L. J. Osterweil, "Language Constructs for Managing Change in Process-Centered Environments", *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, December 1990, 206-217.
24. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", *Proceedings of the 10th International Conference on Software Engineering*, April 1988, 344-354.
25. Terwilliger, R. B. and R. H. Campbell, "PLEASE: Executable Specifications for Incremental Software Development", *Journal of Systems and Software 10*, 2 (September 1989), 97-112.
26. Terwilliger, R. B., "A Formal Specification and Verified Design for Kemmerer's Library Problem", Report No. CU-CS-562-91, Dept. of Computer Science, U. of Colorado at Boulder, December 1991.
27. Wing, J. M., "A Study of 12 Specifications of the Library Problem", *IEEE Software 5*, 4 (July 1988), 66-76.
28. *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987.