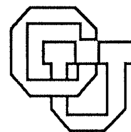
An Example of Process Verification: the Gries/Dijkstra Design Method

Robert B. Terwilliger

CU-CS-646-93 March 1993



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

An Example of Process Verification: the Gries/Dijkstra Design Method

Robert B. Terwilliger

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

ABSTRACT

We have constructed a simulation of the formal design process described by Gries in his book *The Science of Programming*. The system takes a pre- and post-condition specification and transforms it into a provably correct design using a library of *cliches*, complex knowledge structures representing commonly occurring situations. In this paper, we briefly describe the simulation's design and sketch a demonstration of its correctness. The argument has has two parts. First, we show that the system always produces correct designs if all the cliches in the library are valid, and second, we demonstrate that the cliches in the library always produce designs which satisfy their specifications.

1. Introduction

Some have suggested that *formal methods* can enhance both the software specification and design processes [2-7, 12, 16, 17, 22, 27, 38]. For example, Dijkstra and Gries [8, 9, 13, 14] have developed a process which takes a pre- and post-condition specification written in first-order predicate logic and incrementally transforms it into a verified design written using guarded commands. Although this technique has been used and taught for over a decade, at present is it difficult to either define it precisely, determine if it has been applied properly, or evaluate it for effectiveness.

There are many approaches to defining software processes. For example, advocates of *process programming* believe that describing development methods using programming language constructs should ultimately allow these processes to become automated, and their execution to be monitored and tuned for maximum efficiency [15, 20, 23, 29, 30]. To a certain extent, these goals overlap with *knowledge-based software engineering* [1, 10, 11, 19, 21, 24-26, 28]. An important contribution of this community is the notion of *cliche* (or *plan* or *schema*): a complex knowledge structure representing a commonly occurring situation.

We have been investigating the formal design method described by Gries in his book *The Science of Programming*; our three part approach includes walk-throughs, simulations, and process programming [33-37]. Our simulation of the Gries/Dijkstra process uses a library of cliches to create a provably correct design from a pre- and post-condition specification. The simulation is designed using guarded commands, and we have rigorously verified its correctness [33, 36]. This argument has has two parts. First, we show that the simulation always produces correct designs if all the cliches in the library are valid, and second, we demonstrate that the cliches in the library always produce designs which satisfy their specifications.

In the remainder of this paper, we present the design of our simulation and argue for its correctness in more detail. In section two, we discuss the architecture of our system, and in section three, we present the basic data type and correctness definitions. In section four, we describe the design process itself and argue for its correctness assuming that all the cliches it uses behave properly. In section five, we present a fairly detailed description of a simple cliche and its verification, as well as a larger cliche and its proof in less detail. Finally, in section six we summarize and draw some conclusions from our experience.

2. Simulation Architecture

Figure 1 shows the architecture of the design process implemented in our simulation program. It has two levels. At the lower level, the design derivation sub-process transforms formal specifications into verified designs using a library of cliches representing solutions to common programming problems. On the upper level, a cliche derivation sub-process constructs and verifies cliches. Cliche derivation is considerably more difficult than cliche application; therefore, the portion of the process inside the dashed box is automated and the rest is performed by a human.

The simulation inputs both a pre- and post-condition specification and the library of pre-verified cliches. Each cliche has an applicability condition, as well as a rule for transforming specifications into more complete programs. The simulation applies cliches until a complete design is produced or no cliches are applicable. The library of cliches is searched in a fixed order, with the simplest (least expensive to apply) cliches appearing first. Application of a cliche may

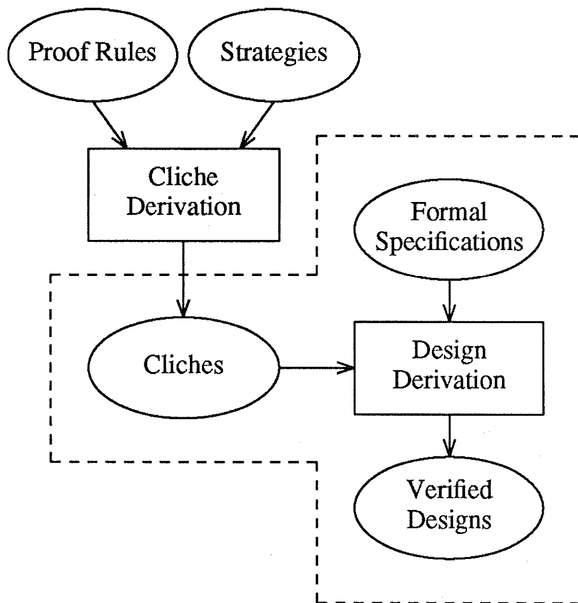


Figure 1. Simulation Architecture

generate sub-specifications for which a design must be created, and a simple backtracking scheme allows transformations to be undone if they do not lead to a complete solution.

Since the correctness of a final design depends on the correctness of the cliches used in its derivation, each cliche must be proven to produce only designs that satisfy the corresponding specification. The advantage of our two level architecture is that proofs are performed mostly at "compile" rather than "run" time. Cliche construction and verification is quite difficult, but is done only once for each cliche and is performed by a human. On the other hand, cliche application is reasonably easy and is performed repeatedly by the machine.

The design of our simulation is based on the architecture in Figure 1 and uses constructs that can be reasonably implemented in most programming languages [33,34,36,37]. A prototype has been written in Prolog that follows the design very closely; in fact, the implementation can be generated using methods similar to [31,32]. The prototype is somewhat sketchy, especially the logic manipulation and theorem proving routines; however, it does demonstrate the design's basic validity.

3. Data Type and Correctness Definitions

Figure 2 shows the type declarations used in the design of our simulation. 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, while 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). A design record with a "cmd" field of "undef" represents

```
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  : bool_expr  ;
                        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 ;
```

Figure 2. Simulation Type Definitions

a specification ("spec"): pre- and post-conditions with no statement in between. (Note that many of the constraints on designs are not enumerated; for example, the types of expressions and variables must match for assignment statements.)

Figure 3 shows the functions that define the correctness of our simulation. The goal of the Gries/Dijkstra process is a provably valid design; therefore, correctness is defined with a proof rule for each construct. Analogously, "correct" is defined in terms of five functions, one for each proof rule (we will present an example function in a later section). While this describes correctness for a single design, it does not define the relationship between designs.

We can think of (partial) designs as representing sets containing all the (complete) designs which can be produced from them. A design ("d1") then refines another ("d2") if the set corresponding to "d1" is a subset of the one corresponding to "d2". The function "refines" formalized this relationship. For example, a design refines a specification if it has the same pre- and post-conditions, its symbol table is a superset of the specification's (this allows for additional declarations), and the design itself is correct.

```
function correct(d:dsgn):boolean is
    d.cmd=skip ∧ skip_rule(d) ∨
    d.cmd=asgn ∧ asgn_rule(d) ∨
    d.cmd=seq  ∧ seq_rule(d)  ∨
    d.cmd=do   ∧ do_rule(d)   ∨
    d.cmd=if   ∧ if_rule(d)   ;

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

function refines(d1,d2:dsgn):boolean is
  d1.cmd=undef ∧
    d1.st⊆d2.st ∧ d1.pre=d2.pre ∧
    d1.post=d2.post ∧ correct(d2) ∨
  dsgneq(d1,d2) ∧ (
    d1.cmd∈{asgn,skip,undef} ∨
    d1.cmd∈{do} ∧
        refines(d1.init,d2.init) ∧
        refines(d1.dec, d2.dec ) ∧
        refines(d1.body,d2.body) ∨
    d1.cmd∈{seq} ∧
        refines(d1.s1,d2.s1) ∧
        refines(d1.s2,d2.s2) ∨
    d1.cmd∈{if} ∧
        (∀j:0≤j<|d1.cmd|:
            refines(d1.cmds[j].cmd,
                    d2.cmds[j].cmd)) );
```

Figure 3. Process Correctness Definitions

"Refines" references the function "dsgneq", which returns true if two designs are equivalent at the top-level; more precisely, if they have the same command, the symbol table of the first is a subset of the one for the second, and their pre- and post-conditions are identical. A design ("d2") "refines" another ("d1) if the two are "dsgneq" and the sub-designs of "d2" refine those of "d1". For example, the designs of two loops refine each other if they are equal at the top-level and the initialization, decrement, and body of the second refine those of the first.

The types and definitions presented so far define the framework in which our simulation is constructed; however, the system is based on cliches. 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.

```
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)));
```

"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".

Our simulation is based on simple constructs that can be reasonably implemented in most programming languages, although cliches may be problematic. We define the correctness of designs in terms of proof rules for each construct, and we describe the correctness of the process in terms of a "refines" relation between designs. We require the correctness of cliches; therefore, for the moment we will assume their validity and turn our attention to the design process itself.

4. Design Process

Figure 4 shows the code for the design process. There is a global variable "all_cliches" that holds all the cliches currently known to the system. The function "derive_design" takes a specification and produces a correct design, while "derive_subs" takes a design

```
var  all_cliches : seq(cliche) ;

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].cmd,
                        ds.cmds[j].cmd))}
    {bnd t: |d.cmds|-k}
    do k≠|d.cmds| →
        ds.cmds[k].cmd,k :=
        derive_design(d.cmds[k].cmd),k+1;
    od ;
 [] d.cmd∈{do} →
    ds.init := derive_design(d.init) ;
    ds.dec  := derive_design(d.dec ) ;
    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

and derives any sub-components necessary to complete it. "Derive_design" uses the functions "optimize", which takes a design as input and returns a new one that has improved performance characteristics, and "complete", which returns true if all the unknowns in a design have been filled in.

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

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

"Derive_design" takes a specification and if possible produces a complete design; in any case, it always preserves the "refines" relation. The body of the function consists of a single loop, each iteration of which concerns 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"; otherwise, nothing is done. The loop terminates when a complete design is produced, or when all the cliches have been tried.

"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. For example, if the top-level design is for an if statement, then "derive_subs" loops through all the guarded commands generating a design for each one. On the other hand, if the top-level design is for an assignment statement or a null command then nothing is done.

The design process just presented is quite simple, but for our purposes this is actually an advantage; it allows at least some hope of rigorously arguing for its correctness.

4.1. Correctness of Design Process

To show that the design process described in Figure 4 is correct, we first demonstrate that "derive_subs" satisfies its specification, and then use this result to show that "derive_design" is also correct. In both cases, the demonstrations are 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 begin by assuming that "refines(d,derive_design(d))", and then showing that "refines(d,derive_subs(d))". The latter is implied by the pre- and post-conditions for "derive_subs"; therefore, we will prove the following.

Theorem 1: {Q} derive_subs.body {R}
    where Q:true, R:refines(d,derive_subs)

    {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}.

The proof of this theorem has three parts. First, we show that "Q1" is true following the assignment "ds:=d"; second, we show that the if statement is correct with respect to "Q1" and "R1"; and third, we show that the assignment "derive_subs:=ds" establishes "R" from any state where "R1" holds. The first and third items are quite simple, but the second requires a lemma of its own.

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}
         by Lemma 1.1
    2.2) {Q1 ∧ d.cmd∈{do}} S2 {R1}
         dsgneq(d,ds) ∧ d.cmd∈{do} ∧
             refines(d.init,ds.init) ∧
             refines(d.dec,ds.dec) ∧
             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}.

The proof of this lemma follows the rule for if statements given in [14]; it has two conditions for correctness:

1) at least one of the guards must be true whenever the statement begins execution
2) execution of any command with an open guard must establish the post-condition.

The if statement has four alternatives. The proofs of the last three are quite simple, but the first requires a lemma.

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_0^k
           => 0≤0≤|d.cmds| ∧ dsgneq(d,ds) ∧
              (∀j:0≤j<0:
               refines(d.cmds[j].cmd,ds.cmds[j].cmd))
    2) {P ∧ k≠|d.cmds|} S {P}
       P ∧ k≠|d.cmds| ∧
           refines(d.cmds[k].cmd,ds.cmds[k].cmd)
           => P_{k+1}^k
           => 0≤k+1≤|d.cmds| ∧ dsgneq(d,ds) ∧
              (∀j:0≤j<k+1:
               refines(d.cmds[j].cmd,ds.cmds[j].cmd))
    3) P ∧ k=|d.cmds| => R
       dsgneq(d,ds) ∧
           (∀j:0≤j<|d.cmds|:
            refines(d.cmds[j].cmd,ds.cmds[j].cmd)) =>
                refines(d,ds) ;
    4) P ∧ 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 this lemma follows the rule for loops given in [14]; it has five conditions for correctness.

1) the invariant must be initialized correctly
2) the loop body must maintain the invariant
3) termination of the loop with the invariant true must guarantee the post-condition
4) the bound function must be non-negative while the loop is running
5) the loop body must decrease the bound.

The proof of all these conditions is straight forward, although somewhat involved. The second condition is the most complicated. In this case, we have taken the result of one part of the multiple assignment ("ds.cmds[k].cmd := derive_design(d,cmds[k].cmd)"), extracted the logical consequences from the post-condition of "derive_design" (refines(d.cmds[k].cmd, ds.cmds[k].cmd), and moved them to the left hand side of the implication. This greatly simplifies the remainder of the computation.

Our argument for the correctness of "derive_subs" is now complete. We can use this result to prove "refines(s,derive_design(d))". This is implied by the pre- and post-conditions for "derive_design"; therefore, it is equivalent to the following.

Theorem 2: {Q} derive_design.body {R}
where Q:true, R:refines(s,derive_design)

The proof of this theorem is reasonably straight forward, fairly lengthy, and similar to Theorem 1; therefore, it will not be given here. This proof, as well as any others not elaborated in this paper, may be found in [33].

Our argument for the correctness of the design process is now complete. 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 each cliche. In other words, we have argued that the process will produce correct designs if it uses correct cliches. We now turn to an examination of cliches and their correctness.

5. Cliches

We have constructed two distinct cliche libraries including a number of cliches which differ in many aspects [37]. However, for the purpose of verification only one distinction is important. For the simpler cliches, it was possible to describe and verify their designs in terms of implementation level constructs. On the other hand, the description and proof of the more complex cliches is at a higher level of abstraction. In this section we present an example cliche at each level and argue that its application preserves the "refines" relation.

5.1. Simple Cliche with Proof

The following is a high level representation of the "simple_assignment" cliche, which generates (multiple) assignment statements.

```
cliche simple_assignment is

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

end simple_assignment ;
```

The cliche states that the assignment "$Var_1..Var_N$:= $Soln_1..Soln_N$" is correct with respect to pre-condition "Q" and post-condition "R" 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; however, it leaves many details to the imagination.

Figure 5 shows a more precise description of "simple_assignment". The cliche is applicable to a specification only if "asgn_able" is true, and application in these situations is guaranteed to produce a design that satisifies "asgn_rule". To discuss this representation, understanding of the following symbol table 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 accessible symbols. The function "newsym" takes

```
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

a sequence of symbols and produces a new sequence that is identical to the original, except that the symbol names in the new list are unique.

To apply a general cliche for assignment statements, we must somehow find a list of expressions that make a logical formula true. In general this problem is undecidable [18]. Our purpose is not to consider the difficulties and technology of theorem proving; therefore, we will encapsulate the problem by defining the following routines.

```
function solvable(m,u:seq(sym);
                f:bool_expr):boolean is
    (∃ss:seq(expr(u)):
        provable(subst(ss,m,f)));

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

function solve(m,u:seq(sym) ;
                f:bool_expr):seq(expr);
    pre  can_solve(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. The function "can_solve" returns true (not necessarily if, but) only if its input is "solvable", while "solve" is called with a solvable problem and returns a solution. Notice that "solvable" represents the absolute solubility of a problem, while "can_solve" is a sound approximation to this function.

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 is
    a.cmd =asgn ∧
    provable( bool_expr(a.pre =>
                subst(a.e,a.v,a.post)));

function asgn_able(s:spec) : boolean is
    can_solve(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 function "asgn_rule" returns 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 can be solved.

The specification of "pre" requires that it return true if and only if "asgn_able(s)" is true. This is implied by the pre- and post-conditions for the function; therefore, we argue as follows.

Lemma 2: {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,
bool_expr(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
=> can_solve(ss',u',f') =
can_solve(ss',u',f')
by definition of asgn_able
therefore, {Q} pre.body {R}.

The proof of this lemma is straight forward, although reasonably involved. The weakest pre-condition for the four assignments that constitute the body of "pre" to establish the post-condition is first calculated, and then the pre-condition is shown to imply it.

The specification of "apply" requires that it return a design that satisfies "asgn_rule' if "asgn_able" is true of the specification given as input. The proof of this is similar to Lemma 2, and so will not be given here.

Lemma 3: asgn_able(s) =>
asgn_rule(simple_assignment.apply(s))

Lemma two can be rewritten as follows.

Lemma 2: simple_assignment.pre(s) = asgn_able(s)

Therefore, it directly follows that

Theorem 3: simple_assignment.pre(s) =>
asgn_rule(simple_assignment.apply(s))

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

This completes our argument for the correctness of this cliche. In this simple situation, we were able to verify the design at a level not far removed from the implementation, although we did encapsulate considerable complexity into external logic manipulation routines. Unfortunately, this level of analysis is not practical for many of the cliches we constructed. In these more complex situations, we verified the design at a

higher level of abstraction and then developed the implementation more informally.

5.2. More Complex Cliche with Proof

For example, Figure 6 shows a simplified representation of the "conditional_iteration_on_set" cliche, application of which can solve problems that require the use of a loop with an embedded conditional. The post-condition of the cliche states that "Var" is equal to the value of "Iop(Set,Cond)"; in other words, that the result is equal to the value of an iteration operator applied to a set with a certain condition.

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. "Lset" is initialized to "Set" and the result to "Id" (the identity element). The loop interates over all the items in "Set". If the item in question satisfies "Cond" then "Var" is set to "Op(Var,Lvar)"; otherwise, nothing is done.

This cliche can be applied if its pre- and post-conditions unify with the current specification and the identity element and result modification operator match one of the elements in a pre-computed table. Each entry in "iop_table" satisfies the following properties.

Property 1:

(Id,Op(Var,Lvar),Iop(Set,Cond)) ∈ iop_table =>

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 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. In a sense, we have encapsulated considerable complexity within this table; each entry requires a (possibly non-trivial) proof that it displays the necessary properties.

Figure 7 shows the fully annotated version of the cliche body which we will use to argue for its correctness.

```
cliche conditional_iteration_on_set is
    {Q}
    var Lset : set(Stype) ;
    var Lvar : Stype      ;
    Lset,Var := Set,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 6. *Conditional_Iteration_on_Set* Cliche

```
{Q}
var Lset : set(Stype) ;
var Lvar : Stype      ;
Lset,Var := Set,Id    ;
{Q': Var=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 7. Fully Annotated Cliche Body

Theorem 4: {Q} conditional_iteration_on_set.body {R}
where Q:true, R:Var=Iop(Set,Cond)

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

1) {Q} Lset,Var := Set,Id {Q'}
   Q => Id=Id ∧ Set=Set
2) {Q'} DO {R'} by Lemma 4
3) R' => R
   Lset⊆Set ∧ Var=Iop(Set-Lset,Cond) ∧ Lset={}
   => Var=Iop(Set,Cond)

To show that the cliche is correct, we prove that the initialization sets up the loop in the proper manner, that the loop is correct, and that correct termination of the loop ensures the post-condition for the routine is satisfied. The proofs of the first and third of these conditions are both simple and straight forward. The proof of the loop is a a bit more complicated; therefore, we will make it a lemma.

Lemma 4: {Q'} DO {R'}
where Q':Var=Id ∧ Lset=Set, R':P ∧ Lset={}

1) Q' => P
   Var=Id ∧ Lset=Set => $P^{Var,Lset}_{Id,Set}$
   Q' => Set⊆Set ∧ Id = Iop(Set-Set,Cond)
   Id = Iop({},Cond) by Property 1.1
2) {P∧B} S {P}
   {P∧B} {Q1'} S1' {Q1} S1 {R1} {P}
   P∧Lset≠{} => Q1'
   {Q1'} S1 {Q1} by Lemma 4.1
   {Q1} S1 {R1} by assumption
   R1 => P by Lemma 4.2
3) P∧¬B => R'
   P∧¬(Lset≠{}) => P∧Lset={}
4) P∧B => (t≥0)
   => |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}
   because S1 modifies only Var

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

The proof of this lemma is somewhat involved. In the first condition, the equalities from the left hand side of the implication (Var=Id, Lset=Set) are transformed into substitutions and then applied to the right hand side. The fifth condition relys on the fact that "S1" does not modify "Lset", and therefore can not change the value of the bound function. The second condition is the most complex; it assumes that the unknown in the loop body is completed correctly and uses two lemmas that we present separately.

Lemma 4.1: {Q1'} S1' {Q1}
where Q1':P ∧ Lset≠{} ∧ Lset=LSET ∧ Var=VAR,
Q1:Var=VAR ∧ LSET⊆Set ∧
VAR = Iop(Set-LSET,Cond) ∧
Lset=LSET-Lvar ∧ Lvar∈LSET

The proof of the first lemma is a rather routine matter of substitution and expansion, so we will not present it here. However, we should note that "Q1" simply names the current values of "Lset" and "Var" with the new constants "LSET" and "VAR" respectively. The proof of the second lemma is somewhat more interesting and demonstrates the necessity of Properties 1.2.1 and 1.2.2

Lemma 4.2: R1 => P
where R1:Q2 ∧ (¬Cond(Lvar) ∧ Var=VAR ∨
Cond(Lvar) ∧ Var=Op(VAR,Lvar)),
P: Lset⊆Set ∧ Var=Iop(Set-Lset,Cond)

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-Lset,Cond))
by Property 1.2.1
T1 ∧ T2 ∧ T4 => P2 by Property 1.2.2
P1 ∧ P2 => P
therefore, R1 => P

The proof of the loop is now complete, and with it our argument for the correctness of "conditional_iteration_on_set". We will take this as a demonstration that application of the cliche produces only designs that are totally correct with respect to their specifications and thereby preserves the "refines" relation. Although we were not able to verify this cliche at a level close to its implementation, we have significantly increased our confidence that it will function properly.

6. Summary and Conclusions

We have constructed a simulation of the formal design method described by Gries in his book *The Science of Programming* [33-37]. The system takes a pre- and post-condition specification and incrementally transforms it into a provably correct design using a library of *cliches*, complex knowledge structures representing commonly occurring situations.

The system is designed using guarded commands, and we have rigorously verified its correctness [33,36]. This argument has has two parts. First, we have shown that the simulation always produces correct designs if all the cliches in the library are valid, and second, we have demonstrated that the cliches in the library only produce designs which satisfy their

specifications.

The cliches are verified at two different levels of abstraction. For the simpler cliches, it was possible to describe and verify their designs in terms of implementation level constructs. Unfortunately, for many of the more complex cliches this was not practical. In these situations, we verified a more abstract version of the design and then developed the implementation informally. In both cases, significant complexity was encapsulated within external routines and precomputed tables.

Our experience with process verification has been both enjoyable and illuminating. Producing rigorous arguments of correctness forced us to think more carefully about the system we were constructing and probably resulted in a cleaner, more elegant structure. Both our designs and proofs went through a number of iterations before reaching the forms presented in this paper. Unfortunately, this process was reasonably time consuming; therefore, for the present such efforts will probably be restricted to smaller, more precisely defined processes.

7. References

1. Barstow, D. R., "Domain-Specific Automatic Programming", *IEEE Trans. Software Eng. SE-11, 11* (Nov. 1985), 1321-1336.
2. Bernot, G. and M. C. Gaudel, "Software Testing Based on Formal Specifications: a Theory and a Tool", *Software Eng. J. 6*, 6 (Nov. 1991), 387.
3. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proc. 9th Intl. Conf. Software Eng.*, 1987, 17-29.
4. Broy, M. and M. Wirsing, eds., *Methods of Programming: Selected Papers on the CIP-Project*, Springer-Verlag, New York, 1991.
5. Chen, W. and J. T. Udding, "Program Inversion: More Than Fun", *Science of Computer Programming 15*, 1 (1990), 1-13.
6. Cunningham, H. C. and G. C. Roman, "A UNITY-Style Programming Logic for Shared Dataspace Programs", *IEEE Trans. Parallel Distributed Systems 1*, 3 (July 1990), 365.
7. Delisle, N. and D. Garlan, "A Formal Specification of an Oscilloscope", *IEEE Software 7*, 5 (Sept. 1990), 29-36.
8. Dijkstra, E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *Comm. ACM 18*, 8 (Aug. 1975), 453-457.
9. Dijkstra, E. W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976.
10. Feather, M. S., "Constructing Specifications by Combining Parallel Elaborations", *IEEE Trans. Software Eng. 15*, 2 (Feb. 1989), 198-208.
11. Fickas, S. and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. Software Eng. 18*, 6 (June 1992), 470-482.
12. Futatsugi, K., J. Goguen, J. Meseguer and K. Okada, "Parameterized Programming in OBJ2", *Proc. 9th Intl. Conf. Software Eng.*, 1987, 51-60.
13. Gries, D., "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs", *IEEE Trans. Software Eng. SE-2*, 4 (Dec. 1976), 238-244.
14. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.
15. *Proc. 7th Intl. Software Process Workshop*, IEEE Computer Society Press, Los Alamitos, CA, 1991.
16. Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
17. Linger, R. C., H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.
18. Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, New York, 1984.
19. Lor, K. E. and D. M. Berry, "Automatic Synthesis of SARA Design Models From System Requirements", *IEEE Trans. Software Eng. 17*, 12 (Dec. 1991), 1229-1240.
20. Madhavji, N. H. and W. Schafer, "Prism - Methodology and Process-Oriented Environment", *IEEE Trans. Software Eng. 17*, 12 (Dec. 1991), 1270-1283.
21. Moriconi, M. S., "A Designer/Verifier's Assistant", *IEEE Trans. Software Eng. 5*, 4 (July 1979), 387-401.
22. Morzenti, A., D. Mandrioli and C. Ghezzi, "A Model Parametric Real-Time Logic", *ACM Trans. Programming Languages Systems 14*, 4 (Oct. 1992), 521.
23. Osterweil, L. J., "Software Processes Are Software Too", *Proc. 9th Intl. Conf. Software Eng.*, 1987, 2-13.
24. Perry, D. E., "Version Control in the Inscape Environment", *Proc. 9th Intl. Conf. Software Eng.*, 1987, 142-149.
25. Potts, C. and G. Bruns, "Recording the Reasons for Design Decisions", *Proc. 10th Intl. Conf. Software Eng.*, April 1988, 418-427.
26. Rich, C. and R. C. Waters, eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman Publishers, Los Altos, CA, 1986.
27. Shaw, M., *Alphard: Form and Content*, Springer-Verlag, New York, 1981.
28. Smith, D. R., "KIDS: a Semiautomatic Program Development System", *IEEE Trans. Software Eng. 16*, 9 (Sept. 1990), 1024-1043.
29. Sutton, S. M., D. Heimbigner and L. J. Osterweil, "Language Constructs for Managing Change in Process-Centered Environments", *Proc. 4th ACM SIGSOFT Symp. Software Development Environments*, Dec. 1990, 206-217.
30. Taylor, R. N., F. C. Belz, L. A. Clarke, L. J. Osterweil, R. W. Selby, J. C. Wileden, A. Wolf and M. Young, "Foundations for the Arcadia Environment Architecture", *Proc. Symp. Practical Software Development Environments*, 1988, 1-13.
31. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", *Proc. 10th Intl. Conf. Software Eng.*, April 1988, 344-354.
32. Terwilliger, R. B. and R. H. Campbell, "PLEASE: Executable Specifications for Incremental Software Development", *J. Systems Software 10*, 2 (Sept. 1989), 97-112.
33. Terwilliger, R. B., "A Process Program for Gries/Dijkstra Design", Rprt. CU-CS-566-91, Dept. Comp. Sci., U. Colorado Boulder, Dec. 1991.
34. Terwilliger, R. B., "Simulating the Gries/Dijkstra Design Process", *Proc. 7th Knowledge-Based Software Engineering Conf.*, Sept. 1992, 144-153.
35. Terwilliger, R. B., "Evolving Tools to Support the Gries/Dijkstra Design Process", Rprt. CU-CS-631-92, Dept. Comp. Sci., U. Colorado Boulder, Dec. 1992.
36. Terwilliger, R. B., "A Second Simulation of the Gries/Dijkstra Design Process", Rprt. CU-CS-618-92, Dept. Comp. Sci., U. Colorado Boulder, Oct. 1992.
37. Terwilliger, R. B., "An Evolving Simulation of the Gries/Dijkstra Design Process", Rprt. CU-CS-632-92, Dept. Comp. Sci., U. Colorado Boulder, Dec. 1992.
38. Woodcock, J. C. P., "Structuring Specifications in Z", *Software Eng. J. 4*, 1 (Jan 1989), 51.