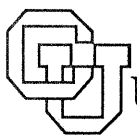
Evolving an Understanding of the Gries/Dijkstra Design Process

Robert B. Terwilliger

CU-CS-624-92 November.1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Evolving an Understanding of the Gries/Dijkstra Design Process

Robert B. Terwilliger

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

ABSTRACT

We are applying a three part, investigative approach to the formal design method described by Gries in his book *The Science of Programming*. First, we walkthrough the process on a number of small problems; second, we construct a simulation program which duplicates the designs produced by the walkthroughs; and third, we produce a process program that supports human application of the method. We have completed two iterations of the three step procedure including construction of a simulation engine and adaptation of a language-oriented editor to support the method. In this paper, we describe the design process using cliches: complex knowledge structures representing commonly occurring situations.

1. Introduction

Some have suggested that *formal methods* can enhance both the software specification and design processes [1, 13, 16, 17]. For example, Dijkstra and Gries [4, 5, 10, 11] 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 correctly apply it, determine if it has been applied properly, or evaluate it for effectiveness.

One approach to gathering information about development processes is the use of *walkthroughs* and *inspections* [6, 7, 32, 34]. In these situations, a software item or the method used to produce it is presented to a group who evaluate it according to an appropriate set of criteria. Even more information can be gathered through *process programming* [12, 18, 23, 24]: describing software processes using programming language constructs and notations. To a certain extent, this is similar to previous work on *automatic programming* [9, 14, 19, 21]. An important contribution of these projects is the notion of *cliche* (or *plan* or *schema*): a complex knowledge structure representing a commonly occurring situation.

We are applying a three part, investigative approach to the formal design method described by Gries in his book *The Science of Programming*. First, we walkthrough, in other words hand simulate, the process on a number of small problems. This produces an increased understanding of the method as well as a suite of example designs. Second, we produce a program that simulates the design process discovered during the walkthroughs; ideally, it should be able to recreate the suite of designs previously produced. Third, we produce a process program that supports human application of the method.

We have currently completed two iterations of our three part approach on the Gries/Dijkstra process [27, 28, 30, 31]; our understanding of the method has evolved significantly during the time we have spent studying it. After performing the initial walkthroughs, we viewed the design method as a linear sequence of small, relatively independent steps; however, our attempts to automate the process based on this model met with no success. Instead, both our simulation and process programs are based on a library of relatively large cliches representing solutions to common design problems.

Our first iteration cliches combine a number of what were considered distinct steps during the initial walkthroughs into a single unit. Our second iteration cliches combine several steps from the initial walkthroughs, but not nearly as many as the first iteration cliches do. They also better reflect the process described by Gries [11]. We believe our second iteration cliches are a significant step towards an accurate and detailed description of the method.

In the remainder of this paper, we describe the results of our investigations in more detail. In section two we describe the Gries/Dijkstra design process as we understood it after our initial walkthroughs, and in section three we describe the architecture used in both our simulation and process programs. In section four, we give an example of our first iteration cliches and their use in design derivation, and in section five we do the same for the second iteration. Finally, in section six, we summarize and draw some conclusions from our experience.

2. Gries/Dijkstra Design

Figure 1 shows a pictorial representation of the process described in [11] as we understood it after completing our initial walkthroughs. The design derivation process uses stepwise refinement to transform pre- and post-condition specifications written in first-order predicate logic into verified programs written using guarded commands. At each step, *strategies* determine how the current partial program is to be elaborated, and *proof rules* are used to verify the correctness of the transformation.

For example, loops are specified 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 is an upper limit on the number of iterations remaining. The proof rule for loops has five conditions for correctness [11]:

1) the invariant must be initialized correctly
2) execution of 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 greater than zero while the loop is running
5) execution of the loop body must decrease the bound.

A loop development strategy closely parallels the proof rule; each step in the strategy suggests an action, whose result can be verified using one of the items from the proof rule.

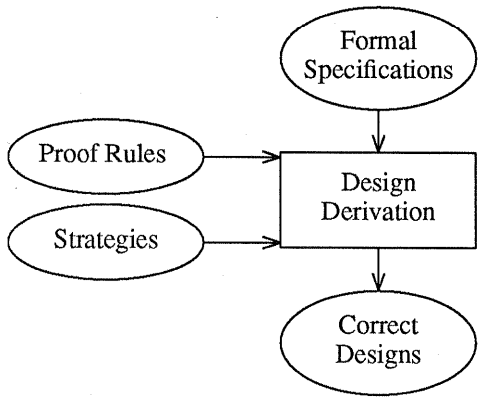


Figure 1. Gries/Dijkstra Design Process

To develop a loop given a pre- and post-condition:

1) weaken the post-condition to obtain an invariant and develop a statement to initialize it
   - check item one of proof rule
2) develop a loop guard
   - check item three of proof rule
3) develop a bound function
   - check item four of proof rule
4) develop a statement that decreases the bound
   - check item five of proof rule
5) develop a statement that restores the invariant
   - check item two of proof rule.

The first step is to develop an invariant by *weakening* the post-condition; in other words, the invariant is an easier to satisfy version of the desired result. There are at least three ways to weaken the post-condition: delete a conjunct, replace a constant by a variable, and enlarge the range of a variable. An initialization for the loop must also be developed and then checked using item one of the proof rule. A good invariant has a simple initialization; if none can be found then the invariant is discarded and another weakening method is tried.

The second step is to develop a loop guard; in general, the guard will follow directly from the weakening method. For example, if the invariant is created by deleting a conjunct from the post-condition, then the guard will be the negation of the deleted conjunct. The guard's correctness is verified using item three of the proof rule.

The third step is to develop a bound function; this is done by discovering a property that should be decreased by each iteration of the loop body and then formalizing it. Item three of the proof rule is used to verify that the invariant, guard and bound are consistent.

The fourth step is to develop a statement that decreases the bound; this will form the first part of the loop body. Item five of the proof rule is used to check that the bound is diminished. The final step is to develop a statement that restores the invariant after the statement which decreases the bound has invalidated it. Item two of the rule is used to check that the loop body maintains the invariant.

The above strategy is a step by step procedure for transforming a pre- and post-condition into a provably correct loop. Each step in the strategy is checked using the appropriate item from the proof rule; the loop and its proof of correctness are developed simultaneously. We can further clarify this process in the context of an example.

Kemmerer's Library problem has received considerable attention in the software engineering literature [15,33]. 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 [28] consists of a single module that encapsulates the database and provides an entry routine for each transaction. The state of the module is modeled abstractly using high-level data types, and the entry routines are specified using pre- and post-conditions.

For example, consider the "who_has" function, which returns the set of all users who currently have a particular book checked out.

```
function who_has( s:user ; b:book
                ) : set(user);
    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 : user ;
               item : book ;
               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. The set "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 currently have the book in question checked out.

Using the Gries/Dijkstra process, design might proceed as follows. First, we replace the constant "users" in the post-condition with the variable (expression) "users-usrs" to obtain the following invariant.

```
usrs⊆users ∧
who_has = {u∈users-usrs:
                corec(u,b)∈checks}}
```

The variable "usrs" holds the users still to be examined. At any point during the loops execution, "usrs" is a subset of "users" and "who_has" contains the set of all users already examined who have the book in question checked out. The invariant can be initialized with the simultaneous assignment "who_has,usrs:={},users", and this satisfies item one of the proof rule.

We now develop a guard for the loop. Item three of the proof rule tells us that the negation of the guard and the invariant together must imply the post-condition. Since we created the invariant from the post-condition by replacing a constant with a variable, the guard is just that the variable does not equal the constant. In our example, the loop should stop when "users-usrs" is equal to "users"; therefore, the guard is "usrs≠{}", and it satisfies item three.

We now develop a bound function. In our example, each iteration should decrease the number of elements in "usrs"; we formalize this as "|usrs|" and check that it satisfies item four of the proof rule. We now have the following.

```
{Q: true}
var usrs : set(user) ;
who_has,usrs:={},users ;
{inv P:usrs⊆users ∧
       who_has = {u∈users-usrs:
                     corec(u,b)∈checks}}
{bnd t: |usrs|}
do usrs≠{} → < S > od
{R: who_has =
       {u∈users:corec(u,b)∈checks}}
```

We now develop a statement that decreases the bound. We declare a local variable "usr" of type "user" and use the statement sequence "choose(usrs,usr); usrs:=usrs-usr" is to remove an element from "usrs". Using item five of the proof rule we verify that this decreases "|usrs|", and it becomes the first part of the loop body.

Finally, we must develop a statement that restores the invariant after the previous command has invalidated it. There are two cases; therefore, the body contains an if statement. If "usr" has the book in question checked out ("corec(usr,b) ∈ checks"), then they must be added to the result set; otherwise, nothing needs to be done.

Using item two of the proof rule we verify that the body of the loop maintains the invariant, and we have now produced the complete design shown in Figure 2. Since we ensured that all five items of the appropriate proof rule were satisfied as we constructed the loop, we have already proven it correct.

As we have just described it, the design process consists of a single level; the developer proceeds through a sequence of small, relatively independent steps to produce a final design. While we found this model adequate for performing walkthroughs, our attempts to automate the process based on it met with no success. Instead, both our simulation and process programs are based on a library of relatively large cliches representing solutions to common design problems.

```
{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 2. Completed *Who_Has* Design

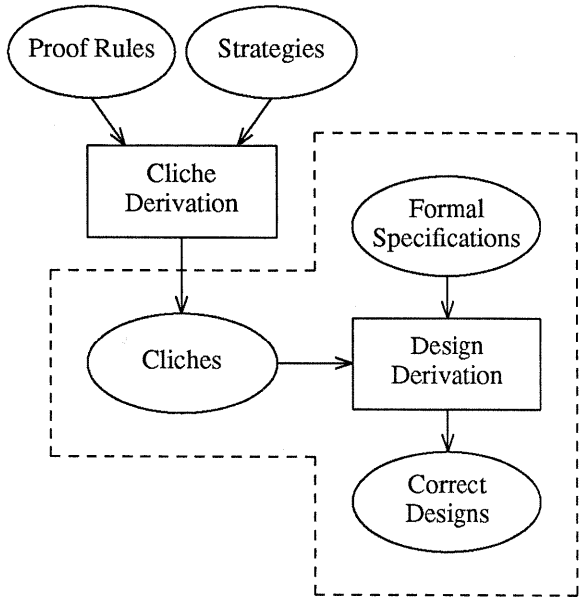


Figure 3. Design Process as Automated

3. Automation Architecture

Figure 3 shows a pictorial representation of the design process implemented in both our simulation and process programs. It has two levels. At the lower level, a design derivation sub-process transforms formal specifications into correct designs using a library of pre-verified cliches. On the upper level, a cliche derivation sub-process uses strategies and proof rules to construct and verify cliches for the library.

The correctness of a final design depends on the correctness of the cliches used in its derivation; therefore, 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 performed by a human. On the other hand, cliche application is reasonably easy and is performed repeatedly by the machine.

We have created both simulation and process programs based on this architecture. The input to our simulation [27, 29-] is a pre- and post-condition for the unit to be constructed, as well as a 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. Application of a cliche may 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.

Our process program [31] is based on ISLET, a language-oriented program/proof editor originally developed as part of the ENCOMPASS project [25, 26]. The editor provides commands to add, delete and refine constructs. As the program/proof is incrementally constructed, the syntax and semantics are constantly checked, and verification conditions are generated when necessary. Our process program also allows a designer to perform the technique described by the simulation; they can browse a library of pre-verified cliches, select a promising candidate, and apply it to a specification.

The architecture presented in this section is quite simple, but adequate for its purpose. The use of a library of cliches allows the process to be separated into a difficult, possibly intuitive part performed by a human, and a simple, mechanistic part performed by a machine. The simplicity of the design derivation sub-process implies that the power the overall process depends on the complexity of the cliches in its library.

4. Cliches - First Iteration

The number of cliches that can be used in the design process is literally infinite; however, a single example is sufficient to transmit the flavor of our first iteration. The real thrust of this effort was the construction of relatively powerful cliches that combined a number of steps from the walkthroughs into a single unit.

For example, Figure 4 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. The post-condition for the cliche states that the result variable, "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.

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 the identity element, "Id". The loop interates over all the items in "Set". If the item in question satisfies "Cond" then "Var" is set to "Op(Var,Lvar)". The correct result has been calculated when all the items have been considered.

```
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 4. *Conditional_Iteration_on_Set* Cliche

In general, knowing when this cliche can be applied is difficult; how can we determine which operators are allowed, what the identity elements are, and how the result is updated when an appropriate element is found? 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" contains the above information and is guaranteed to satisfy 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.

A number of small cliches are also necessary to handle the details of design derivation. For example, the "simple_assignment" cliche generates (multiple) assignment statements.

```
cliche simple_assignment is

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

end simple_assignment ;
```

This cliche states that a simultaneous assignment is correct if the pre-condition implies the post-condition with the left hand side of the assignment substituted for the right. This cliche presents certain difficulties in that a completely general implementation of its applicability test involves an undecidable problem [8,17]. Our solution is to use a somewhat less general, but much less expensive test.

The "simple_if_then_else" cliche is somewhat more complex; it generates generates two branch if-then-else statements where one guard is the logical negation of the other.

```
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 ;
```

A number of small, necessary, but relatively uninteresting cliches were developed during our first iteration, as well as some relatively powerful cliches which combined a number of steps from the initial walkthroughs into a single unit. The first iteration cliches are presented in more detail and verified in [27]. In line with our two level process model, cliche application was made as easy as possible, even at the cost of more effort in cliche construction. Where appropriate, specialized tables are used to "cache" computationally expensive items.

Although the framework we constructed is minimal, it does have enough power to duplicate some of the designs produced by a human.

4.1. Example Design Derivation

Let us reconsider the "who_has" function presented in section two. We can see that the "conditional_iteration_on_set" cliche is applicable to the specification.

```
{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 with "Var" = "who_has", "Iop" = "set_of_all", "Set" = "users", and "Cond" = "corec(u,b)∈ checks".

Figure 5 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. 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. The "simple_if_then_else" cliche is applicable, and instantiation produces the following design for the loop body.

```
{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 5. Instantiated Cliche

```
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
```

For each user, the loop 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 design of the loop body is completed by applying the "simple_assignment" cliche twice. As a final flourish, "who_has:=who_has" is optimized to "skip", thereby reproducing the hand derived design shown in Figure 2. This development is presented in more detail in [27, 30].

5. Cliches - Second Iteration

In the first iteration of our investigative process, we concentrated on constructing a system that could reproduce some of the designs created by a human; we did not devote a lot of effort to developing a process

that closely matched the descriptions of the original authors [4, 5, 10, 11], or the detailed activities of a human using pencil and paper. In the second iteration of our three part approach, we produced a new library of cliches that more closely reflect the method described by Gries in his book *The Science of Programming* [11].

The first iteration cliches combine a large number of steps from the walkthroughs into a single unit. In general, the second iteration cliches combine fewer steps than the first iteration cliches do; therefore, it may take more cliche applications to transform a specification into a completed design.

As an illustration, let us again consider the "who_has" function. In this case, three second iteration cliches replace "conditional_iteration_on_set" from the first iteration. The "simple_replace_constant" cliche replaces a constant in a post-condition with a variable expression to create a loop invariant; the "decrease_then_restore" cliche decomposes a loop body into a statement which decreases the bound function and another that restores the invariant; and the "conditional_operation" cliche generates a loop body with an embedded conditional.

Figure 6 shows a simplified representation of the "simple_replace_constant" cliche. This cliche replaces a constant in a post-condition with a variable expression to create a loop invariant. Specifically, the cliche states that a loop with initialization "S0" and body "S1" is correct if: "C" is a constant in the post-condition, the tuple "(C,E,V,Rng)" is an element of the relation "mkvariable", and the loop invariant is equivalent to "Rng" conjoined with the result of substituting "E" for "C" in the post-condition. Here, "V" is a variable, "E" is an expression in "C" and "V", and "Rng" is a boolean expression restricting the range of "V".

The applicability test for this cliche is simplified by caching computationally expensive items. "Mkvariable" can be viewed both as a relation and as a Prolog-style procedure. Given a constant "C", it produces an expression "E" to be substituted in the post condition, as well as the appropriate declaration and range. Each tuple in (or produced by) "mkvariable" is guaranteed to satisfy the following property.

```
(C,E,V,Rng) ∈ mkvariable =>
        Rng ∧ E≠C => (|C|-|E|) ≥ 0
```

This is exactly what is necessary to prove the correctness of the cliche body and thereby ensure that all the designs produced by the cliche will satisfy their specifications.

Figure 7 shows a simplified representation of the "decrease_then_restore" cliche. This cliche decomposes a loop body into a statement which decreases the bound function and another that restores the invariant. The applicability test for this cliche requires that the tuple "(T,V,E,S0,F,D)" is an element of the relation "decrease_bnd", and that "S1" does not modify any of the variables referenced in "T".

```
cliche simple_replace_constant is
    {Q}
    var V ;
    < S0 > ;
    {inv P: Rng ∧ R[[ C / E ]]}
    {bnd t: |C|-|E|}
    do B: E≠C →
        {Q1: P ∧ B ∧ t=t1}
        < S1 > ;
        {R1: P ∧ t<t1}
    od
    {R}
if
    C ∈ constants(R) ∧
    (C,E,V,Rng) ∈ mkvariable ;

end simple_replace_constant ;
```

Figure 6. *Simple_Replace_Constant* Cliche

```
cliche decrease_then_restore is

    {Q: P ∧ B ∧ T=t1}
    var D ;
    < S0 >
    {Q1: P[[ V / E ]] ∧ F}
    < S1 >
    {R1: P}
    {R: P ∧ T<t1}

if
    (T,V,E,S0,F,D) ∈ decrease_bnd ∧
    modify(S1) ∩ use(T) = ∅ ;

end decrease_then_restore ;
```

Figure 7. *Decrease_Then_Restore* Cliche

Here, "T" is the bound function for the loop; "V" is a variable that will be modified to decrease "T"; "S0" is a statement that modifies "V"; "E" is an expression reflecting the modifications to "V"; "F" is a formula stating additional facts concerning the modification; and "D" is a declaration of the iteration variable.

To prove the correctness of the cliche body, and thereby guarantee the correctness of all the designs produced from it, every tuple "(T,V,E,S0,F,D)" in the "decrease_bnd" relation must satisfy the following three properties .

1. {T=t1} S0 {T<t1}
2. {true} S0 {F}
3. {W} S0 {(W) $^{V}_{E}$} for any W

Figure 8 shows a simplified representation of the "conditional_operation" cliche. Application of this cliche generates a loop body with an embedded conditional. The post-condition for "conditional_operation" is the invariant for the enclosing loop. It states that "Lset" is a subset of "Set", and that "Var" is equal to the value of an iteration operator applied to the difference of "Set" and "Lset" with a certain condition.

The pre-condition for the cliche requires that the loop invariant hold with "LSET" substituted for "Lset", and that "Lset" be the result of removing "Lvar" from "LSET". Here, "Lset" is a set containing all the items still to be considered; "LSET" is the value of "Lset" from the previous iteration; and "Lvar" is the item currently being processed.

```
cliche conditional_operation is

    {Q: LSET⊆Set ∧
        Var = Iop(Set-LSET,Cond) ∧
        Lset=LSET-Lvar ∧ Lvar∈LSET}
    {Q1: Var=VAR}
    < S1 >( Var:inout Rtype ) ;
    {R1: ( ¬Cond(Lvar) ∧ Var=VAR ∨
          Cond(Lvar) ∧ Var=Op(VAR,Lvar) )}
    {R: Lset⊆Set ∧
        Var = Iop(Set-Lset,Cond)}
if
    (Iop(Set,Cond),Op(Var,Lvar))
        ∈ iteration_ops ;

end conditional_operation ;
```

Figure 8. *Conditional_Operation* Cliche

"Conditional_operation" can be applied to a specification if the pre- and post-conditions unify and the tuple "(Iop(Set,Cond), Op(Var,Lvar))" is an element of "iteration_ops". The cliche's proof of correctness requires that every tuple in "iteration_ops" satisfies two properties. These are exactly those designated 2.1 and 2.2 in the description of "iop_table" for the "conditional_iteration_on_set" cliche in section four.

The cliches presented in this section differ significantly from those developed in the first iteration of our investigative process. They more closely match the process described in [11], and in general combine fewer of what were considered independent steps in our initial walkthroughs. The second iteration cliches are discused in more detail and verified in [29]. Since they are in some sense "smaller", more applications may be required to produce a complete algorithm from a specification; however, they are still adequate to the task of reproducing the some of the designs created by a human.

5.1. Example Design Derivation

Let us again consider the "who_has" function first presented in section two. The "simple_replace_constant" cliche is applicable to the initial specification. The constant "users" is selected from the post-condition; the "mkvariable" relation produces an expression "users-usrs" and range "usrs⊆users"; and substitution yields the following invariant.

```
usrs⊆users ∧
who_has = {u∈users-usrs:
                corec(u,b)∈checks}
```

Instantiation of the guard yields "users-usrs≠users", which simplifies to "usrs≠{}" under the assumption that "usrs" is a subset of "users". Similarly, the bound function instantiates to "|users|-|users-usrs|", which simplifies to "|usrs|". Therefore, application of "simple_replace_constant" produces the partial design shown in Figure 9.

Application of the "simple_assignment" cliche produces a loop initialization, and the "decrease_then_restore" cliche is applied to the specification of the loop body. The "decrease_bnd" relation produces a bound "|usrs|", variable "usrs", expression "USRS", statement "choose(usrs,usr); usrs:=usrs-usr", and formula "usrs=USRS-usr ∧ usr∈USRS". Therefore, instantiation of "decrease_then_restore" produces the following body for the loop.

```
choose(usrs,usr); usrs:=usrs-usr;
{Q1:USRS⊆users ∧
    who_has = {u∈users-USRS:
                    corec(u,b)∈checks} ∧
    usrs=USRS-usr ∧ usr∈USRS}
< S >(who_has:inout set(user));
{R1:usrs⊆users ∧
    who_has = {u∈users-usrs:
                    corec(u,b)∈checks}}
```

The "conditional_operation" cliche is applicable to the remaining specification, and instantiation produces the following.

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

The rest of the derivation proceeds exactly as described in section four; a single application of "simple_if_then_else" and two applications of "simple_assignment" reproduce the hand derived design shown in Figure 2. This development is presented in more detail in [29].

```
{Q: true}
var usrs : set(user) ;
< S0 >(who_has,usrs:inout set(user)) ;
{inv P:usrs⊆users ∧
       who_has = {u∈users-usrs:
                    corec(u,b)∈checks}}
{bnd t: |usrs|}
do usrs≠{} →
    {Q1:usrs⊆users ∧
       who_has = {u∈users-usrs:
                    corec(u,b)∈checks} ∧
        usrs≠{} ∧ |usrs|=t1} ;
    < S1 >(who_has:inout set(user));
    {R1:usrs⊆users ∧
       who_has = {u∈users-usrs:
                    corec(u,b)∈checks} ∧
        |usrs|<t1} ;
od
{R: who_has =
       {u∈users:corec(u,b)∈checks}}
```

Figure 9. Instantiated Cliche

6. Summary and Conclusions

We have been applying a three part, investigative approach to the formal design method described by Gries in his book *The Science of Programming* [11]. First, we walkthrough, in other words hand simulate, the process on a number of small problems. This produces an increased understanding of the method as well as a suite of example designs. Second, we produce a program that simulates the design process discovered during the walkthroughs; ideally, it should be able to recreate the suite of designs previously produced. Third, we produce a process program that supports human application of the method.

We have currently completed two iterations of our three part approach on the Gries/Dijkstra process [27, 28, 30, 31]; our understanding of the method has evolved significantly during the time we have spent studying it. After performing the initial walkthroughs, we viewed the design method as a linear sequence of small, relatively independent steps; however, our attempts to automate the process based on this model met with no success. Instead, both our simulation and process programs are based on a library of relatively large cliches representing solutions to common design problems.

This view of the process has two levels. At the lower level, a design derivation sub-process transforms formal specifications into correct designs using a library of cliches. On the upper level, a cliche derivation sub-process uses strategies and proof rules to construct and verify cliches. 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 performed by a human. On the other hand, cliche application is reasonably easy and is performed repeatedly by the machine.

The question of how a human performs Gries/Dijkstra design using pencil and paper is beyond the scope of our work. It might be argued that even when someone performs a linear sequence of steps as described in section two, they are relying on a (possibly sub-conscious) cliche. On the other hand, some might say that the cliches are just a convenient way to store information that can be easily rederived when needed. As we understand it, the use of cliches is supported by work on the psychological aspects of programming [2, 3, 20, 22].

Our first iteration cliches combine a number of what were considered distinct steps during the initial walkthroughs into a single unit. Our second iteration cliches combine several steps from the initial walkthroughs, but not nearly as many as the first iteration cliches do. They also better reflect the process

described by Gries [11]. We believe our second iteration cliches are a significant step towards an accurate and detailed description of the method.

7. References

1. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proc. 9th Intl. Conf. Software Eng.*, 1987, 17-29.
2. Curtis, B., ed., *Tutorial: Human Factors in Software Development (Second Edition)*, IEEE Computer Society, Washington, D.C., 1985.
3. Davies, S. P., "The Role of Notation and Knowledge Representation in the Determination of Programming Strategy", *Cognitive Science 15* (1991), 547-572.
4. Dijkstra, E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *Comm. ACM 18*, 8 (Aug. 1975), 453-457.
5. Dijkstra, E. W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976.
6. Fagan, M. E., "Advances in Software Inspections", *IEEE Trans. Software Eng. SE-12, 7* (July 1986), 744-751.
7. Freedman, D. P. and G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, Little, Brown and Company, Boston, 1982.
8. Godel, K., "Uber Formal Unentscheidbare Satze der Principia Mathematica und Verwandter Systeme I", in *From Frege to Godel*, Heijenoort, J. (editor), Harvard U. Press, Cambridge, Mass., 1967.
9. Goldberg, A. T., "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques", *IEEE Trans. Software Eng. SE-12, 7* (July 1986), 752-768.
10. 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.
11. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.
12. *Proc. 7th Intl. Software Process Workshop*, IEEE Computer Society Press, Los Alamitos, CA, 1991.
13. Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
14. Kant, E., "Understanding and Automating Algorithm Design", *IEEE Trans. Software Eng. SE-11, 11* (Nov. 1985), 1361-1374.
15. Kemmerer, R. A., "Testing Formal Specifications to Detect Design Errors", *IEEE Trans. Software Eng. SE-11, 1* (Jan. 1985), 32-43.
16. Linger, R. C., H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.
17. Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, New York, 1984.
18. Osterweil, L. J., "Software Processes Are Software Too", *Proc. 9th Intl. Conf. Software Eng.*, 1987, 2-13.
19. Rich, C. and R. C. Waters, eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman Publishers, Los Altos, CA, 1986.
20. Rist, R. S., "Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Experienced Student Programmers", *Human-Computer Interaction 6*, 1 (1991), 1-46.
21. Smith, D. R., "KIDS: a Semiautomatic Program Development System", *IEEE Trans. Software Eng. 16*, 9 (Sept. 1990), 1024-1043.
22. Soloway, E. and K. Ehrlich, "Empirical Studies of Programming Knowledge", *IEEE Trans. Software Eng. SE-10*, 5 (1984), 595-609.
23. 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.
24. 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.
25. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", *Proc. 10th Intl. Conf. Software Eng.*, April 1988, 344-354.
26. Terwilliger, R. B. and R. H. Campbell, "ENCOMPASS: an Environment for the Incremental Development of Software", *J. Systems Software 10*, 1 (July 1989), 41-53.
27. Terwilliger, R. B., "A Process Program for Gries/Dijkstra Design", Rprt. CU-CS-566-91, Dept. Comp. Sci., U. Colorado Boulder, Dec. 1991.
28. Terwilliger, R. B., "A Formal Specification and Verified Design for Kemmerer's Library Problem", Rprt. CU-CS-562-91, Dept. Comp. Sci., U. Colorado Boulder, Dec. 1991.
29. 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.
30. Terwilliger, R. B., "Simulating the Gries/Dijkstra Design Process", *Proc. 7th Knowledge-Based Software Engineering Conf.*, Sept. 1992, 144-153.
31. Terwilliger, R. B., "Towards Tools to Support the Gries/Dijkstra Design Process", Rprt. CU-CS-594-92, Dept. Comp. Sci., U. Colorado Boulder, May 1992.
32. Weinberg, G. M. and D. P. Freedman, "Reviews, Walkthroughs, and Inspections", *IEEE Trans. Software Eng. SE-10, 1* (Jan. 1984), 68-72.
33. Wing, J. M., "A Study of 12 Specifications of the Library Problem", *IEEE Software 5*, 4 (July 1988), 66-76.
34. Yourdon, E. N., *Structured Walkthroughs*, Yourdon Press, New York, 1989.