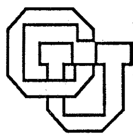
Simulating the Gries/Dijkstra Design Process

Robert B. Terwilliger

CU-CS-588-92 April 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Simulating 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 investigating software design processes using a three part approach. For a design method of interest, we first perform walkthroughs on a number of small problems. Second, we construct a simulation program which duplicates the designs produced by the walkthroughs, and third, we a construct a process program that supports human application of the method. We have been pursuing this 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. In this paper, we describe a simulation program for this process that is based on a library of cliches describing solutions to common programming problems. We have constructed a prototype implementation in Prolog and used it to generate a number of example designs.

1. Introduction

The development of software consumes a significant portion of our society's economic resources [9]. In the traditional, or waterfall lifecycle model, the *design phase* defines the overall structure of the system and the basic methods it will use to perform its functions. This phase is important because design quality significantly impacts both the performance and maintainability of the final system. There are many different techniques for software design [13]; unfortunately, at present is it difficult to either correctly apply a design technique, 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* [8, 12, 24, 29, 42, 44]. In these situations, a software item or the process used to produce it is presented to a group of engineers who evaluate it according to an appropriate set of criteria. Although the distinction is somewhat arbitrary, we can differentiate between a walkthrough, which is guided by the structure of the item or process in question, and an inspection, which is guided by an apriori, fixed set of issues to be addressed. Both walkthroughs and inspections can be used to evaluate both software designs and design methods.

Process programming is an ambitious approach to improving the effectiveness of software development [16-18, 28, 36, 37]. The basic idea is simple: describe software processes using programming language constructs and notations. Ultimately, this should allow software development to become automated, and process execution to be monitored, evaluated, and tuned for maximum efficiency. Unfortunately, the processes being used currently are typically not well defined or understood; fortunately, process programming can also enhance process understanding.

To a certain extent, process programming overlaps with previous work on knowledge-based software engineering [1, 10, 11, 14, 19, 21, 22, 30, 32-34]. There are many different approaches to this general problem. For example, a *very high-level language* presents a non-interactive interface to the programmer; it simply extends conventional programming languages with high-level control and data structures. On the other hand, an *intelligent assistant* is highly interactive; it performs routine chores, while leaving all important decisions to the human. In either case, many projects include some notion of *cliche* (or *plan* or *schema*): a complex knowledge structure representing a commonly occurring situation.

We are investigating software design processes using a three part approach. For a design method of interest, we first 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. The result of the third step is a new, partially automated process that can then be subjected to another iteration of the entire three step approach.

From our point of view, formal methods are an interesting class of specification and development processes [2, 3, 20, 25-27]. They are precisely defined and show significant variations, even when applied to

small problems. We have been applying our three step approach to the formal design process developed by Dijkstra and Gries [6, 7, 15]. This method 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. We have currently completed steps one (walkthrough) and two (simulation) on this method [40, 41].

In this paper we present our simulation program for the Gries/Dijkstra process. The system is based on a library of cliches describing solutions to common programming problems. Execution 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 simulation is completely non-interactive and can be viewed as a translator for a very high-level language. We are now constructing an interactive process program that more closely resembles an intelligent assistant.

In the remainder of this paper, we describe this simulation program in more detail. In section two we give some background on the Gries/Dijkstra design process and give an example walkthrough. In section three we present our simulation program and discuss how it differs from the method as presented by Gries [15]. In section four we present some cliches used by the program to generate verified designs from specifications, and in section five we describe how the simulation generates the design produced by the walkthrough in section two. 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 design process developed by Dijkstra and Gries [6, 7, 15]; our view of the method is based primarily on [15]. 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. Since each step is verified before the next if applied, errors are detected sooner and corrected at lower cost. The process is in some sense general, but is most applicable to problems in algorithm design.

For example, Kemmerer's Library problem has received considerable attention in the software engineering literature and has been formally specified a number of times [23, 43, 45]. The problem is concerned with a small library database that provides both query and update transactions to library staff and users.

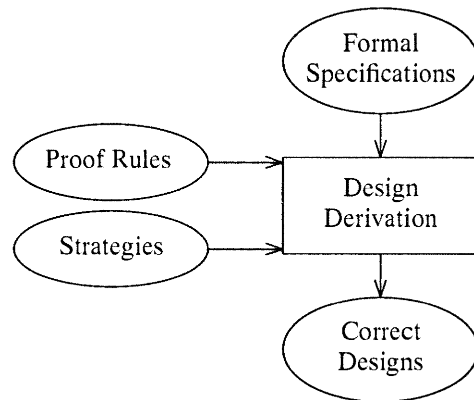


Figure 1. Gries/Dijkstra Design Process

The architectural design for our solution [41] 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: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 currently have the book in question checked out.

Using the Gries/Dijkstra method, design might proceed as follows. First, we notice that our programming language does not contain an operator to compute a subset based on a selection predicate; therefore, we use a loop to iterate over the array. We specify this 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 is an upper limit on the number of iterations remaining.

The proof rule for loops has five conditions for correctness [15]. Three are concerned with partial correctness:

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,

and two are used to insure termination:

4) the bound function must be greater than zero while the loop is running
5) execution of the loop body must decrease the bound.

In our example, we construct the loop and simultaneously verify its correctness using this rule.

First, we develop the 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. In this case, we replace the constant "users" with the variable (expression) "users-usrs" to obtain the following invariant.

```
{inv P: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 is initialized with the simultaneous assignment "`who_has,usrs:={},users`"; this satisfies item one of the proof rule.

We now develop a guard for the loop body. Item three of the proof rule for loops 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 loop 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 loop guard is "usrs≠{}", and it satisfies item three.

We now develop a bound function for the loop. We do this by discovering a property that should be decreased by each iteration of the loop body and then formalizing it. In our example, each iteration should decrease the number of elements in "usrs". We formalize this as "|usrs|". We check that this function satisfies item four of the proof rule: "|usrs|" is greater than zero as long as the loop is running. 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}}
```

Item five of the proof rule for loops requires that the body of the loop decrease the bound function. The simplest way to accomplish this to remove an element from "usrs". If we declare a local variable "usr" of type "user" then we can accomplish this as follows.

```
choose(usrs,usr) ; usrs:=usrs-usr ;
```

Item two of the proof rule requires that the body of the loop maintain the invariant. 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, nothings needs to be done. The following alternative command serves this purpose.

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

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

```
{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

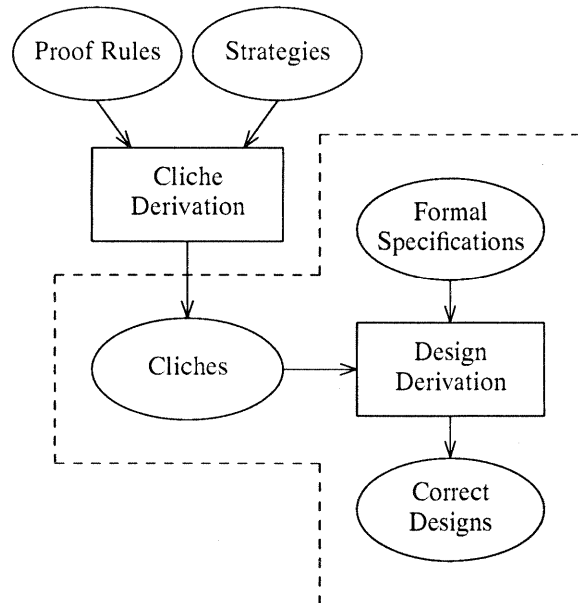


Figure 3. Design Process as Programmed

the loop, we have already proven the design correct. The reset of this example is presented in [41].

As we have described it, the design process consists of a single level: The developer proceeds through a sequence of relatively independent steps to produce a final design. While this model is adequate for performing design walkthroughs, we did not use it in the construction of our simulation program.

3. Simulation Program

Figure 3 shows a pictorial representation 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 correct designs using a library of cliches representing solutions to common programming problems. On the upper level, a cliche derivation sub-process uses strategies and proof rules to construct and verify cliches. These two sub-processes have significantly different complexities: 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 input to the simulation is a pre- and post-condition specification for the unit to be constructed, as well as 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 process 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 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 simulation 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 the Gries/Dijkstra process is beyond the scope of this paper. 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 [4, 5, 31, 35].

We have constructed a running simulation based on the architecture in Figure 3. The program was designed using guarded commands and its correctness rigorously verified [40]. It uses constructs that can be reasonably implemented in most programming languages. A prototype implementation has been written in Prolog that generates a complete design for several small examples including Kemmerer's Library Problem. The prototype follows the formal design very closely; in fact, the implementation can be generated from the design using methods similar to [38, 39]. The implementation is somewhat sketchy, especially the logic manipulation and theorem proving routines; however, it does demonstrate that the design is fundamentally correct.

For example, Figure 4 shows the declarations for the basic types used in the process 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 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 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 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.

```
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 4. Design Process Type Definitions

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. There is a global variable "all_cliches" that holds all the cliches currently known to the system.

The "refines" relation defines the correctness of the design process. A correct program is one that "refines" the specification from which it was produced, and any specification refines itself. 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". In other words, application of any cliche produces a correct design whenever its pre-condition is true.

Figure 5 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" uses the following sub-routines.

```
function optimize(d:dsgn)    : dsgn   ;
function derive_subs(d:dsgn): dsgn   ;
function complete(d:dsgn)    :boolean;
```

"Optimize" is a function that takes a design as input and returns a new one that has improved performance characteristics, while the function "derive_subs" takes a design and, if necessary, derives sub-designs to produce a complete program. Both of these functions preserve the "refines" relation. "Complete" is a function that returns true if all the unknowns in a design have been filled in; in other words, if the design is complete.

```
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 ;
```

Figure 5. Design Process Code

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 simulation process 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 humans, and a simple, mechanistic part performed by a machine. The simplicity of the design derivation process implies that the power the overall process depends on the complexity of the cliches in its library.

4. Cliches

The number of cliches that can be used in the design process is literally 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; and the "conditional_iteration_on_set" cliche generates loops with an embedded conditional. The cliches are presented in more detail and verified in [40].

Figure 6 shows simplified representations of the "simple_assignment" and "simple_if_then_else" cliches. The "simple_assignment" cliche states that the statement "$Var_1..Var_N$:= $Soln_1..Soln_N$" is correct with respect to a pre-condition "Q" and post-condition "R" if "Q" implies "R" with "$Soln_1..Soln_N$" substituted for "$Var_1..Var_N$". The "simple_if_then_else" cliche says that the statement "if B1 → S1 [] B2 → S2 fi" is correct with respect to a pre-condition "Q" and post-condition "$B1 \wedge E1 \vee B2 \wedge E2$" if: "B1" is the logical negation of "B2"; "S1" is correct with respect to pre- and post-conditions "$Q \wedge B1$" and "$B1 \wedge E1$" respectively; and "S2" is correct with respect to "$Q \wedge B2$" and "$B2 \wedge E2$".

The above representations make understanding the cliches simple, and we can see that their correctness follows directly from the standard proof rules [15,26]; however, they do not give much detail about how the cliches are implemented. The implementations of these cliches are complex; at present, suffice it to say that a fairly standard symbol table underlies the simulation, and since our purpose is not to consider the difficulties and technology of theorem proving, such problems are encapsulated within a small number of routines.

```
cliche simple_assignment is

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

end simple_assignment ;


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 ;
```

Figure 6. Simple Cliches

While the above cliches are usable, they are not particularly interesting. "Simple_assignment" presents certain difficulties in that a completely general implementation of its applicability test involves an undecidable problem [26]. Our solution is to use a somewhat less general, but much less expensive test. "Simple_if_then_else" is easy to implement but not very powerful. Since our design derivation process is quite simple, we must use more complex cliches to achieve interesting results.

For example, 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 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. Many different concrete post-conditions can be

```
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 7. *Conditional_Iteration_on_Set* Cliche

unified with this abstract one. 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 loop interates over all the items in "Set". If the item in question satisfies "Cond" then "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, 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.

The cliches presented in this section are fairly simple, but their implementation is much more complex than the derivation engine presented in section three. In line with our two level process model, we have attempted to make cliche application as easy as possible, even at the cost of more effort in cliche construction. Although the framework presented so far is minimal, it has enough power to duplicate some of the designs produced by a human.

5. Example Design Derivation

For example, 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 as follows.

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

Figure 8 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. We can see that the "simple_if_then_else" cliche is applicable to its specification.

```
{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 8. Instantiated Loop Cliche

```
{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}

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

The specification and cliche unify as follows.

```
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
```

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. 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
```

Our simulation program has now automatically produced the hand derived design shown in Figure 2.

6. Summary and Conclusions

We are investigating software design processes using a three part approach. For a design method of interest, we first perform walkthroughs on a number of small problems. Second, we construct a simulation program which duplicates the designs produced by the walkthroughs, and third, we a construct a process program that supports human application of the method. We feel that this approach can increase our understanding of software design processes; for example, what knowledge can be formalized and what activities can be automated.

We have been applying our three step approach to the formal design process developed by Dijkstra and Gries [6, 7, 15]. This method 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. We have currently completed steps one (walkthrough) and two (simulation) on this method [40, 41]. Our experience so far leads us to believe that the cliches underlying the process are more important than is sometimes stated; furthermore, we believe that they can be formalized and automatically applied.

Our simulation of the Gries/Dijkstra process has two levels. At the lower level, the 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.

We have constructed a running simulation based on our two level architecture. The program was designed using guarded commands and its correctness rigorously verified. A prototype implementation has been written in Prolog that generates a complete design for several small examples including Kemmerer's Library Problem. The implementation is somewhat sketchy, especially the logic manipulation and theorem proving routines; however, it does demonstrate that the design is fundamentally correct. Where a choice was necessary, we have traded off logical power and generality for simplicity and efficient execution.

Construction of the simulation has given us considerable insight into the Gries/Dijkstra process. We are now constructing a prototype process program that supports human application of the method. Ideally, we would like it to operate in a standard environment and interact with other tools; for example, in the Arcadia framework [36, 37]. Finally, although the Gries/Dijkstra process is quite valuable, it is not commonly used in industrial settings. We are pleased with our three part approach to process understanding and improvement. Eventually, we would like to apply it to a more widely used technique.

7. References

1. Balzer, R., "A 15 Year Perspective on Automatic Programming", *IEEE Transactions on Software Engineering SE-11, 11* (November 1985), 1257-1268.
2. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 17-29.
3. Bloomfield, R. E. and P. K. D. Froome, "The Application of Formal Methods to the Assessment of High Integrity Software", *IEEE Transactions on Software Engineering SE-12, 9* (September 1986), 988-993.
4. Curtis, B., ed., *Tutorial: Human Factors in Software Development (Second Edition)*, IEEE Computer Society, Washington, D.C., 1985.
5. Davies, S. P., "The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior", *Cognitive Science 15* (1991), 547-572.
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. Fagan, M. E., ''Advances in Software Inspections'', *IEEE Transactions on Software Engineering SE-12, 7* (July 1986), 744-751.
9. Fairley, R., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
10. Feather, M. S., ''Constructing Specifications by Combining Parallel Elaborations'', *IEEE Transactions on Software Engineering 15*, 2 (February 1989), 198-208.
11. Fickas, S. F., ''Automating the Transformational Development of Software'', *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1268-1277.
12. Freedman, D. P. and G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, Little, Brown and Company, Boston, 1982.
13. Freeman, P. and A. I. Wasserman, eds., *Tutorial on Software Design Techniques (fourth edition)*, IEEE Computer Society, Silver Spring, MD, 1983.
14. 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.
15. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.
16. *Proceedings of the First International Conference on the Software Process*, IEEE Computer Science Press, Los Alamitos, CA, October 1991.
17. *Proceedings of the 6th International Software Process Workshop*, IEEE Computer Society Press, Los Alamitos, CA, October 1990.
18. *Proceedings of the 7th International Software Process Workshop*, IEEE Computer Society Press, Los Alamitos, CA, 1991.
19. Johnson, W. L., ''Deriving Specifications from Requirements'', *Proceedings of the 10th International Conference on Software Engineering*, April 1988, 428-438.
20. Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
21. Kaiser, G. E., P. H. Feiler and S. S. Popovich, ''Intelligent Assistance for Software Development and Maintenance'', *IEEE Software 5*, 3 (May 1988), 40-49.
22. Kant, E., ''Understanding and Automating Algorithm Design'', *IEEE Transactions on Software Engineering SE-11, 11* (November 1985), 1361-1374.
23. Kemmerer, R. A., ''Testing Formal Specifications to Detect Design Errors'', *IEEE Transactions on Software Engineering SE-11, 1* (January 1985), 32-43.
24. Lewis, C., P. Polson, J. Rieman and C. Wharton, ''Testing a Walkthrough Methodology for Theory-Based Design of Walk-Up-And-Use Interfaces'', *Proceedings of the ACM Conference on Computer-Human Interaction*, 1990, 235-242.
25. Linger, R. C., H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.
26. Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, New York, 1984.
27. Mills, H. D., M. Dyer and R. Linger, ''Cleanroom Software Engineering'', *IEEE Software 4*, 5 (September 1987), 19-25.
28. Osterweil, L. J., ''Software Processes Are Software Too'', *Proceedings of the 9th International Conference on Software Engineering*, 1987, 2-13.
29. Polson, P., C. Lewis, J. Rieman and C. Wharton, ''Cognitive Walkthroughs: A Method for Theory-Based Evaluation of User Interfaces'', *International Journal of Man-Machine Studies*, in press.
30. Rich, C. and R. C. Waters, eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman Publishers, Los Altos, CA, 1986.
31. 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.
32. 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.
33. 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.
34. Smith, D. R., ''KIDS: a Semiautomatic Program Development System'', *IEEE Transactions on Software Engineering 16*, 9 (September 1990), 1024-1043.
35. Soloway, E. and K. Ehrlich, ''Empirical Studies of Programming Knowledge'', *IEEE Transactions on Software Engineering SE-10*, 5 (1984), 595-609.
36. 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.
37. 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'', *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1988, 1-13.
38. 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.
39. 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.
40. Terwilliger, R. B., ''A Process Program for Gries/Dijkstra Design'', Report No. CU-CS-566-91 , Dept. of Computer Science, U. of Colorado at Boulder, December 1991.
41. 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.
42. Weinberg, G. M. and D. P. Freedman, ''Reviews, Walkthroughs, and Inspections'', *IEEE Transactions on Software Engineering SE-10, 1* (January 1984), 68-72.
43. Wing, J. M., ''A Study of 12 Specifications of the Library Problem'', *IEEE Software 5*, 4 (July 1988), 66-76.
44. Yourdon, E. N., *Structured Walkthroughs*, Yourdon Press, New York, 1989.
45. *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987.

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.